



Using VRML

<http://kickme.to/tiger/>

Special Edition Using VRML

[Introduction - A Road Map](#)

Part 1 - The Internet

- [Chapter 1 - The Internet Its History and How It Works](#)
- [Chapter 2 - The Languages of the Web](#)

Part 2 - From Vr to VRML

- [Chapter 3 - What Is VR?](#)
- [Chapter 4 - How Does VR Work?](#)
- [Chapter 5 - Networks and Distributed VR](#)
- [Chapter 6 - VR and VRML](#)

Part 3 - An Introduction to VRML

- [Chapter 7 - The Mechanics of VRML](#)
- [Chapter 8 - What Do You Need to Use VRML?](#)
- [Chapter 9 - VRML Browsers](#)

Part 4 - Creating a VRML Environment from Scratch

- [Chapter 10 - Laying the Foundation Planning Your VRML Environment](#)
- [Chapter 11 - VRML Standards and Practices](#)
- [Chapter 12 - Putting Up Walls](#)
- [Chapter 13 - Furnishing Your Room](#)
- [Chapter 14 - Adding Realism to Your VRML Environment](#)
- [Chapter 15 - Linking Your VRML Environment to the Internet the WWWInline and WWWAnchor Nodes](#)
- [Chapter 16 - Setting Up the Camera](#)
- [Chapter 17 - Improving the Performance of Your VRML Environment](#)
- [Chapter 18 - Opening Your House on the World Wide Web](#)

- [Chapter 19 - Proposed Additions and Changes to VRML 1.0](#)

Part 5 - Learning From the Pros: Sample VRML Environments

- [Chapter 20 - What Is VRML About?](#)
- [Chapter 21 - Using Perl and the LOD Node in VRML Environments](#)
- [Chapter 22 - Some Hand-Crafted Worlds](#)
- [Chapter 23 - A Guide to Other Sample Environments on the Web](#)

Part 6 - Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities

- [Chapter 24 - An Overview of 3D Modelers](#)
- [Chapter 25 - An Overview of Worldbuilding Toolkits](#)
- [Chapter 26 - VRML Utilities](#)

Part 7 - VRML 2.0 and Beyond

- [Chapter 27 - Proposed Changes for VRML 2.0](#)
- [Chapter 28 - Behaviors](#)
- [Chapter 29 - What Comes Next?](#)

Part 8 - The Principles of Artificial Environment Design

- [Chapter 30 - The Zen of Virtual Reality](#)
- [Chapter 31 - Architecture, Design, and VRML](#)

Part 9 - Building a Better Mousetrap: programming Your VRML Browser

- [Chapter 32 - Writing VRML Software](#)
- [Appendix A - Essential Math for Using VRML](#)
- [Appendix B - The VRML 1.0 Specification](#)

- [Appendix C - Proposed Changes for VRML 1.1](#)
- [Appendix D - Proposed Changes for VRML 2.0](#)
- [Appendix E - C Source Code](#)
- [Appendix F - The VRML CD](#)

| [Buy This Book](#) |

To order books from QUE, call us at 800-716-0044 or 317-361-5400.

© 1996, QUE Corporation, an imprint of Macmillan Publishing USA, a Simon and Schuster Company.

Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.

Introduction

A lot of things happened in 1995. With the end of the millenium looming, war broke out in some parts of the world, peace broke out in others. Science and medicine made great advances. Some great and not-so-great works of literature were created. And generally, society moved forward in some areas and back in others.

From the point of view of a web-surfer, however, 1995 was marked by one particularly important event: the introduction of the VRML standard.

VRML is basically a language, very similar in some respects to other languages used by computers. If you look at a ".wrl" file, you'll see that it's ordinary text with keywords and curly braces and so forth. We'll be analyzing the syntax in excruciating detail later in this book, but at this point we're going to concentrate on what Hollywood moguls call the "high concept."

The "high concept" is simply this: VRML is a language for describing worlds.

We're not talking about text, or static images, or even movies; we're talking about real, three-dimensional worlds that you can move around in and explore. When you call up a VRML file, you're not retrieving informationóyou're entering a world.

The effect on the online community will be profound. Virtual environments will begin to appear all over the net. Some will be realistic, some abstract. Some will be functional, others creative. Some will be informative, some fanciful. But they will all be "real" in a way that text and pictures by themselves can never be.

In time, the objects in those worlds will acquire the ability to move and change and respond to human interaction. Eventually the worlds will be populated by multiple users at the same time, and the Web will begin to develop into a huge, interactive, participatory virtual environment. You won't just be looking at information. You'll be moving around it and through it. You'll touch it and manipulate it.

It will be nothing short of a revolutionary change in the nature of online communication and human interaction. You can be part of that change. This book is designed to give you the tools, the skills, the resources, and the motivation to be part of the process.

A Road Map

Virtual worlds are often designed with multiple "entry points," and we have designed this book in much the same way.

If you know very little about the Internet, you may want to take a look at Part I, in which Stephen takes us on a brief swim in the alphabet soup of IP, TCP, PPP, and SLIP. By the time he's done, you'll have a clear idea of what the Internet is, where it came from, and where it's going.

If you're a little uncertain about the "VR" part of VRML, you may want to take a look at Part II. In it, Bernie examines the nature of virtual reality, some of the technology that underlies it, and the relationship between VR and VRML. In that same section, Stephen gives a quick guided tour of existing online virtual worlds.

Part III provides our first real introduction to VRML itself. Bernie provides a brief introduction to the mechanics of the language, then Stephen discusses the hardware and software requirements and examines some of the available VRML browsers.

If you've already used VRML a bit and want to get into the nitty-gritty details of the language, then Part IV is the entry point for you. We've taken a two-pronged approach: Bernie describes the language itself (Chapters 11, 14, 16, 17, and 19), while Stephen works through some practical examples (Chapters 10, 12, 13, 15, and 18).

Part V examines half a dozen virtual worlds, contributed by a variety of authors.

You can't do much without tools, and Part VI is an overview of some of the better ones that are available. Stephen explores modelers and world-building packages, and Bernie examines utilities for working with VRML.

Part VII is Bernie's glimpse into the VRML crystal ball, with predictions of what the future may hold. Beware! The information there is purely speculative, but it's the best we have as of mid-October 1995. Caveat reader!

In Part VIII, Stephen and Bernie share some of their individual philosophies about VRML; if you're more interested in ideas than in nodes, then dive right on in.

Part IX, by contrast, is strictly for programmers. If you're not a programmer, imagine that there's a great big "do not enter" sign over it. However, if coding in C is old hat for you, you'll find that Bernie has a few useful techniques for generating VRML worlds.

Finally, the Appendixes provide a handy reference to VRML, the changes that have been proposed to it, the math behind it, and references to other resources.

Assembling this book has been a real challenge, but we feel the results speak for themselves. We hope you enjoy reading it as much as we've enjoyed writing it.

If you have any questions or comments about anything in the book, feel free to drop us a line. Bernie's e-mail address is broehl@sunee.uwaterloo.ca and Stephen's is engl5105@nexus.yorku.ca. Although we don't have time to help you with your world-building (which would be a full-time, unpaid job!) we'll be happy to discuss the book with you and help you find other useful resources.

See you in cyberspace!

For technical support for our books and software contact support@mcp.com

Copyright ©1996, Que Corporation

Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



1-The Internet Its History and How It Works

In the last few years, the Internet has become one of the most discussed computer technologies. Every day we hear about its content, its future, and how we have to get access to it. Businesses are scrambling to mark their presence on "the Net." And for individuals and educational institutions, the Internet has become the "thing" to be on. If it were to continue growing at the current rate, the entire population of the world would be on the Internet by the Year 2004!

Calling it a "network of networks" would be true if it stuck to its origins as ARPANET and NSFNET. But the Internet has become something greater than the sum of its connections. Certainly no one could have predicted how its move from a military computer network to a public and commercial network would affect the way we think about information and communication.

This chapter provides an overview of the Internet and its offspring—the World Wide Web. It examines the following issues:

- What are the origins of the Internet and the World Wide Web?
- How do they work?
- How do I get connected to them?
- What affects traffic on the Internet and the WWW?

A Brief History of the Internet

During the 1960s, the United States Department of Defense saw a need to maintain some kind of communications network in the event of a nuclear attack. The RAND Corporation, a U.S. military think-tank, proposed a decentralized communications network as a solution. This network would use a series of interconnected nodes—each with the ability to originate, pass, and receive messages. These messages would in turn be divided into smaller chunks called packets. Each packet would be given a specific address with specified source and destination nodes, and would travel through the network on an individual basis. The particular route that any packet took would be unimportant. If a node was destroyed, the packets would continue to travel through another node.

A packet is a series of bytes that are sent in an order set by a particular format. It can include information about packet size, packet type, and error detection.

The Defense Department's Advanced Research Projects Agency (ARPA, now known as DARPA) decided to expand this idea by making the nodes computers rather than communications stations. ARPA believed that it would be cheaper to develop a network of computers for its sponsored organizations than to purchase a new computer for each of them. In 1969, the first "Interface Message Processor," a predecessor to today's routers, was installed at the University of California in Los Angeles (UCLA) and ARPANET came into being.

ARPANET included some of the services that are important features of the present day Internet such as File Transfer Protocol (FTP), remote login (TELNET), and electronic mail. Electronic mail "was added to the File Transfer Protocol as an afterthought" (Vittal, J., Crocker, D., and Henderson, A. [1977]. Proposed Official Standard for the Format of ARPA Network Messages. May 12, Requests for Comments 724), and its format would not be standardized until the late 1970s. ARPANET would eventually become outmoded and was finally shut down in 1989.

At the same time that ARPANET began to grow, companies like Xerox were developing local area network (LAN) technology. The one with the most impact was Ethernet, which was invented by Xerox and then jointly developed by Xerox, Intel, and DEC. Ethernet allowed different computers to be linked together. The first version had a theoretical maximum speed of 3 Mbps and later versions would allow up to 10 Mbps. These speeds allowed networks to have distributed file systems—a feature that was not practical on ARPANET's 56 Kbps standard.

Researchers at ARPANET began to see that it could be useful to connect LANs to ARPANET. In order to accomplish this, a standard protocol had to be developed to connect different types of equipment to each other. Thus, the Transmission Control Protocol (TCP) and the Internet Protocol (IP) were born. In 1983, the growth of the Internet was given a boost with the release of the 4.2BSD version of the UNIX operating system, which included an implementation of the complete TCP/IP protocol suite. This release, which was developed using public funds, was provided for the cost of distribution.

In 1984, the National Science Foundation began to provide funding for the establishment of research and academic networks throughout the U.S. In 1986, it began to link those networks into a high speed network using the TCP/IP protocols developed for ARPANET. The result was NSFNET. NSFNET became the backbone of the Internet. Similar networks were created in other countries and the Internet was on its way.

[Fig. 1.1](#)

World map of countries with Internet access. Copyright _ 1995 Larry Landweber.

The Commercialization of the Internet

Although the Internet was growing by leaps and bounds during the 1980s and early 1990s, commercial applications were restricted by the NSF Acceptable Use Policy (AUP). The general principle of this policy stated that NSFNET's backbone services were provided to support research and education in U.S. institutions. Companies could use NSFNET only when they were engaged in "open, scholarly

communication and research." Any use of the network for for-profit activities and extensive use for private or personal business was considered to be unacceptable.

The term backbone refers to a series of high-capacity links that carry a large volume of network traffic. It usually refers to the main links in a network. In the United States, the national backbones included ESNet (Department of Energy), Milnet (Department of Defense), NSI (NASA), TWB (DARPA), and NSFNET. When people talked about the Internet backbone in the U.S., they were usually referring to NSFNET (see fig. 1.2). In Canada, the national backbone is called CA*Net. Europe has several multinational IP backbones and Japan has three IP backbone networks. Australia and New Zealand also have their own.

In April 1994, the NSF began to shift its funding from directly supporting the Internet backbone to subsidizing regional access to backbone services. This change meant that connections to Network Access Points (NAPs) could be provided on "an AUP-free basis." Today, companies like MCI are beginning to take over the backbone. On 30 April 1995, the NSFNET Backbone Service was successfully switched to the NAP architecture and NSFNET officially came to an end.

The commercialization of the Internet has a number of important implications for all usersóparticularly concerning access. As commercial backbones take over, the financial considerations of the Internet will become larger and larger. Universities and companies that used to spend between \$50,000 and \$100,000 per year to maintain their Internet hardware might find their costs rocketing to over \$500,000.

As a graduate student, I receive free access to the Internet in order to do my work. Will I and other students have to pay for that access in the future? And how will these changes affect people who are on commercial services? Any increases in the cost of setting up and maintaining the Internet will hit the consumer. Some proposals envision a per-bit cost for using it. But no matter what changes are made, the result will see the end-user having to shoulder a bigger share of the costs of the Internet.

How Does the Internet Work?

In order to understand how the Internet works, you need to understand that the Internet is not a homogenous entity. Indeed, it is something of a miracle that it works at all. Its structure is such that sending an electronic message across the street may require that data to travel thousands of miles. Moreover, the machines that are hooked up to it were never designed to communicate with each other directly. And yet the Internet worksódue in large measure to the standards that were created for it.

Networks, Networks, and More Networks The Structure of the Internet

If the Internet were to be defined strictly in terms of how it is put together, then calling it a "network of networks" would be appropriate. Another way to understand the Internet is to see it as layers of networks that increase the amount of connectivity as you move up (see Fig. 1.2). As you move up to higher levels, the number of networks that can be directly accessed rises.

[Fig. 1.2](#)

The structure of the Internet.

The starting point for everyone on the Internet is an Internet Service Provider (ISP). An ISP can be a commercial provider, a corporate computer network, a school, college, university, or the government. Some users may be connected on a LAN; some may get access using a modem and a SLIP or PPP connection; and some may have access to a mainframe computer shell account using a modem.

All ISPs are connected to the Internet through other ISPs. On the lowest layer, one local ISP is connected to another local ISP. On the next level are the ISPs that are the access points to regional backbone networks. These networks cover specific areas: usually states or provinces. For example, ONet links universities, community colleges, private companies, and government agencies in the province of Ontario, Canada. Most local ISPs will be connected to these regional access points. At the next level are the national backbone access points that are the main links of the Internet. Like the regional backbones, these national access points are ISPs and are connected to regional access points. They also may be connected to local ISPs. Finally, the access points of one country's national backbone network are connected to the access points of another country's national backbone, thereby making the Internet an international network.

When you look at the structure of the Internet, it becomes clear that getting information from one place to another is not a simple task. Information does not travel directly from the source to the receiver. Instead, it must travel along a series of nodes that do not necessarily make sense at first glance. For example, say that you want to access a VRML environment from a site just across town. However, the source site is not connected to the same regional network as your Internet access provider. Your request must then be routed to the national backbone node. If the source's network is not linked to the national backbone, then your request must be routed to a site that is connected to it. Your request may have to travel from the national backbone, across the border, and move through a number of different national and regional nodes before it reaches its destination.

Routers and TCP/IP

How your request finds its way to the right site and how that information finds its way to you is determined by two elements: routers and the TCP/IP protocols. Routers (also referred to as gateways) are like telephone operators. They connect one network to one or more other networks. They also have routing tables that determine how information from your network should be sent through the Internet. Routers have an IP address for each link: for example, a router with a PPP link and two Ethernet connections would have three IP addresses.

When a packet arrives at an IP router, the router decides where to send it next. The way that a router makes its decision can vary. Some simply send packets in a set sequence. Some randomly alternate the routes by which they send a packet. More sophisticated systems measure the traffic and send packets through the least busy lines.

Where the packets have gone is recorded on a routing table. Regional networks handle their own traffic, reroute their own packets, and update their routing tables. In this way, the amount of routing information that is updated on national backbones can be kept to a minimum.

The secret brew that makes the Internet work is the TCP/IP set of protocols. The Transmission Control Protocol (TCP) and the Internet Protocol (IP) are the main components of the Internet, and a part of the Internet's growth is due to their compatibility with so many networking technologies, computers, and

operating systems. IP is the component that moves packets of data from one node to another. The TCP component is what verifies the delivery of data from your computer to another computer.

IP assigns a unique number to every networked computer in the world. If you watch your SLIP or PPP program when it connects to your ISP, you will notice that you are assigned an IP address. This address identifies both the network you are on and your computer, and is made up of four numbers separated by periods. There are three classes of IP addresses. Class A addresses use the first number to identify the network and the last three numbers to identify the computer. Class B addresses use the first two numbers to identify the network and the last two to identify the machine. Class C addresses (the most common ones available) use the first three numbers to identify the network and only the last number to identify the computer. Your IP class is related to the number of computers on your network.

For example, the IP address of the Nexus system at York University, which is the site of my account, is 130.63.236.20. But since it is easier to remember a name rather than a number, the Internet also lets you use a Domain Name. If routers are the operators of the Internet, then the Domain Name Service (DNS) is its telephone book. It is a database of Domain Names that determines the IP address of the data's destination. All Domain Names are registered with a national Domain Registrar or with InterNIC Registration Services. In this way, you can send an e-mail message to me using either my username and the IP address for Nexus (engl5105@130.63.236.20) or my username and the Domain Name for Nexus (engl5105@nexus.yorku.ca). I assure you that both will work, so please don't send me a message to test if they do.

Modems, Direct Links, and Connecting to the Internet

There are two things you need in order to access the Internet: an ISP and a way to connect to that provider. Once you have found an ISP, you need to find a way to connect to that provider. The most common ways of making that connection are to use a modem over a regular telephone line or to get direct access to the Internet.

Connecting to the Internet via a Modem and Regular Telephone Lines

The most common method of making this connection is by using a regular phone line and a modem. A modem (which stands for MOdulator/DEModulator) is needed to convert the digital signals from your computer into analog signals that can be transmitted over regular phone lines, and to convert the analog signals sent by the remote computer's modem back into digital ones.

Modems are available in both external and internal versions. External modems have their own case, separate power supply, and connect to your computer with a serial cable. Internal modems are cards that plug directly into a bus slot in your computer and use your computer's power supply.

Whether you use an external or internal modem is really a matter of personal choice rather than a matter of performance. Internal modems are usually less expensive because they do not need their own cases and power supplies. They do not need extra cables and take absolutely no desktop space. External

modems provide displays that are useful when a problem arises, and they can be easily moved to a different computer.

Troubleshooting

Troubleshooting Modems

Have you ever wondered if all those flashing lights on your modem have a purpose?

The answer to this question is, in fact, yes: those lights tell you a great deal about the status of your communications session. Here is a list of the most commonly used lights and what they mean:

HS	High Speed. This light means that your modem is ready to use its highest speed.
AA	Auto Answer. If this light is on, it means that your modem is ready to answer the phone.
CD	Carrier Detect. This means that your modem is connected to another modem.
OH	Off Hook. This light comes on when your modem connects to your phone line to dial out.
RD	Receive Data. This means that your modem is receiving data from the other modem.
SD or TD	Send Data or Transmit Data. This light indicates that your modem is sending data.
TR	Data Transmission Ready. This light comes on when your modem is ready to transmit data.
MR	Modem Ready/Test Mode. This light comes on when you turn on your modem, or when it drops its speed to a lower setting.

Another important consideration when choosing a modem is its speed. The fastest modems available for regular telephone lines can transfer data at rates of 28,800 bits per second (or 28.8 Kbps). Without data compression, a 28.8 Kbps modem is up to twelve times faster than a 2400 Kbps modem; with data compression, it can be 25 to 50 times faster. To access graphics and VRML environments (especially if they use texture maps) on the World Wide Web, you need to have at least a 14.4 Kbps modem. However, if you want to download complex VRML environments, we recommend that you have at least a 28.8 Kbps modem.

It's important to note that there is a difference between bits and bytes. When you hear about a computer program being 500 K or 1.2 megs, those numbers are being given as bytes. A byte is made up of 8 bits. Modem speeds are given in bits per second. A 28.8 Kbps modem, therefore, can transfer up to 3600 bytes, or 3.6K per second. To put it another way, it would take a 28.8 Kbps modem at least 4.63 minutes to download a one megabyte file. In contrast, it would take a 2400 Kbps modem over 55 minutes to download the same file. Moreover, all these times can be affected by the quality of your telephone line. Figure 1.3 shows the number of minutes it takes to download a one megabyte (1 MB) file at different modem speeds.

Fig. 1.3

The number of minutes to download a one megabyte file at different modem speeds.

Although 38,400 Kbps modems have appeared on the market, the ability of regular telephone lines to handle high-speed data transfers is still in question. The bandwidth (the amount of data that can be

transmitted) available on your phone line may not allow your modem to operate at its maximum speed. There are questions whether a regular phone line can actually accommodate a 28.8 Kbps connection. You may find that you are actually limited to 16.8 to 19.2 Kbps. Before investing in a 28.8 Kbps or higher modem, check with your telephone company to see if the phone lines are capable of high-speed data transmissions.

When you buy a modem, make sure that it conforms to CCITT standards. CCITT stands for the Comité Consultatif International Téléphonique et Télégraphique: the United Nations body that is responsible for communications standards. The CCITT modem standards always have "V." and number (for example, V.34 which is pronounced "vee-dot thirty-four"). Sometimes, you will see the words "bis" or "ter" placed after the number. bis is French for "second item in the series" and ter means "third item in the series." Therefore "V.34 bis" means that the modem you are looking at is using the second version of the V.34 standard. The main standard for 28.8 Kbps modems is V.34 and for 14.4 Kbps modems it is V.42 bis. Older 28.8 Kbps modems may use the V.FC standard which is less robust than V.34.

You must also make sure that the modem you are going to buy is compatible with the AT command set developed by Hayes. Most communications software uses this standard, and you will run into problems if you buy a modem that does not use it.

The Basic Hayes (AT) Command Set

ATZ	This command resets the modem's internal settings to what is stored in nonvolatile RAM (NVRAM). If your modem is working, it should respond with OK. This is a good way of checking your modem connection.
AT&W	This command writes your current settings into nonvolatile RAM and lets you recall them by using the ATZ command. You can reset the NVRAM to the original factory settings by typing in AT&F.
ATS0=1	This sets the modem to answer after the first ring.
+++ATH	This command tells your modem to hang up during a session.
ATDT	This command dials out on a touch tone phone. It is followed by the phone number you wish to dial: for example, ATDT 1416555555. If you have a call waiting feature on your telephone line, you will need to turn it off before dialing into your ISP. In many phone areas, you type in *70w before the number to turn this feature off: for example, ATDT *70w1416555555. However, the prefix used to turn off call waiting is system-specific. Therefore, you should check with your phone company to get the right one.
ATDP	This command dials out on a pulse or rotary phone. It is followed by the phone number you wish to dial: for example, ATDP 1416555555. If you have a call waiting feature on your telephone line, you should disable it as described above.
ATM0	This command turns your modem speaker off (especially useful if you are connecting in the middle of the night and don't want to wake up anyone who is close by). You can turn your modem speaker back on by typing ATM1.

A recent development in 14,400 Kbps modems is the introduction of RPI modems. These devices use a special Rockwell chip that sets up its error correction and data compression in software rather than

hardware. The advantage of these modems is that they are cheaper. The disadvantage at the time of this writing is that the software drivers have displayed a number of problems that may prevent you from using the modem. So remember: caveat emptor.

SLIP and PPP

Two important developments for users connecting to the Internet using a modem are the Serial Line Interface Protocol (SLIP) and the Point to Point Protocol (PPP). Both allow users to connect to the Internet.

If your ISP allows you to use both SLIP and PPP software, choose PPP. SLIP is a less powerful protocol than PPP. SLIP does not route and cannot error-correct IP packets. PPP supports error correction and allows multiple protocols. However, SLIP is easier to set up on your computer than PPP.

In order to use SLIP/PPP, your ISP must provide SLIP/PPP access. You must also have software that supports SLIP/PPP. For Windows 3.1, you can use programs like Trumpet Winsock and Chameleon. Windows 95 has support built in for Internet connectivity. Macintosh users can use MacSLIP or MacPPP. Moreover, these programs must be configured for your ISP. The information you will need to have includes the IP address for your ISP, your ISP's domain name, and its name server (see fig. 1.4).

Fig. 1.4

The configuration screen for Trumpet Winsock.

If your ISP does not have a preconfigured SLIP/PPP package for your computer, you can obtain nonconfigured copies via anonymous FTP. For Windows, you can get Trumpet Winsock at the SimTel FTP mirror sites (oak.oakland.edu and wuarchive.wustl.edu). Look in the directory /win3/winsock for twsk*.zip (* will be the version number). Macintosh users can get a copy of MacPPP at merit.edu. Look in the /pub/ppp directory for macppp2.0.1.hqx.

Digital Lines and Direct Links

For some people, "fast" is never enough. They want to go faster. And with the growth of the World Wide Web, the prevalence of graphics and video, and the growing interest in VRML, the need to have faster connections to the Internet is becoming an important issue. One of the key elements in achieving this goal is the digital phone line.

Digital phone lines permit much higher transmission speeds than conventional telephone lines. One type of digital line service is the Integrated Services Digital Network (ISDN). As the hardware improves, ISDN will eventually allow up to 128 Kbps on a single line.

ISDN is significantly better than conventional modem connections. The time it takes to connect to an ISP with a modem and a regular telephone line can be as high as 45 seconds. With an ISDN line, the time is only two to three seconds. With a conventional modem and telephone line, the maximum bandwidth that can be achieved is around 28.8 Kbps. With an ISDN line, the bandwidth can be as high as 128 Kbps. Moreover, sending data over a digital line is more reliable.

But while ISDN promises to bring a faster standard for connectivity, other technologies are being

developed that will eclipse even these rates. T1 and T3 digital lines have bandwidths of 1.544 Mbps (that is 1,544,000 bps) and 45 Mbps respectively. These technologies are currently available only to major providers and are extremely expensive.

In fact, one of the main reasons that digital phone lines have not caught on just yet is the cost. Setting up an ISDN connection ranges up to \$2,500. Moreover, these digital services are not available in all areas and a single standard has not yet been established for their use. Little wonder that another meaning for ISDN is "I Still Don't Need it." However, it will not be long before digital phone lines become more affordable for the average person.

However, AT&T Paradyne has announced that it has developed a new digital transmission technology called GlobeSpan'. GlobeSpan is a digital communications platform that allows subscribers to have simultaneous voice and data transmissions at T1 to E1 speeds over a single, copper phone line. It is described as a symmetric-digital-subscriber-line (SDSL). The estimated cost of using this line is less than that of other digital lines because of the reduced need for new hardware. The bidirectional speeds for GlobeSpan SDSL will range from 128 Kbps to 2.048 Mbps. The initial availability date for this technology was December 1995.

The World Wide Web What It Is and How It Came to Be

If there is one reason for the need for faster access times on the Internet, it is the World Wide Web. In the last few years, telling someone that you are "surfing the Net" has become synonymous with "being on the Web." The World Wide Web (also known as WWW, W3, or simply as "the Web") has made accessing information on the Internet much easier for the average person.

The Web is also the Internet service that has seen the greatest growth in the last few years. In a recent study, the Web was listed as the service that had the largest amount of traffic on the Internet. In the Third WWW User Survey conducted by the Graphics, Visualization, and Usability Center (GVU) at the Georgia Institute of Technology, 41% of the respondents estimated that they spend between 6 and 10 hours per week on the WWW, and 21% stated that they spend between 11 and 20 hours per week.

The Graphics, Visualization, and Usability Center publishes information and statistics on the Internet. The URL for the Gvu Home Page is <http://www.cc.gatech.edu/gvu/gvutop.html>. Its NSFNET Backbone Statistics Page is at <http://www.cc.gatech.edu/gvu/stats/NSF/merit.html>. Its WWW Users Survey Home Page is at http://www.cc.gatech.edu/gvu/user_surveys/User_Survey_Home.html.

Table 1.1

Service Name	Port	Packet Count	% Pkts	Byte Count	% Bytes
WWW	80	12694282100	21.443	3518742182400	26.250
Other	999	11702491800	19.767	2441774711050	18.216TCP/UDP ports
FTP	20	8301694400	14.023	2886742144950	21.535
NNTP	119	480677470	8.119	1160496874800	8.657

April 1995 Statistics (Visualization and Usability Center, Georgia Institute of Technology).

What Is the World Wide Web?

Strictly speaking, the Web is a system for creating and browsing hypertexts on the Internet. It is made up of documents comprised of text, images, and/or sounds, as well as links to other documents and servers. A Web document can also contain links to other Internet services such as FTP, Gopher, Archie, and Telnet.

A better way of looking at the Web is to see it as the interactive, graphic interface for the world's largest information source. You can search for information on any subject, from baking cookies to smashing atoms. You can learn about the latest research on artificial intelligence or find out what Homer Simpson was talking about when he claimed to be "the first non-Brazilian to travel backwards through time." And if you are looking for someone to fill a job, you can look at the resume of someone half a world away. Moreover, to access this information you don't have to remember commands or special codes; you just need to point and click to access the information you want.

And finding one information site on the Web will lead you to other information sources, often referred to as Web documents or webpages, that could be of interest to you. You can tour an art exhibition in Paris (without paying for a plane ticket, hotel, and the museum fee). And while you are at the gallery, you discover that there is a tour of Paris webpage for you to explore. While looking for information about VRML, you can discover a virtual reality model of Stonehenge or a 3D model of an enzyme. And with each new Web page that you find, dozens more are made available for you to explore.

How Did the Web Get Started?

The Web began in 1989 as an initiative from the European Laboratory for Particle Physics (also known as CERN). Tim Berners-Lee proposed a set of protocols for an information distribution system on the Internet to be used among research groups exploring high energy physics. These protocols were eventually adopted by other organizations.

Eventually, the need to develop standards for the Web spurred the formation of the W3 Consortium. This organization oversees the development of new standards such as new versions of HTML. It is presently led by CERN, the Massachusetts Institute of Technology, and the French National Institute for Research in Computer Science and Control.

In 1993, the National Center for Supercomputing Applications (NCSA) released a Web browser for the UNIX, Windows, and Macintosh platforms. This browser, called Mosaic, opened the Web to a wider audience and was the catalyst that helped the phenomenal growth of the Internet.

And it hasn't stopped since.

How Does the Web Work?

The World Wide Web works on a client-server model. The client, or user, sends a request to a server. The server takes the request, finds the appropriate information, and sends it back to the client. On the Web, information is requested by using a Uniform Resource Locator (URL).

The URL is like an address. It identifies the Internet service you are requesting (the scheme), its location (the host.domain and, if necessary, the port), the directory on which the data resides (the path), and the name of the data. For example, to get the home page for Early Modern Literary Studies, an academic journal on Renaissance literature that is published on the Web, I would send a request for <http://unixg.ubc.ca:7001/0/e-sources/emls/emlshome.html>. This request means that I am accessing a file called emlshome.html using the HyperText Transfer Protocol (HTTP) at the host.domain unixg.ubc.ca (which is the University of British Columbia) through port 7001, and that the directory path the file is in is /0/e-sources/emls.

While it is possible to access the Web on a shell account using telnet, its main feature is its ability to incorporate graphics and sound. And to see and hear on the Web, you need to use a Web browser like Mosaic or Netscape. For example, I can send a request to see the homepage of the VRML Repository (<http://www.sdsc.edu/vrml>). Figure 1.5 shows how this webpage appears using Netscape.

Most Freenets provide Internet access using shell accounts that require you to use a terminal emulation program. This limits your access to a variety of services. However, if you wish to try using the WWW on your shell account, telnet to fatty.law.cornell.edu and log in as www. You can also access the WWW by e-mail. For more information, send a message to listserv@info.cern.ch. Leave the subject line blank and type help in the body.

[Fig. 1.5](#)

The VRML Repository's home page.

Reading through this Webpage, you will notice that parts of the text are underlined and are a different color. This highlighted text is a link to another webpage or an Internet service. If you want to see the webpage that is linked to the highlighted text, move your cursor over the text and click your mouse. For example, when you read through the VRML Repository home page, you will find a table of contents. One of the items on that list is "Documentation and Specifications." To look at that webpage, move your cursor over that line and click the mouse (see Fig. 1.6).

Your browser sends a request to the VRML Repository's server and the server returns the appropriate page to your computer (see Fig. 1.7).

As well as accessing text, you can view video clips from forthcoming movies and television programs, listen to new and alternative music releases, and download software from the Web. You can also fill out a user survey or vote on your favorite operating system and send it through the Web. And you will be able to do even more as new applications are developed—including the ability to "walk" through a 3D environment using VRML.

[Fig. 1.6](#)

Choosing a link.

[Fig. 1.7](#)

The VRML Repository's Documentation and Specifications page.

HTTP and httpd

An important reason why the Web is able to work with so many different computer systems is the HyperText Transfer Protocol (HTTP). HTTP is the standard set of commands that allows Web browsers and Web servers to communicate with each other. Each time you request a Web document, your browser sets up an HTTP transaction that will tell it how to read the data coming from the server. Without HTTP, your browser would not know whether the incoming data was an HTML document, a VRML environment, or a program file being downloaded by FTP.

All HTTP transactions basically work in four parts. The first part is the connection. The client (your computer) establishes a TCP/IP connection with the server (the computer that holds the Web page you want to see). This connection is made when you type in the URL of the Web page you are requesting.

The second part of an HTTP transaction is the request. It includes information about the type of request, the URL, and information on how the server is to handle the request. For example, if you fill out a form (see Fig. 1.8) and send it back to the server, the server must know that the data you are sending is a form and must be processed before sending a response.

Fig. 1.8

A sample Web form.

The third step is the response. The Web server reads the URL and other information included in the request. It then locates the appropriate file and performs whatever tasks are necessary: process a form, FTP a file, or simply send the HTML code for a webpage.

The final step is disconnection. Once the response has been sent from the server to the client, the server breaks the TCP/IP connection. The entire process begins again when you type another URL into your browser, or when you click on a link in a Web document.

You will also see references to something called httpd. httpd is not a version of HTTP. It stands for HTTP Daemon. The httpd is a program that runs on the Web server and looks for any HTTP requests that are sent to it. It then performs the necessary operations to fulfill the request.

To learn more about HTTP and httpd, go to the W3 Consortium's Website at <http://www.w3.org/>.

MIME Types

The Multipurpose Internet Mail Extension (MIME) was developed in 1992 and was originally designed to allow the transfer of different types of data files by e-mail. However, it quickly became an important component of the Web. It now provides a common method of transferring data from your computer to a server and vice versa, and tells both the server and your computer what to do with it.

Different types of data files have their own MIME types. This means that you can set up your Web browser to either perform a task itself (for example, to display a GIF image file) or to run a specific program. For instance, when you download a VRML file, your browser will determine that it must run your VRML browser. In contrast, downloading an Adobe Acrobat file will cause your browser to call up your Acrobat reader.

If your browser does not recognize a file type, you can add the necessary information in the Helper Applications dialog box of your browser. To reconfigure your Web browser to run your VRML browser, see "An Overview of VRML Browsers," Ch.9.

Why Is Everything So Slow Today? Downloading and Internet Traffic

Living in Toronto, Canada, I have noticed that whenever I access the Web on a weekday afternoon, I am spending most of my time waiting for files to download. And if I try to FTP a file from one of the SimTel archives, I usually can go off for a nice, leisurely coffee break and come back just in time to see the downloading finish. And yet I have a 14.4 Kbps modem set to use data compression and a relatively clean, urban telephone line. Why should I have to wait so long?

The fault is not that of my modem or my telephone line. It is the amount of traffic that is being bounced around the Internet at that particular time. For one thing, my ISP does not have a single router and connection to the Internet set up just for me. Both are shared by myself and any number of other users simultaneously. If a large number of users are on the system at the same time (a pretty likely occurrence at a university) and most are trying to access the Internet at the same time (another pretty likely occurrence), then the router is going to be able to allow me only a small fraction of its time to actually connect to the Internet.

NOTE

One way to improve your access to your ISP is to use a dedicated line. Many ISPs will sell this type of line to you. And while it is more expensive, it does guarantee that you will have access to your ISP when you want it.

And it is not always a single source that is the problem. Other ISPs providing the regional and national access points may also be burdened with a large volume of traffic at their sites. Moreover, the regional backbone sites have to handle the traffic being sent by the local sites connected to them, and the national backbone sites have to contend with all the regional traffic being routed through them. And if you are connecting to a server on the other side of the continent, your request will have to move from local point to regional point to national point to another national point to another regional point, etcetera. And at each point, the router may be working at capacity and holding up your request.

The result of these bottlenecks is that your connection may not give you instantaneous access. You may be hooked up to your ISP with an ISDN line running at 128 Kbps, but you may be waiting for ten to twenty seconds between each burst of data. Therefore, trying to download that 12.5 MB VRML model of the Jerusalem City Hall (yes, such a thing exists at <http://www.lightscape.com/models/chamber.wrl>) even with a good ISDN connection could still take forever.

From Here...

The World Wide Web is a rich source of information that allows you to use an interactive, graphical interface to explore it. To learn about how Web documents are created, or to start programming your own VRML environments, see the following chapters and parts:

- i [Chapter 2, "The Languages of the Web,"](#) outlines the various authoring/programming languages that are used to create documents on the World Wide Web.
- i [Chapter 6, "VR and VRML,"](#) describes the connection between VRML and VR.
- i [Chapter 9, "Overview of VRML Browsers,"](#) describes how VRML browsers work with the World Wide Web and the different types that are available.
- i If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) That section provides a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



2-The Languages of the Web

In the beginning, there was ASCII and the Internet. And for a while, it was good. But soon people needed more than ASCII text to communicate their ideas. So HTML was created, and text and images were being exchanged online. And then people needed to let others have greater interaction with Web documents, and so CGI was created. So it went on and new languages were created to fill the need for the Internet to do more.

Today, the popularity of the World Wide Web is due in large measure to the growing interactivity that is possible on it. Web documents can have forms that e-mail your information directly to the server. You can display animation directly on your Web pages. You can even set up you own 3D environment for others to explore over the Internet. But for this interactivity to work, new languages are needed.

It is important to note that there is more than one type of language being used on the Web. Markup languages like HTML and SGML are the most common. They use tags to describe the different elements of a document's structure and layout. A modeling language like VRML uses various nodes and fields to describe how an object is to be rendered on your display. Programming languages like Perl and C are used to create executable code.

Today, HTML is no longer the only language of the Web. You can program Webforms using Perl, you can use Java to set up an animation on your Webpage, or you can have a 3D virtual environment using VRML.

This chapter provides an overview of the languages of the World Wide Web. While it is not intended to act as a manual for using any of these languages, it does deal with the following issues:

- What are the markup, modeling, and programming languages that can be used to make up documents on the World Wide Web?
- What do they do?
- How do they work?

SGML

Standard Generalized Markup Language (SGML) was developed in the 1980s as a way of describing how an electronic text should be printed or laid out. It is an international standard that uses a set of codes (called markup) to mark different features of a text: for example, a block quotation, italics, etcetera. These codes are embedded in the text. Any word processor or browser that accepts the SGML standard can print or display a text using these markup conventions with its original layout.

SGML allows different machines and programs to use the same set of data to produce a document that looks the same. The embedded codes are not specialized binary tags, like those used in WordPerfect or Microsoft Word, but are instead descriptive tags that are written in ASCII. In this way, an SGML document written on a Sun workstation can be used on a PC or a Macintosh to get the same result.

Another key element of SGML is that it is a "metalanguage." A metalanguage is a language that is used to describe languages in general. This means that you can use SGML to create an infinite number of descriptive markup languages. For example, you could set up an SGML-derived language to describe syntactic features in a text, or one that is designed specifically for poetry. The HyperText Markup Language (HTML) used on the World Wide Web is one example of an SGML-derived markup language.

An important feature of SGML is its descriptive markup system for categorizing the different parts of a text. A descriptive tag provides names to label the different parts of a document. For example, the beginning of a paragraph is always indicated with <p> and the end of a list always uses </list>. This type of markup system enables an SGML document to be processed by different makes of software and hardware.

Another distinguishing feature of SGML is its use of a document type definition (DTD). All documents can be categorized as a report, a letter, a poem, a novel, etcetera and each category can be defined by its constituent parts. For example, a novel is made up of a title, an author, and a sequence of one or more chapters consisting of one or more paragraphs. In contrast, a report might be made up of a title and possibly an author, an abstract, and a sequence of one or more paragraphs.

In an SGML document, the DTD is expressed as a set of declarative statements. In an SGML version of a poem, for example, you would use the following DTD declarations:

<!ELEMENT poem - - (title?, stanza+)>

<!ELEMENT title - o (#PCDATA)>

<!ELEMENT stanza - o (line+)>

<!ELEMENT line o o (#PCDATA)>

Each of the above statements begins with the !ELEMENT tag to indicate that they are all elements. In SGML, an element is a component of a text. The first line of this DTD states that poems may or may not have a title, but all poems have stanzas. The next line indicates that the title is parsed character data (PCDATA). PCDATA is recognized by an SGML parser after the data has been processed. The third line indicates that all stanzas have lines and the fourth line states that a line will be identified in the PCDATA.

NOTE

Parsing is the process of deciding whether a string of characters can be identified according to a predefined set of criteria. A syntactic parser, for example, analyses a string to see if it conforms to a certain language and then determines its syntactic structure. An SGML-compliant parser is able to distinguish the data from markup tags.

The second part of the declaration, the "-" and "o" in the DTD, states whether or not start or end tags are necessary for each element. A hyphen indicates that a tag is necessary; the letter O states that it is optional. Thus, a stanza must be indicated with a start tag "-", but does not need an end tag "o".

Once the DTD declarations have been set, you can begin preparing the body of your SGML text. This process involves inputting the text along with any tags that are needed to indicate document type elements or other features. For example, Shakespeare's Sonnet 130:

My mistress' eyes are nothing like the sun
Coral is far more red than her lips' red
If snow be white, why then her breasts are dun
If hairs be wires, black wires grow on her head:

I have seen roses demasked, red and white,
But no such roses see I in her cheeks,
And in some perfumes is there more delight
Than in the breath that from my mistress reeks.

I love to hear her speak, yet well I know
That music hath a far more pleasing sound.
I grant that I never saw a goddess go;
My mistress when she walks treads on the ground.

And yet by heav'n I think my love as rare
As any she belied with false compare.

would look like the following in SGML:

<!ELEMENT poem - - (title?, stanza+)>

<!ELEMENT title - o (#PCDATA)>

<!ELEMENT stanza - o (line+)>

<!ELEMENT line o o (#PCDATA)>

<poem><title>Sonnet 130

<stanza>

<line><highlighted rendition=italic>My mistress'</highlighted> eyes are nothing like the sun<mdash>

<line>Coral is far more red than her lips' red<mdash>

<line>If snow be white, why then her breasts are dun<mdash>

<line>If hairs be wires, black wires grow on her head:

<stanza>

<line>I have seen roses demasked, red and white,

<line>But no such roses see I in her cheeks,
<line>And in some pérffumes is there more delight
<line>Than in the breath that from my mistress reeks.
<stanza>
<line>I love to hear her speak, yet well I know
<line>That music hath a far more pleasing sound.
<line>I grant that I never saw a goddess go;
<line>My mistress when she walks treads on the ground.
<stanza>
<line>And yet by heav'n I think my love as rare
<line>As any she belied with false compare.
</poem>

You can see how SGML uses tags to indicate different typographic features in the text. The "em" dashes are indicated with the `<mdash>` tag. The e with an acute accent in pÈrfume is indicated by `é`; and the italicized text is surrounded by the `<highlighted>` tag. In fact, an important feature of an SGML text is that it can be "read" by both humans and machines.

Once you have an SGML text, it can be parsed automatically by any machine that has an SGML-compliant parser. The parser will separate the markup tags from the data, recognize when any graphics must be inserted into the text, and display the text in the way that the author intended (see Fig. 2.1). The parser can also tell you if anything is missing from your SGML file before you send it out.

Fig. 2.1

Screenshot of Electronic Book Technologies, Inc.'s DynaText software for SGML browsing and authoring. More information is available at EBT's homepage: <http://www.ebt.com/>. Used with permission of Electronic Book Technologies, Inc. (EBT), Providence, RI, (401) 421-9550.

The development of SGML as a standard for text markup has already begun to affect the way that we are using text. As well as being adopted by word-processing programs as a conversion standard, SGML is also being used by researchers for text analysis. The Text Encoding Initiative (TEI) has adopted SGML as its standard for encoding texts. And even more important for the Internet, SGML set the standard for the main language of the World Wide Web: HyperText Markup Language (HTML).

HTML

In many respects, HyperText Markup Language (HTML) is the foundation language of the World Wide Web. Over the last few years, it has evolved from being a language that combines text and graphics over the Internet into a conduit that lets you integrate applications that bring a greater degree of interactivity to

your Web documents. Whether you are using a Perl-based form or a Java applet, access to these applications is provided through an HTML document.

HTML was first developed by Tim Berners-Lee (TBL) using the DTD declaration of SGML. TBL released the HTML 0.0 specification to the public in 1990. In 1992, Dan Connolly began to develop the HTML 1.0 DTD. The HTML 2.0 specification is the current standard for the Web and the HTML 3.0 specification is already in development.

NOTE

Although the HTML 2.0 specification is the current standard version, all Web browsers are not created equal in the way that they handle HTML. Netscape Navigator is able to use non-standard HTML elements known as the Netscape extensions, thereby creating incompatibilities between it and other Web browsers. The Netscape extensions have proven to be extremely popular, and the makers of other browsers have begun to adopt them. Moreover, the W3 Consortium is under pressure to adopt them for the HTML 3.0 specification. However, there is no guarantee that the Netscape extensions will be adopted for HTML 3.0 or that all browsers will recognize them. The extensions' widening acceptance by the Web community at large makes it likely that at least some of them will be adopted by the W3 Consortium. Other browsers such as Microsoft's Internet Explorer have also developed their own proprietary tags. However, it is yet not clear which ones will be adopted.

Because it is an SGML-derived language, HTML is defined as a descriptive markup language that uses the ASCII character set. This means that HTML can be used on a wide range of computer platforms. In order to maintain this flexibility, graphics, sound, and applications are not stored directly in an HTML file. Instead, they are stored in separate files that are called up by instructions in the HTML file. The Web browser sends a request to the server to download the appropriate file and then determines how to handle the file according to its MIME type (see "Mime Types," Ch. 1).

All HTML documents contain a head and a body. The head elements supply the document title and establish the relationship between the HTML document and the server's file directories. You can define the relationship between your various Webpages and index the material on your Webpage.

All HTML documents begin with the following:

<HTML>

<HEAD>

<TITLE>Your Web Document's Title</TITLE>

</HEAD>

The first tag (<HTML>) alerts the Web browser that the file is an HTML file. The <HEAD> tag defines the starting point of the head element. The third element defines the document's title. The title does not appear in the document itself. It is what names your document and usually appears in the title bar of your browser. The title element of your document is different from its main header (as defined by the <H1> and </H1> tags), which will appear in the viewer portion of your browser along with the rest of your document's body (see Fig. 2.2).

Fig. 2.2

The HTML Title and Header. A screenshot of the Netscape Web browser displaying the document's title in the bar and the document's main header in the viewer section.

The body of an HTML document is where the content of your Web document resides. A Web browser downloading an HTML document knows that anything following the <BODY> tag must be interpreted until the </BODY> tag is reached. Following the </BODY> tag, you need to mark the end of your HTML document with the </HTML> tag. Figure 2.3 shows a simple HTML file that displays the title of this book, the authors, and a short blurb.

NOTE

The HTML editor used for this chapter is HTMLed because it presents the actual HTML code. However, there are a number of HTML editors that process the code to show you how your document will look in a browser. For an overview of HTML editors, check "Mag's Big List of HTML Editors" at <http://union.ncsa.uiuc.edu/HyperNews/get/www/html/editors.html>. However, any ASCII text editor will work just as well.

Fig. 2.3

A simple HTML document.

And when the same file is viewed using a Web browser like Netscape, we get the following:

Fig. 2.4

A simple HTML document as seen with a Web browser.

As you can see, the text is easier to read, the main heading is distinguished by its font size, and emphasis can be made using typographic features like italics.

But improving the reading quality of a text transmitted over the Internet is not the most important feature of HTML. In fact, including the term hypertext in its name is a bit misleading. Hypertext refers to computer soft-ware that lets you read texts that are linked in linear and non-linear ways. But while HTML's main component is text, the key elements and links in these documents have expanded tremendously. Indeed, a more accurate term would have been hypermedia: the presentation of information using media that most effectively communicates its content and that provides the user with the means to access that information in the most appropriate manner for that task.

HTML allows other media forms and applications to be incorporated into its documents through the use of links. All hypermedia applications allow for the creation of links between different elements. In HTML, these links are programmed into the file as URL references. We begin to get an idea of how these links work if we look at how we can incorporate an image into a Web document.

As was mentioned earlier, an HTML document does not embed an image into its file. It instead sets up a link that tells the Web browser where the file containing the image is located and, in the case of GIF and JPEG image files, where to position them in relation to the text. For still pictures and graphics, HTML uses the tag to alert your browser that an image needs to be inserted at a particular point in your

Webpage. The standard image file types recognized by all Web browsers are GIF (using the .gif file extension) and JPEG (using the .jpg file extension). The IMG syntax is

The "SRC=" tag is used to indicate that what follows is the location of the image file, and the image_URL variable is the image file's URL. For example, if I wished to set up a GIF file called vrml.gif as the masthead of my Web document, I would also have to include all the URL information in order to let the browser know where it was located (see Fig. 2.5).

Fig. 2.5

Adding an image to the beginning of an HTML document.

This line informs the browser that it needs to send a request to the server at www.yorku.ca to send the file called vrml.gif located in the directory ~smatsuba. The browser would then display the image at the location specified in the HTML file (see Fig. 2.6).

Fig. 2.6

The image as seen with a Web browser.

As well as allowing you to display images on a Web page, HTML can set up links to video and sound files, to applications, or to other sites and Internet services. Like images, these other elements are not embedded in the HTML file but are referenced using an HTML tag and a URL. We can understand how these links work by setting up a link to another Webpage.

Let's say that we wish to add a link to the Que Publishing home page in our Web document. We first need to find the URL for that site (<http://www.mcp.com/que>). We then need to decide where we want to place that link. In the case of our Web document, we will set the link at the words "Que Publishing" in the second sentence.

To make the link, we use the anchor element (<A>) and the HREF statement. The anchor element signals where a link to another document or source is located. It can also define the location that is to be linked to. The HREF statement defines the destination of the link. To make our link to the Que Publishing home page, we take the following line:

Que Publishing

and insert it where the words "Que Publishing" are located in the text of the blurb (see Fig. 2.7). This line indicates that a link is to be made to the site at <http://www.mcp.com/que>. Moreover, the link is set at the words "Que Publishing."

Fig. 2.7

Adding a link to an HTML file.

Now when a Web browser displays our document, "Que Publishing" is highlighted in the second sentence in the blurb (see Fig. 2.8).

Fig. 2.8

The link as it appears on a Web browser.

When someone points her or his cursor at the highlighted text and clicks the mouse, the browser will read the anchor and send a request to the Macmillan server (www.mcp.com) to send the Que Publishing home page to you. This link exists only in that occurrence of "Que Publishing." If you want other instances of the same words to act as links, you must repeat the same anchor element for each one.

You can also add links to graphic images using the same command. However, rather than surrounding a string of characters with the HTML anchor elements, you replace the string with the image element. For example, we can set up a link to call up a short animation file (vrml_log.mov) using the masthead graphic we have already set up. Again, we use the anchor element and the HREF statement to define the link in our Web document:

```
<A HREF="http://www.yorku.ca/~smatsuba/vrml_log.mov"><IMG  
SRC="http://www.yorku.ca/~smatsuba/vrml.gif" BORDER=0></A>
```

And we insert this line into our HTML file at the same point as the original image (see Fig. 2.9). You'll notice that the IMG element is the same as before except that it now includes the statement BORDER=0. We add this statement to avoid having a border around the masthead being added by the anchor element.

Fig. 2.9

Adding a link to a graphic image.

Now when you view the HTML file in a Web browser and click on the masthead (see Fig. 2.10), the browser will send a request to the server at www.yorku.ca to send the file vrml_log.mov in the directory ~smatsuba.

Fig. 2.10

Clicking on the image in a Web browser.

How your browser handles this file will depend on the MIME type set for .mov files. If your browser's MIME setting is configured for a viewer (in this case, a Quicktime for Windows viewer), the browser will call up the program and display the animation (see Fig. 2.11).

Fig. 2.11

The video image as called up by the anchor.

If your browser's MIME settings are not configured to run a viewer, then a dialog box will appear asking you if you wish to save the file, cancel the download, or configure a viewer (see Fig. 2.12).

Fig. 2.12

Netscape dialog box for unknown MIME types.

You can also configure MIME types and subtypes manually. For configuring VRML browsers, see "How Do I Get My Web Browser to Use My VRML Browser?" (Ch. 9). To learn how to configure your

browser for other helper applications in Netscape, see Special Edition Using Netscape (Que Publishing, 1995).

As well as providing links to other sources, HTML provides ways for you to post e-mail (using the mailto statement), access FTP and Gopher sites, and create forms that can be processed using CGI scripting languages such as Perl. And as new applications (such as Java and VRML) are developed, the ability of HTML to adapt guarantees that it will have a major role to play in the formation of the information superhighway.

Common Gateway Interface (CGI)

To be perfectly honest, the Common Gateway Interface (CGI) is not a Web language. It is a standard protocol along the lines of TCP/IP and HTTP. However, CGI and Web languages are inexorably linked together, and so it is more appropriate to deal with it here.

CGI is a standard for incorporating executable programs into Web documents by way of the server. In other words, it is a way of providing dynamic output as opposed to the static output of a normal HTML document. CGI applications can perform information processing, retrieval, and formatting tasks on a Web server that can be transmitted to you over the Internet. For example, when you do a search on the Yahoo Directory (you can find the Yahoo Directory using the following URL: <http://www.yahoo.com/>), your request is sent to the Yahoo server. The server's Web daemon then takes your request and executes the desired program. The daemon will then take the output of the program and send it back to you. This transaction is an example of the functioning of a gateway. These gateways are usually handled using the <ISINDEX> and <FORM> HTML tags.

Every time you send a URL request for a CGI program, the server where the program is located will execute it in real time. Its output will then be sent directly to you. To accomplish this feat, CGI uses environment variables to send the information to the program. An environment variable is a value passed to a program and/or a programming script by the operating environment on the system on which it is running.

Any input to a CGI application is given by the server to the application on its standard input. The server also provides information about the type of data being sent so that the application can decide how to interpret this data.

A CGI application sends its results to its standard output. It also tells the client how to interpret the information that is being sent. For example, if the application returns an HTML formatted document, it must include the MIME Content-Type "text/html" so that the browser will know how to handle the output.

An important issue concerning using CGI applications over the Internet is security. If you have ever used a Webform, you have probably seen a warning from your browser about sending sensitive information using this method. And the recent cracking of the Netscape 1.1 Web browser's encryption algorithm has made this issue even more critical. If you want to incorporate a CGI application into your Web document or VRML environment, you should learn more about data security. You can find out more about this subject as it relates to CGI at the Common Gateway Interface home page

(<http://hoohoo.ncsa.uiuc.edu/cgi/>).

A CGI program can be written in any computer language that is available on the server's system. There are two categories of CGI languages. The first is scripting, or interpreted CGI languages like Perl, TCL, the UNIX scripting languages (the Bourne shell and the C shell), and Python. Compiled languages like C and C++ comprise the second.

A CGI application (or script) written using one of the scripting languages can be stored as-is in the /cgi-bin directory. Any time a script is requested through a Web browser, it is run through a program that interprets its instructions and performs whatever operations are required. An important advantage of these languages is that a script is easier to debug, modify, and maintain than a compiled application.

A CGI application that uses one of the compiled languages must be processed into machine code before making it available on the Web. This process involves running a compiler for the language you are using. If you are writing a C++ program you must use a C++ compiler; if you are writing a FORTRAN application, you must use a FORTRAN compiler. Moreover, each compiler has a specific version for each computer platform. Thus, a C++ program compiled on a UNIX machine cannot be run on a PC. Therefore, anyone wanting to create an application using a compiled language must use the compiler configured for his or her Web server.

To get a better idea of how these types of languages work, we can look at two specific examples: C++ and Perl.

C++

C++ is an extension of the C programming language and was developed by Bjarne Stroustrup in the early 1980s. It is an object-oriented programming (OOP) language. This type of computer language permits you to reuse classes of computer code more easily, thereby reducing the cost of developing programs. It also permits you to add to the abilities of these classes.

A C++ application can be compiled into a very tight binary object that requires little file space. This allows you to produce very fast running programs—an important feature when having to deal with the transmission of data over the Internet. Moreover, compiled languages run much faster than interpreted ones. Therefore, if your CGI application requires speedy processing, C++ is a good language to use.

However, C++ is not optimized to handle strings. A string consists of character data: "Stephen" and "Bernie," for example, are strings. Unfortunately, 90% of all CGI applications involve heavy string manipulation. This process takes time, and undermines the speed advantage of using a compiled application.

C++ is much better at handling numbers. You can produce a program that analyses and outputs statistical data that is very fast and small in size. Therefore, using C++ to produce CGI applications that need to process a great deal of numerical data makes much more sense.

Although there are a huge number of public domain C++ libraries available, there are a very limited number of CGI-oriented C++ libraries. CGI is a relatively new area, and most developers in this area are using scripting languages. However, the need is there and it is only a matter of time before more C++ libraries for C++ begin to appear. One example is Eugene Kim's "cgihtml," found at

<http://hcs.harvard.edu/~eekim/web/cgihtml/>.

Perl

The Practical Extension and Report Language (Perl) is an interpreted language that is well-suited for scanning text files, extracting information from them, and printing reports based on that information. It is designed to be easy to use and efficient, and combines elements of C and UNIX scripting languages. Moreover, it is fairly easy to learn. First, it is derived from existing languages so that people who have some programming or UNIX experience will already be familiar with some aspect of Perl. Second, you do not have to know everything about Perl in order to use it. You only have to learn a small part of the language in order to use it effectively.

The reason that Perl is so popular on the Web (the vast majority of CGI application on the Web use it) is because it is very good at manipulating strings. It uses a sophisticated pattern-matching system that can quickly and efficiently scan a large amount of text. It can also manipulate binary data with relative ease.

Here is one example of a Perl script called matsuba.pl. It was written for a registration form for an e-mail seminar. The form was made using the HTML form codes and asks people for their name, institution, e-mail address, status (student, faculty, or other), fluency in English, French, and Latin, and any additional comment they might have. The Perl script is designed to send an HTML formatted message to the registrants that their message has been received, and then takes the information from the HTML form and sends it directly to me.

```
#!/usr/local/bin/perl  
#header error  
do "cgi.lib.pl" || die "Fatal Error: Can't load cgi library";  
&ReadParse;  
print "Content-type: text/html\n\n";  
#Returns acknowledgment of registration submission  
print "<HTML>";  
print "<HEAD><TITLE>Your Registration Form Has Been Submitted!\</TITLE>";  
print "</HEAD><BODY>";  
print "<H2>Your Submission Has Been Submitted to the Moderator!\</H2>";  
print "<P>You will be informed by moderator about the\n";  
print "the seminar schedule as soon as possible\n";  
print "<HR>";  
print "<H3>Back to the <A HREF=\"/matsuba.html\">";  
print "More Seminar Registration Page</A></H3>";
```

```
print "</BODY></HTML>";
#assign process id to $pid
$pid=$$;
#open COMMENTSFILE
open (COMMENTSFILE,">/tmp/my_comment.$pid");
#enter data into file for mailing
print COMMENTSFILE "Shakespeare Seminar Registration\n";
print COMMENTSFILE "óóóóóóóóóóóóóóóóóóóó\n";
print COMMENTSFILE "Name: $in{'name'}\n";
print COMMENTSFILE "E-mail Address: $in{'e-mail'}\n";
print COMMENTSFILE "Institution: $in{'institution'}\n";
print COMMENTSFILE;
print COMMENTSFILE "Status: $in{'Status'}\n";
print COMMENTSFILE "Other: $in{'Other'}\n";
print COMMENTSFILE;
print COMMENTSFILE "Languages\n";
print COMMENTSFILE "óóóóó\n";
print COMMENTSFILE "English: $in{'English'}\n";
print COMMENTSFILE "French: $in{'French'}\n";
print COMMENTSFILE "Latin: $in{'Latin'}\n";
print COMMENTSFILE;
print COMMENTSFILE "Additional Comments: $in{'add'}\n";
#close out file
close COMMENTSFILE;
#send file to me
$command="mail engl5105@nexus.yorku.ca < /tmp/my_comment.$pid";
system ($command);
#erase temp file
```


unlink ("/tmp/my_comment.\$pid);

To send the information to me by e-mail, the Perl script instructs the server to create a temporary file called my_comment and to insert the information in separate lines in the file. It then tells the server to e-mail the file to my address and then to erase my_comment.

There are many online resources available to you if you wish to learn more about programming in Perl. A good place to start is the University of Florida's Perl Archive (<http://www.cis.ufl.edu/perl/>). It is a collection of Perl resources that includes Weblinks to other online material and an FTP archive of sample Perl applications.

Java

One of the most recent and most talked about developments are the compiled/interpreted languages. They are written in a script and then compiled into a binary file that can be used on any computer platform (such a file is called an architecture-neutral binary object). This file is then transmitted to a client and is processed through an interpreter on the client's computer.

A compiled/interpreted language that is getting a great deal of attention lately is Java. It was developed by Sun Microsystems, which defines it as "a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language" ("The Java Language: A White Paper," <http://webrunner.neato.org/1.0alpha3/doc/overview/java/index.html>). In other words, it is a simple compiled language that can be used on any computer and can be transmitted over a network—including the Internet. In fact, it resembles a simple, modified version of C++.

Java expands the interactivity of a Webpage by letting you insert small software programs that run on your computer through a Java-compliant Web browser. Sun Microsystems has released its HotJava browser and Netscape Communications Corporation has incorporated Java into its Netscape 2.0 Web browser. HotJava and Netscape 2.0 will let you see Java-based animation and other Java applications running directly on your Webpage. And some of the applications you will be able to use include interactive 3D science experiments, live stock-portfolio management, and online auctions. You are only limited by your imagination.

The advantage of Java over scripting languages is speed. Since it is a compiled language, applications written using Java can run at much faster rates than applications written in languages like Perl. Its advantage over compiled languages like C and C++ lies in its relatively easy learning curve and its ability to be transferred in binary form to different computer platforms.

The Java language is an attractive middle ground between scripting languages like Perl and fast compiled languages like C++. It is a simple object-oriented programming language that is secure enough to be used freely in network-based environments. And it is poised to make its mark as one of the key languages of the Web. For more information about Java and its related products, see the Java Home Page (<http://java.sun.com/>).

.

A recent thread in the VRML Mailing list (www-vrml@wired.com) has looked at using Java applets in VRML environments. Some of the people in the discussion have suggested that Java be used as a possible standard for defining behaviors in VRML 2.0. No resolution has been reached and the thread

will probably continue for some time. For more about the future of VRML with reference to behaviors, see "Behaviors," (Ch. 28).

Adobe Acrobat

In the strictest sense, Adobe Systems Incorporated's Acrobat file format is not a Web language. Acrobat was designed to serve as a "universal creation and viewing tool for electronic documents." The file extension for this format is PDF. However, you cannot write a PDF file yourself. You must take a document file that you have already written and use Adobe software to convert it into an Acrobat file. However, there is a growing interest in making Acrobat a standard data file format for the Web, and so it is relevant for discussion in this chapter.

The interest in Acrobat lies in its ability to retain the layout, font, and illustrations of the original document even if the machine reading the file does not have the same font set. Moreover, the same PDF file can be viewed on any computer platform using Adobe's Acrobat Reader (see Fig. 2.13). Thus, you can see exactly the same document whether you are using a UNIX workstation, an IBM-compatible PC, or a Macintosh computer. All you need to have is the Acrobat Reader for that platform.

Fig. 2.13

A screen shot of the Adobe Acrobat Reader for Windows.

Adobe bases its Acrobat software on the Postscript_ page-description language. Postscript is the same language that is used as a standard in many printers. And unlike HTML, Acrobat stores its graphics directly in its files. Therefore, you cannot simply take a PDF file and edit it with an ASCII editor; you must use Adobe's proprietary software to make any changes to the file.

So what are the advantages to using Acrobat? Well, whereas HTML will allow you to use only different font sizes in your document, Acrobat allows you to combine different fonts and font sizes. You also have more control over how you format your document. In fact, an Acrobat document can look more like a printed book than HTML can because you can convert a document made with a desktop publisher into a PDF file.

However, while Adobe distributes its Acrobat Reader for free, you must purchase Acrobat if you want to write your own PDF files. Moreover, a seven page Acrobat document can take up as much as a megabyte (1 MB) of space as opposed to an HTML document that would take up only a few kilobytes.

At this time, the role that the Acrobat file format will play on the Web is not entirely clear. On one hand, people are downloading the free reader and others are posting Acrobat format files at a number of Web sites. You can find Acrobat "books" on subjects as diverse as herbal remedies and the DIS standard for networking virtual reality applications. On the other hand, Acrobat is a proprietary format that can, at least for now, only be viewed and created using Adobe software. Acrobat's position as a part of the Web, therefore, is still to be determined.

Virtual Reality Modeling Language (VRML)

It seems a bit (well, really more than a bit) redundant to have a short section on the Virtual Reality Modeling Language (VRML) in a book dedicated to the subject. However, I would like to describe how VRML is positioned in relation to the other Web languages discussed in this chapter.

In many respects, VRML has more parallels to HTML than any of the other languages. Like HTML, VRML uses the ASCII format for its files. And like HTML, VRML is a descriptive language that is interpreted by a browser.

But whereas HTML produces a static interface that needs to use other applications to give it a dynamic element, VRML is dynamic by its very nature. VRML does not describe the format of a page; it describes the layout of 3D environments and objects. One way to understand the difference between HTML and VRML is to think about the difference between a blueprint of a building and a model of the same building. You cannot manipulate the contents of a fixed, two-dimensional entity like a page; you can manipulate a 3D model to consider different perspectives.

But VRML adds an additional level of interactivity by allowing you to provide links to other Internet resources. Imagine walking through a museum, seeing an object that interests you, and being able to simply point to it in order to access information about that object or to be shown related objects. VRML gives you this kind of ability by allowing you to render an object and link it to a Web document or to another VRML object or environment. Rather than flipping the pages of a book to find information, you can wander through a three-dimensional representation of that information.

What is most important, however, is that the VRML standard is only at the initial stage of its evolution. We are already beginning to see extensions to the VRML standard. Sony has come up with its own extensions. Worlds Inc. is developing VRML+. Microsoft's Blackbird is using a variant of the VRML spec. And there are already plans to integrate VRML with Java. We will see massive changes in the VRML specification over the next few years with each change bringing a greater degree of interactivity and power. And as computer hardware changes, so too will the VRML standard. It is going to be a wild and exciting journey.

And so, let us begin....

From Here...

The languages of the World Wide Web open up the Internet to a wide variety of applications. To learn more about these languages, or to start programming your own VRML environments, see the following chapters and parts:

- If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) These chapters provide a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.
- [Appendix B, "The VRML 1.0 Specification,"](#) lists the various nodes and fields that are used to create VRML environments on the World Wide Web.

i The "Annotated Bibliography," lists books and online references on various subjects covered in this book including the various languages of the Web.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



3-What Is VR?

VRML is intimately connected with the entire concept known as "virtual reality." Before we can begin our VRML explorations in earnest, we need to come to terms with what this strange beast called VR is really all about.

This chapter

- Defines virtual reality
- Clarifies the differences between VR and multimedia
- Gives some history of the field
- Examines the current "state of the art"
- Gives examples of applications in which VR might be used

A Definition

There is no "official" definition of what virtual reality really is. The result is that the "VR" buzzword has been applied to everything from computer games to 3D movies, and a lot of people aren't sure just what qualifies as "true" VR and what doesn't. With all the marketing hype that abounds in the industry, how do we define VR?

Well, here's a good working definition which serves to separate the hype from the reality:

- Virtual reality is a computer-generated simulation of a three-dimensional environment, in which the user is able to both view and manipulate the contents of that environment.

The key elements of this definition are that virtual reality is computer-generated, three-dimensional, and interactive. A lot of people confuse VR with its dim-witted cousin, multimedia. It's not surprising that people get them mixed up; both technologies came to maturity around the same time, both involve images and sounds, and both have gotten comparable amounts of hype in the popular press. However, the differences between them are important.

VR, Television, and Multimedia

Multimedia is about preassembled, preprogrammed, "canned" information being presented with a novel interface; virtual reality is dynamic and ever-changing. Multimedia is basically two-dimensional, a series of flat images presented in sequence on a screen; VR is (by its very nature) three-dimensional, with depth as well as width and height. In multimedia, the only "interaction" the user has available is the ability to select a different sequence for presenting the recorded information; the user can't alter what's there, and he or she can't add to it. VR, on the other hand, is intensely interactive and malleable. Even the most primitive VR system is much more interactive than the most advanced combination of multimedia hardware and software.

You sometimes hear people say that television has turned us into a society of couch potatoes, sitting on our sofas and watching images flow past us on a flickering screen. Some people wonder whether VR will have the same effect on us, increasing our tendency to be observers rather than participants.

The truth is that VR will have exactly the opposite effect. The very nature of virtual reality is that it's a participatory experience. The user of a VR system has the freedom to explore the world, and interact with it in new and exciting ways. There's no point in using VR, if all you're going to do is stand around and look at it. The whole idea is to be an integral part of the virtual environment, and to have a direct effect on it in the same way that we have on the real world.

History

That's a pretty lofty goal, and the dream of bringing imaginary worlds to life has been with us a long time. However, it's only in this century that we've developed the technology to take the first few tentative steps down that particular road.

Most people assume that virtual reality is a recent development, something born in a computer lab somewhere in the early '90s. The fact is that the basic concepts have been around since the 1950s, and the first primitive systems were created long before man first walked on the moon.

In the late '50s, a quiet man named Morton Heilig began designing the first multisensory virtual experiences. He developed something called the Sensorama. Resembling one of today's arcade machines, the Sensorama combined projected film, audio, vibration, wind, and even prepackaged odors—all designed to make the users feel as if they were actually in the film rather than simply watching it. Since real-time computer graphics were many years away, the entire experience was prerecorded, and played back for the user.

Although he was a gifted and visionary inventor, Heilig was less successful as a businessman. He was unable to get funding for his Sensorama machines, and they were never manufactured. Fortunately, he didn't give up there; Heilig had an idea that would later prove to be the basis for an entire industry: the first Head-Mounted Display, which he patented in 1962. The Head-Mounted Display, often just referred to as an "HMD," has come to be associated with immersive virtual reality systems.

Heilig was one of the real pioneers in the field; for more information about him, check out Howard Rheingold's book "Virtual Reality" (Simon & Schuster, 1991).

Several years later, a computer graphics professor named Ivan Sutherland invented the term "The Ultimate Display," to describe a display system for computer graphics that tracked the user's head and updated the display to match. The result would be that the user felt surrounded by, or immersed in, a world that existed only inside the computer.

Despite the limitations of 1960's-era computer technology, Sutherland nevertheless managed to build the first computer-based VR system. Called the "Sword of Damocles," it consisted of a computer monitor, a mechanical tracking system, and a computer to drive the whole thing. The computer produced black-and-white line drawings of simple shapes, which the user could view from different angles. Although it was quite primitive, it did prove that the concept worked. It also laid the foundation for much of the work that was to follow.

In the 1970s the NASA Ames Research Center began exploring the idea of virtual reality. It built a simple HMD out of parts from Radio Shack, and hooked it up via a magnetic tracking system to a host computer. Again, it was primitive but it worked. Users could even use a "data glove" to manipulate objects in the virtual environment. It truly was "a computer-generated simulation of a three-dimensional environment, in which the user was free to both view and manipulate the contents of that environment." In other words, the first true VR system.

Media coverage of the work at NASA Ames, including the fact that the parts were from Radio Shack, led many hobbyists to begin building their own "homebrew" VR systems. It wasn't long before they were putting together parts from pocket TVs, welders' helmets and stereo headphones, and writing their own software from scratch.

And so it began, just like in the early days of the microcomputer industry. Hundreds, possibly thousands of VR enthusiasts working away in garages and small, underfunded university research labs, gradually developing the technology that would spawn a new industry.

There were some early pioneers. Joe Gradecki's magazine, PCVR, dealt with VR technology at the solder-and-assembly-language level. Robert Sudding sold kits for primitive but functional "graphics accelerator boards," complete with NTSC (i.e., television signal) output for HMDs. I created (with Dave Stampe) a program called REND386, which quickly became the most popular VR software package for use by hobbyists. Hardware hackers adapted such devices as the Nintendo Powerglove and the Sega Shutter Glasses for use as VR peripherals, and newsgroups and mailing lists sprang up on the Internet.

The Present

Those days are long gone. Although some die-hard hobbyists (such as the talented and innovative Robin Hollands) are still determined to do it all by themselves, more and more companies have entered the field of VR with dollars and determination. The basic technology of VR is now available in the form of off-the-shelf consumer products. Companies like Virtual i/o, VictorMaxx Technologies, and Forte are making HMDs that are better than anything hobbyists could build for themselves, at a comparable price. There's no point in hacking the Nintendo glove, since ready-made gloves are now available from companies like 5DT and AGE. The time that hobbyists once spent scrounging for parts and handcrafting assembly language can now be spent on the more creative aspects of VR, and that's where VRML enters the picture.

Just as the World Wide Web has made the Internet accessible to the general public, the arrival of consumer-level hardware and software will bring VR to the masses. And just as HTML became the "lingua franca" of the Web, VRML is becoming the standard for sharing virtual worlds.

Realistic Expectations

With all the hype surrounding VR, it's important to avoid setting your expectation levels too high. From watching films and television programs that have a VR theme, some people have come to expect that virtual reality is better than life itself. When they put on their first HMD, they're usually disappointed, both by the lack of realism and the limited content.

The first of those problems is being addressed by improved technology. It's important to remember that VR is in its infancy; it will be several years before it lives up to its full potential. True, VR is only just barely possible on the fastest desktop computers; producing something that simulates reality is a difficult technological challenge. Compare early silent films to the wide-screen, surround-sound spectacles of today; that's the kind of transformation that virtual reality will undergo as faster processors, 3D graphics accelerators, and higher-resolution displays become available.

The Holodeck is science fiction. The reality of today's technology is that we can produce reasonably good graphics, excellent sound, and some limited tactile feedback. In order to bring any realism to the virtual environment, world-builders and world-participants must meet each other halfway. They must use a technique that predates modern technology by thousands of years: the willing suspension of disbelief. For centuries, actors would create amazing "virtual worlds" on stage, using nothing more than a few simple props and the imaginations of their audience. It is that same willing suspension of disbelief that must be used to bridge the gap between the visions of the future and the realities of today's technology.

Don't get me wrong; technology will progress. If history is any indication, the progress will be very rapid indeed. In the meantime, it's essential that determined world-builders concentrate their efforts on creating good content, which gets participants involved in new and exciting ways.

Applications

Despite all its limitations, virtual reality technology is being used in all kinds of different areas. There have been some amazing success stories, even in the brief history of the field.

Architects are taking their customers on virtual walkthroughs of buildings that don't exist in the real world, making changes as they go. Educators are creating classroom tutorials that exist only within the virtual worlds of their desktop computers, and using them to teach biology, physics, math, chemistry, and history. Doctors are preplanning complex surgery, and medical students are being trained on "virtual cadavers." Engineers are collaborating in new and exciting ways which were undreamed of even a few short years ago.

Perhaps the best example of a real-world application for VR is the "Motorola project" undertaken by Nina Adams. A cellular-phone assembly line was re-created in virtual reality, and people who trained on the virtual version actually performed better than those who trained on the real thing.

Artists are finding a whole new medium in the form of VR. Anything they can visualize, they can create

and share with others. Not just static sculptures or paintings, but dynamic experiences that combine the best elements of other art forms including film and theatre.

Whole worlds are being created for the sole purpose of giving a context to social interaction. People are meeting in cyberspace, and with VR they can interact in ways they never could in a purely textual medium.

From Here__

Now that we understand VR in general terms, the next chapter will let us dive in and take a look at the technology itself. If you don't care much about technology, you may want to skip ahead to Chapter 6 where we take a look at how VRML fits into VR.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



4 - How Does VR Work?

At the moment, VRML browsers are basically just tools for examining three-dimensional objects on your computer screen. However, they're about to evolve into something much more powerful; with the addition of programmability, behavior and networking, they will turn your computer into a full-blown VR system.

In order to understand VRML, you need to understand how a VR system works. This chapter will give you all the technical background you need.

After reading this chapter, you'll

- Know what a VR system does
- Recognize the major components of a VR system
- Understand the different types of input and output devices
- Be able to evaluate new hardware and software as it becomes available

To start with, let's find out just what a VR system does.

Behind the Scenes

When you use a VR system, it seems very straightforward. You're in a virtual world, wandering around and interacting with it—what could be simpler? Well, in order to make it seem simple, the VR system has to do a tremendous amount of work. It's like the old saying: "A VR system is like a duck. On the surface, it's calm and unruffled—but underneath, it's paddling like crazy!"

Every VR system has to perform at least the following basic functions:

- Keep track of every object in the virtual world
- Store and update information about each object's location and appearance
- Simulate the behavior of the objects
- Render (i.e., draw) the world in three dimensions
- Generate sounds for the virtual objects

- Allow the user to navigate through the virtual environment
- Provide the user with some means of interacting with objects in the environment

In order to do some of these things effectively, special hardware is required. In particular, navigating and interacting with the environment is best done using special input devices.

Goggles and Gloves

When most people think of virtual reality, the image that comes to mind is someone wearing a head-mounted display of some sort, and possibly instrumented gloves. The media has latched onto the "goggles and gloves" approach because it's visually interesting; it's much better to show a picture of someone "suited up" for VR, rather than just sitting and looking at a computer screen.

As we'll see later, there are other ways to do VR that don't involve wearing any special gear; nevertheless, most people think of Head-Mounted Displays and gloves as essential components. In any case, it's clear that a true immersive experience does require some special-purpose hardware, so let's take a look at what's involved.

An Overview

In Chapter 3, we looked at some of the differences between virtual reality and multimedia. Well, there's another important difference that we didn't mention: the hardware requirements are completely different. Sure, there's a computer involved in both casesóbut that's where the similarity ends.

Figure 4.1 shows a typical computer system.

[Fig. 4.1](#)

A block diagram of an ordinary computer system.

In addition to the computer, there are input devices (keyboard, mouse, and joystick) and output devices (monitor and speakers).

A VR system is very similar, as shown in Figure 4.2.

[Fig. 4.2](#)

A block diagram of a typical VR system.

The similarities are obvious, but it's the differences that matter. Instead of keyboards, mice and joysticks, you find strange things called "forceballs," "gloves," and "head trackers." In place of the familiar monitor and speakers, you find an HMD.

This is the important point: what makes a computer into a VR system is the peripherals you add to it, and the software you run on it. Obviously, you need to start with a fast processor (and a 3D graphics accelerator and wavetable-based sound card wouldn't hurt, either!), but it's the input and output devices that really count.

Degrees of Freedom

There are a number of input devices for virtual reality, but the one thing they all have in common is that they provide more than two "degrees of freedom." In order to understand what is meant by "degrees of freedom," we need to look at the way you move around in the real world.

From a standing start, you can move forwards or backwards, left or right, and up or down. In other words, you can move along any of three axes, the forward/back axis, the left/right axis, and the up/down axis. Those represent three degrees of freedom, the ones we call the translational degrees of freedom. In VR, we talk about moving around as "translating" from one location to another. That may seem odd, until you realize that television cameramen refer to that same action as "dollying!"

However, the ability to move around in three-dimensional space is only half the story. The other half has to do with rotation.

In addition to moving forward/back, left/right, and up/down, you can turn to face in any direction while staying in the same spot. You can look up and down, left and right, and even tilt your head over onto one shoulder or the other. These three actions, called pitch, yaw, and roll (or sometimes tilt, pan, and roll), are the three rotational degrees of freedom.

Three translational and three rotational degrees of freedom give you a total of six degrees of freedom altogether. That's why input devices for VR are often referred to as "6 degree of freedom," or simply "6 DOF."

Any device you use for navigating in VR should be capable of controlling your movement in 6 DOF, allowing you to move in any direction and to look wherever you want. In addition, any device you use for interacting with objects should allow you to move and rotate them using 6 DOF.

Mice, Joysticks, and Keyboards

A standard computer mouse has only two degrees of freedom: you can move it forwards/backwards on its mouse pad, or from side to side. You can't lift it, push down on it, or twist or turn it in any way. A joystick is the same; you can tilt it forward/back or left/right, but that's all. The arrows on a computer keyboard are also designed with 2 DOF in mind. This makes sense, since all these devices were designed to be used with a two-dimensional screen, not a three-dimensional space.

However, it's possible to use these 2 DOF devices as if they were 6 DOF, by using the buttons on the device to "map" the two available degrees of freedom into six. For example, moving the mouse forward with the left button down might make you travel forward; the same action with the right button selected might make you move up. The same thing can be done with a joystick, or using the shift keys on a keyboard.

Another approach is to use on-screen control panels. These are used in some VR systems (such as Virtual Reality Studio) and some VRML browsers (such as WorldView). The user can click on a set of arrows on screen that let them control all six degrees of freedom.

Although these approaches work, they do have some problems. The use of buttons to shift axes is

cumbersome and unnatural, and on-screen controls take up valuable screen space. What's worse, there are no standards; if you switch from one VRML browser to another, you'll have to learn a whole new way of navigating.

Devices like mice and joysticks are even more difficult to use if you're wearing a Head-Mounted Display, since they're all "attached" to your desk. Some other devices are a sort of compromise; for example, there are "wands" or "pucks" which are handheld devices that use a combination of buttons, "hat" controllers and possibly tilt sensors. You still have to do the axis-mapping trick, but at least you're free to stand up and move around while using them.

All things considered, though, it's probably better to use a better input device—one with 6 degrees of freedom.

6 DOF Controllers

The first widely-used 6 DOF controller was the original Spaceball, shown in Figure 4.3.

The tennis-ball-sized sphere of the Spaceball could be pushed or pulled in any direction, as well as being twisted every which way. The result was a very natural, 6 DOF interface. If you wanted to move forward in the virtual environment, you pushed the ball forward; if you wanted to look to your left, you twisted the ball to the left.

[Fig. 4.3](#)

The original Spaceball.

Spaceball has since come out with a lower-cost version, called the Spaceball Avenger, which is a handheld unit for computer game and VR applications.

Another 6 DOF controller is the Logitech Cyberman, which has a mouse-like handgrip that's free to slide in between two DOF, and can be lifted up, pushed down, or twisted.

One big advantage that these 6 DOF controllers have is that they accelerate the "learning curve" that most VR users have to travel in order to start really using the system.

These sorts of devices are great for providing additional degrees of freedom to the user for navigating through the virtual world. However, they can be awkward to use for manipulating objects. What's needed is a way of tracking the user's hand position. While we're at it, we should track the user's head position if they're using an HMD. That brings us to the complicated subject of "tracking."

Tracking

Tracking systems measure at least three, and usually six, degrees of freedom. They report the information back to the computer, which updates its idea of where the user's head and hand are located in the virtual environment.

There are five basic tracking technologies: mechanical, ultrasonic, magnetic, optical, and sourceless.

Mechanical Tracking

Mechanical tracking uses linkages or pulleys to measure the location and orientation of a point. For example, the mechanical arm arrangement used by the Immersion Probe is shown in Figure 4.4.

Fig. 4.4

The Immersion Probe, a mechanical tracker.

The technology behind mechanical tracking is really very simple. There are sensors at each joint, measuring its angles. Since the computer knows the length of each segment of the arm, and can measure the joint angles, it can compute the location and orientation (6 DOF) of the tip of the stylus. The joint sensors are usually either potentiometers (variable resistors) or optical shaft encoders (disks with alternating black and white areas that pass under a photocell).

Mechanical tracking has a number of advantages. It's potentially very cheap, very reliable, and has minimal lag (which, as we'll see later, is very important). The main disadvantage is that you're limited by the maximum length of the arms. You can also only track one object at a time (head or hand) since multiple arms would be awkward and would get in each others' way.

Ultrasonic Tracking

In an ultrasonic tracking system, there's a set of transmitters and a set of receivers. Each transmitter is "pulsed," causing it to send out a high-frequency sound. That sound is detected at each receiver, and the length of time it took the sound to travel can be used to find the distance from the transmitter to the receiver. By using three receivers, and measuring all three delays, the computer can pinpoint the location of the transmitter relative to the receivers. Alternatively, you can have three transmitters and one receiver. The overall concept is illustrated in Figure 4.5.

Fig. 4.5

An ultrasonic tracking system.

For example, in the Kantek Ringmouse system, the transmitter is a cordless ring worn on the user's finger. When the ring receives an infrared signal from the computer, it sends out an ultrasonic pulse, which is picked up by the three receivers. The delays correspond to the distances, and the location can be accurately measured.

That works very well for reporting X, Y, Z location (in other words, 3 DOF). In order to get rotation information, you have to add more transmitters and pulse them in sequence. This technique was used in the Nintendo Powerglove and in the Logitech ultrasonic tracker.

In addition to the delay between the transmitted and received pulses, more accurate measurements can be made using the "phase difference" between the two. If the signal being sent out is a sine wave, and the one being received is also a (perhaps slightly distorted) sine wave, there'll be a difference in the phase of the two signals; in other words, if you draw them both on the same graph, the received signal will be to the right of the transmitted one by an amount that's proportional to the distance between the transmitter and the receiver.

Ultrasonic systems have strengths and weaknesses. Among their strengths is the fact that they can be

quite accurate, they are potentially quite cheap (which is why they were used in the Powerglove), and they can be made cordless (like the Ringmouse).

Unfortunately, they usually have a fair amount of lag. It takes time for the sound to travel from the transmitter to the receiver, and you have to wait for the signal to arrive at the last receiver before sending another pulse. If you're measuring orientation, you need to pulse multiple transmitters. Worse yet, you have to wait for echoes to die down before pulsing again. All of that adds up to a significant lag.

Ultrasonic systems require a line-of-sight path between the transmitters and the receivers; if anything "occludes" (i.e., hides) the transmitter from the receiver, the signal won't get through.

Ultrasonic systems are also subject to room noise. For example, every ultrasonic system I've ever used will simply stop working if someone jingles their keys anywhere in the same room. The keys produce harmonics at the same frequency used by the transmitters, which confuses the system. They all recover immediately once the jingling stops, but the ultrasonic systems are impossible to use in arcades or on trade show floors. For home use, where the acoustic environment can be controlled, they work much better.

Magnetic Tracking

Magnetic tracking is similar to ultrasonic tracking, in that there's a transmitter and a set of receivers. More specifically, there are three transmitting coils grouped together into a small cube and three receiving coils in an even smaller cube.

Each transmitting coil is pulsed, and the intensity of the magnetic field is measured by each receiving coil. With three transmit coils and three receive coils, there's enough information to compute a full set of location and orientation values (6 DOF).

There are two magnetic tracking techniques. The first, which was patented by a company called Polhemus, uses alternating current (AC) magnetic fields. The second, patented by a company called Ascension, uses pulsed direct current (DC). Since there really are only AC and DC, the two companies have, between them, patented the entire technology of magnetic tracking. In fact, Ascension was started by former Polhemus employees, and there was a patent battle between the two companies over whether AC tracking and DC tracking were sufficiently different; Ascension won.

Magnetic tracking isn't subject to occlusion; in other words, it doesn't matter if there's some (non-metallic) object between the transmitter and the receiver. Magnetic tracking also doesn't have as much lag as ultrasonic, but it does take time for the fields to build up and collapse. You also need to pulse all three coils, which adds to the lag.

Another important limitation of magnetic tracking is the use of metal in the environment. AC tracking (the kind used by Polhemus) induces small electrical currents in any kind of metal. Those currents produce their own magnetic fields, which distort the ones used by the tracker. That distortion limits the accuracy of the overall system.

Pulsed DC systems (the ones used by Ascension) don't have that problem, but are still affected by ferrous metals. Ferrous metals (like steel, for example) contain iron, and can therefore have their own magnetic field which distorts the tracker's.

Despite these limitations, most VR systems have used magnetic tracking quite successfully. Magnetic tracking has been quite expensive, but the costs should decrease with volume production for large markets; for example, Ascension has announced an "under-\$1000" tracker called the Spacepad.

Optical Tracking

Since ultrasonic sound and magnetic fields have been used for tracking, why not light? As it turns out, there are a variety of approaches to optical tracking.

For example, you might attach light sources or reflective color markers to the object being tracked and then watch them with a video camera. Alternatively, you might attach the video camera to the object, and put the light sources in the surrounding room; the advantage is that small movements of the object will result in large (apparent) movements of the light sources in the camera's field of view, giving much greater accuracy.

Another approach is to use infrared transmitters and receivers instead of markers and cameras. This is the approach taken by the Virtuality/Atari HMD, for example.

Optical systems have minimal lag, and can be made very accurate. Unfortunately, camera-based systems are very expensive due to the cost of the camera itself and the image-capture or signal-processing boards. Optical systems are also subject to "occlusion"—that is, objects can block the light from the camera or sensor. For example, if you have a colored marker on the back of your hand, and then turn your hand over, the system stops working. You could add another sensor below, but what if your hand was turned sideways? You'd need a third, and a fourth, and so on.

For some applications, though, this isn't a problem. For example, if you're tracking the head movement of someone sitting in a chair, occlusion is never going to happen.

Sourceless Tracking

All the systems discussed so far have involved a "source" of some kind, whether it's an ultrasonic or magnetic transmitter, a light source or reflector, or even the base of a mechanical arm.

A new kind of technology is called "sourceless" tracking, and it combines a two-axis inclinometer with a sensor that measures the magnetic field in the surrounding room. There's no transmitter; the Earth's gravity and magnetic field are used as the reference.

The three-axis magnetometer measures the direction of the ambient magnetic field, the two-axis inclinometer measures pitch and roll, and the pitch/roll values are then used to compute the yaw angle from the magnetic field information. The result is a measurement of orientation (3 DOF).

Although no location information is reported, sourceless tracking can be used for HMDs in a sit-down configuration; the user's head doesn't move, but it does rotate. The pitch/roll sensors only measure about +/- 45 degrees, but that's about all the user's head can move anyway.

A Quick Summary

Table 4.1 summarizes the strengths and weaknesses of the various tracking technologies:

Table 4.1 Tracking Technologies

	Mechanical	Ultrasonic	Magnetic	Optical	Sourceless
Accuracy	Very good	Moderate	Good	Good	Good
Cost	Low	Low	Moderate	High	Low
Lag	Very low	Very high	Moderate	Very low	Low
Problems	Awkward	Ambient sounds	Metal in environment	Occlusion	Rotation only

As the table shows, there's no single tracking system that's suitable for all applications. You need to pick and choose, depending on your needs for accuracy, cost, and low lag as well as the characteristics of your environment.

Gloves

For a long time, the only gloves available for VR use were made by a company called VPL. They patented a number of ideas related to the use of gloves as an input device, then lost the patents to a creditor. The patents have since been purchased by Greenleaf Medical Systems, and several companies are now producing low-cost gloves. Among them are 5DT and AGE, which have both released gloves in the "consumer" price range.

All of them work in basically the same way. The user wears a glove that has flex sensors built into each finger; when they bend their fingers, the sensors report that information to the computer, which updates the virtual representation of the hand. The computer might also recognize various gestures made with the glove, and interpret them. For example, a pointing gesture might be interpreted as "move in that direction," and a grabbing gesture might mean "pick up this object."

There are two basic flex sensing technologies. The simplest (used in the original Powerglove and the new AGE PC Powerglove) consists of strips of plastic with conductive metal that changes resistance when the fingers bend. The other technique (used in the VPL Dataglove) is a length of optical fibre with some of the cladding stripped off. As the fibre is bent, some of the light leaks out and the photosensor at the far end receives less light.

The most recent glove to hit the market is the so-called "Pinch" glove, which uses electrical contacts in the fingertips. Although it doesn't measure finger flexion, it does allow certain specific gestures to be made using different combinations of fingers in contact with each other.

In addition to finger flexion sensing, gloves are often equipped with one of the tracking systems described earlier, usually magnetic or ultrasonic.

Regardless of the technology being used, gloves represent a very natural and intuitive interface for immersive VR systems. Someday, we can expect full-body data suits to measure every joint angle in our body, allowing us to bring ourselves into VR as more than just disembodied heads and hands.

Displays

Of course, there's more to VR hardware than just the input devices. The computer needs to be able to display the virtual world for the user to see, and to do that it needs an output device. There are three basic approaches: "Window on the World," projection-based systems, and HMDs.

Window on the World

The simplest display system uses your computer's screen as a window that looks into the virtual world. All of the first-generation VRML browsers work this way, simply because it reaches the broadest base of potential users.

The advantage of a Window on the World display is that it uses hardware that the user already has. The disadvantage is that there is no real sense of "immersion," no feeling of being transported into the virtual world.

You can do things to increase the feeling of immersion; for example, you can present the images stereoscopically. There are two basic techniques: shutter glasses and optical separation. Both techniques work by presenting slightly different images to each eye, so that the brain fuses the two images and gets a sense of depth.

In a shutter-glasses based system (such as the 3D Max), the user wears special glasses that have a liquid-crystal "shutter" over each eye. The computer can independently control each shutter, effectively covering or uncovering one eye at a time.

The computer presents the left eye image for one frame, with the right lens being "shut" and the left lens being "open." Then it displays the right eye image, with the left lens being "shut" and the right lens "open." This repeats, giving each eye its own view.

Shutter glasses work well, but low-cost ones are often subject to flicker; since each eye is only seeing half the frames, the action doesn't seem as smooth, and cheap shutter glasses don't switch from clear to opaque quickly enough.

In an optical separation system, the computer draws the left-eye image in one part of the screen and the right-eye image in another. A set of lenses and/or mirrors are used to make sure each eye only sees its intended image.

Window on the World displays are often made more effective by being placed in the context of a "pod" or "cab." The user sits inside an enclosed cubicle (sometimes mounted on a motion platform) viewing the monitor. Although the view of the virtual world outside the pod is the same as in a standard Window on the World system, the fact that the user is surrounded by the pod enhances the sense of being in a different world.

Projection Systems

A simple projection-based system just puts the view of the virtual world on a wide screen. The width of the screen serves to increase the sense of immersion, just as it does for motion pictures or IMAX films.

A "CAVE" system involves the use of multiple projectors, and screens that surround the user on three or four sides. There's a projector for each screen, projecting onto the back of the screen (so that the user is surrounded by the world), as shown in Figure 4.6.

Fig. 4.6 A CAVE.

Cave systems are very useful for small groups, since everyone can look at the world simultaneously. However, they do have some disadvantages: they require multiple projection systems, huge amounts of computing power to generate all those images simultaneously, and lots of floor space for setting the system up. These limitations make CAVEs impractical for home use, but they're well-suited to a museum or Location-Based Entertainment facility.

Note that CAVEs can be made stereoscopic, by using the shutter glasses described above.

Head-Mounted Displays

Finally, we come to the one piece of hardware that people associate most strongly with VR: the Head Mounted Display, or HMD.

An HMD is a combination of four things: one or two display screens, some optics, a set of stereo audio speakers, and a tracking system. We've already talked about tracking. Stereo speakers are straightforward, and optics is a complex and involved subject that you (as a user or world-builder) should never need to worry about. So, let's concentrate on the displays themselves.

There are two basic types of displays found in HMDs: CRT and LCD.

CRT Displays

A CRT is a picture tube, just like the ones found in your TV set or your computer monitor. CRT stands for Cathode Ray Tube; a beam of electrons is emitted from an electrode (the "cathode") and deflected onto a screen using electric and magnetic fields.

CRTs can have very high resolution, which is useful for HMDs. However, it's difficult to manufacture a CRT small enough for use in an HMD, and the cost of producing them is very high. What's more, they tend to be heavy (because of the glass, magnetic coils, and so on) and require high voltages and strong magnetic fields in close proximity to the user's head.

Despite these limitations, many of the "high-end" commercially available HMDs (such as the n-Vision) use CRTs. In some cases, a white-phosphor display is used, and color is produced using either a color wheel or a device called a "PI-cell filter." In either case, the approach is the same: the red component of the image is shown first (in black and white), with a red filter in front of it, followed by the green and finally the blue components. The brain fuses the three to get full RGB color.

LCD Displays

LCDs are most often used in "pocket" TV sets or camcorder viewfinders. They're flat panels, composed of an array of "pixels" (spots of color) that are controlled by the electronics.

There are several advantages to LCDs over CRTs. They're less expensive, they weigh a lot less

(important for HMDs!) and they don't require high voltages or magnetic fields. Unfortunately, they tend to be fairly low-resolution, so the images seem "grainy." Diffusion filters can be placed over the displays to soften the edges of the pixels, and higher-resolution LCD panels will soon be available.

Stereoscopic Viewing

In order to view images stereoscopically, an HMD needs two displays (one for each eye). Since the displays are usually the most expensive single item in the entire HMD, the cost of the product is strongly influenced by whether or not it is stereoscopic.

There's also the problem of interfacing; most computer systems have just one video card in them, so driving two display screens is difficult. One approach is field-sequential encoding (the same technique described earlier for supporting shutter glasses), but a more popular technique is to use alternate-scanline encoding. The idea is to put the left-eye image on the even scanlines of the display, and the right-eye image on the odd scanlines (or vice-versa). The HMD electronics will separate the two images and send them to the appropriate displays.

Of course, there's no such thing as a free lunch. In a field-sequential system you're only getting half the frame rate to each eye (just as with shutter glasses), while in the alternate-scanline system, you're only getting half the number of scanlines per eye.

Resolution

Speaking of scanlines, one of the most important factors to consider if you're shopping for an HMD to use with your VR system is the resolution of the display.

Resolution refers to the number of dots that make up the image. Common resolutions for desktop computers are 320 by 200, 640 by 480, and 1024 by 768. These refer to the number of columns and rows of pixels (color dots) on the screen. For example, a 320 by 200 display (the most common resolution for 3D games) has 200 rows of pixels (i.e., scanlines) with 320 pixels in each row.

HMDs using LCD displays measure things a little differently, in that they group the individual red, green, and blue color spots into "triads." We would normally think of each triad as one pixel, since each pixel would have red, green, and blue components. However, the LCDs are actually measured not in rows and columns but rather by the number of individual color spots. For example, many first-generation consumer HMDs have 180,000 "pixels." Since each of these so-called "pixels" is really only 1/3 of a true pixel (i.e., each one is either red, green, or blue) there are really 60,000 pixels on the display. This is roughly equivalent to a 300 by 200 display, a grainier image than even the lowest-resolution computer game.

Field of View

Another important factor to consider in an HMD is the field of view. The field of view is the angle through which you can see the virtual world; if it's too narrow, it feels like you're wearing blinders. In the real world, we have a field of view of about 180 degrees horizontally and 150 degrees vertically. Some HMDs have a field of view as low as 30 degrees, and very few go over 50. Some HMD manufacturers attempt to "cheat," by quoting the diagonal field of view instead of separate horizontal and vertical measurements. Assuming an aspect ratio of 4:3 (horizontal to vertical), which is the standard for

television, then the horizontal field of view is 80% of the diagonal, and the vertical field of view is 60% of the diagonal.

One reason that many HMDs have such a narrow field of view has to do with image sharpness. Let's say you have a 320 pixel wide display, spread over a 32 degree field of view; that's 10 pixels per degree. Now, increase the field of view to 64 degrees; that's 5 pixels per degree—much, much worse. In other words, the more you spread the image out over a wider field of view, the grainier it looks. A narrow field of view keeps the image nice and sharp, but reduces your sense of immersion. This is a basic trade-off that will always be with us, but as display resolutions increase we'll be able to have a wider field of view with the same image sharpness.

One common measure of "sharpness" is the angle over which each pixel is spread out. Degrees can be subdivided into "minutes of arc"; there are 60 arc-minutes per degree. The human eye can only perceive down to about 1 arc-minute, so a near-perfect display would have 1 arc-minute per pixel. However, a real-world display would have many more arc-minutes per pixel, which is bad. To find out how much worse it would be, let's do some math: 30 degrees times 60 arc-minutes per degree means that the field of view is 1800 arc-minutes. Divide that by 320 pixels, and you get 5.625 arc-minutes per pixel—very bad. Some displays will be even worse, if they have a wider field of view.

Ergonomics

Perhaps the most important factor in evaluating an HMD is its ergonomics. How much does it weigh? Less is better, if you're going to be wearing it for any length of time. How well-distributed is the weight? A lot of HMDs are front-heavy, because of the displays and the optics; this can produce neck strain, especially if you turn your head quickly. How well-padded is it? Does all the weight rest in one spot?

The best advice if you're shopping for an HMD is "try before you buy."

Sound

In addition to the visual display, most VR systems support some type of audio output. The effect of good sound is overwhelming; studies have shown that people using identical VR systems, one with sound and one without, actually report seeing better graphics on the one with sound!

Even an ordinary sound card is adequate for most VR applications, but a card with wave-table synthesis will take some of the burden off the main processor (leaving it more time to do the graphics). True 3D sound placement requires an expensive board like the Beachtron, which does real-time convolution of the sound (using something called a Head-Related Transfer Function, or HRTF) to place it in three-dimensional space. Most VR systems don't require that level of sophistication, but it all depends on what your applications are.

Version 1.1 of VRML is expected to support sound, including the specification of a sound's location in three dimensions.

Haptic Feedback

There's no question that our two most important senses are sight and hearing. However, we also rely on touch for a variety of different tasks; haptic feedback is all about stimulating our sense of touch.

There are two important parts to our haptic sense: tactile stimulation and force feedback. Tactile refers to our sense of how a surface feels: smooth or rough, wet or dry, hot or cold. Force feedback refers to how hard our muscles have to work to accomplish some task, such as lifting or pushing a heavy object.

Both tactile stimulation and force feedback have been studied extensively, and there has been a lot of work done in an effort to create low-cost stimulators. Unfortunately, most of the hardware is very bulky and expensive, and not yet practical for home use. Some exceptions are the Aura System's Interactor vest, the Logitech Cyberman (which has a built-in motor to provide vibration) and the AGE Powerglove. A company called Exos (which has provided force-feedback products for the VR industry for many years) has a force-feedback joystick coming out called the PowerStick.

There is no special support in VRML for tactile feedback, though there has been some discussion about extending its material descriptions in the future to provide surface texture information. Force feedback could be supported as well, by adding information to the VRML file describing the weight, center of gravity, and other physical properties of an object.

Motion Platforms

Motion platforms use hydraulic, pneumatic, or electromechanical actuators to tilt the user from side to side and forward/backward. They're being used increasingly in arcade and Location-Based Entertainment facilities, but are unlikely to be found in home-based systems any time soon. VRML does not make any special provisions for motion platforms, nor does it need to; as the user navigates, the platform simply matches their orientation. Collision detection systems could be used to provide the dips and bumps of a landscape as it's traversed. The proposed ElevationGrid node in VRML could be used for that purpose.

Interfaces

One important consideration in assembling a VR system for home or arcade use is the question of interfacing. All the input devices (and some output devices, such as motion platforms) need to be interfaced to the computer in some way. Standard computer systems have two serial ports (plus a joystick port). Once you have a 6 DOF controller, a glove and a head tracker attached, you've run out of ports.

Some systems get around this by having their own interface card, but that uses up a slot on the motherboard. This could be a problem; after all, the system will need a video card, a sound card, and possibly a multi-function card with serial ports, disk controllers, and so on. Slots fill up almost as quickly as serial ports do. There are other standards on the horizon, such as the "ACCESS.bus," but it's too soon to tell if they'll find widespread acceptance. In any case, the vast majority of devices use a serial interface.

And hardware interfacing is only half the battle; you also have to get your VRML browser to recognize the devices. Hope may be on the way on that front; Microsoft has announced their Virtual Input API, a

standard way of interfacing to input devices that have multiple degrees of freedom. With any luck, this standard will allow users to hook up almost any kind of peripheral to their VR system and have it work the way it should.

Health and Safety Issues

Perhaps the most controversial subject when it comes to VR hardware is the issue of health and safety. There are a number of concerns to be addressed, and it's important to remember that the research isn't complete by any means.

Safety

The first issue is safety. The scenario a lot of people are worried about is that children will put on an HMD and go running around the house; they might fall out a window, or tumble down a flight of stairs, or trip and land head-first in the bathtub. Amusing though this may be, there is a serious danger of lawsuits against the companies that make the hardware involved. Some manufacturers (e.g. Virtuality/Atari) are even going so far as to make their system impossible to use unless you're sitting in front of your computer (of course, since that's a limitation of their tracking system, it may be a case of turning a bug into a feature).

Hygiene

A second area of concern relates to hygiene. Obviously, this is less of a concern in a home-based VR system. However, in an arcade setting, people may be sharing HMDs and gloves—a choice opportunity for various kinds of lice, bacteria, and other nasty little critters. Interchangeable headbands, gloves with a changeable inner liner, and occasional spraying with disinfectant are all wise precautions.

Eyestrain

Another source of problems is the optical system in an HMD. A badly designed optical system can produce severe eyestrain, especially after prolonged use. Most if not all of the modern low-cost HMDs have addressed this issue, and have spent a great deal of time and money on their optics. It's also important that the systems be adjusted properly to the user; for example, the IPD (Inter-Pupillary Distance) varies from one person to another, and many HMDs provide some means of adjusting it. Focus adjustments are also important, unless the HMD can be worn over eyeglasses.

Lag

One of the biggest areas of concern is lag. We touched on it earlier in this chapter, but the basic idea is that you want to minimize the lag as much as possible.

Where does lag come from? Well, let's consider a typical VR system. Let's say it uses ultrasonic tracking, with three transmitters and three receivers. The tracker pulses a transmitter, and it takes time for the pulse to reach all three receivers. The tracker then has to wait until the echoes die down before pulsing the next transmitter, then wait again before pulsing the third. All this takes time. It also takes time for the

electronics in the tracker to convert those time readings into location and orientation information, and still more time to send it over a serial connection to the computer system. It takes time for the computer to process that information, re-compute the user's head location and orientation, and re-render the scene. It also takes up to 1/30th of a second more before the rendered image actually appears on the screen. Add all those numbers up, and you could wind up with so much lag that the system becomes unusable.

Imagine how it would feel if you turned your head, and the whole world took a second to catch up with you. It wouldn't be very long at all before you started getting nauseous. If the total lag exceeds 100 milliseconds, it becomes difficult to use the system. Above 200 milliseconds, you enter what researchers have nicknamed "the barfogenic zone."

Faster systems, both in tracking and in rendering, will help reduce the lag considerably. Of considerable importance will be hardware accelerators and (eventually) the use of parallel processing.

CAUTION

Regardless of how well-designed the optics are, how good the ergonomics are, how fast the graphics run and how much fun it is to use, you generally shouldn't use an HMD for long periods of time. Let common sense prevail, and take frequent breaks to give yourself a chance to get re-adjusted to the real world.

From Here...

Well, we've talked about the hardware involved in virtual reality; later in the book, we'll talk about the software issues, including three-dimensional rendering (the toughest part of any VR system). From here you should move on to [Chapter 6](#) to start learning about VRML itself, or to [Chapter 8](#) to find out what kind of computer hardware you need to use VRML.





5-Networks and Distributed VR

In 1984, William Gibson's *Neuromancer* presented "a consensual hallucination" called Cyberspace. But in that artificial world, the only real person is the main character. Everyone else is a computer-generated being. Eight years later, Neal Stephenson's *Snow Crash* opened the Black Sunóa bar in a networked virtual environment. In the Black Sun, people from around the world construct virtual images of themselves to meet and mingle with other people.

In many respects, networked VR has a similar division. Some versions of VR on networks were designed to share computer resources rather than connect people to each other in a virtual world. Others were designed with communication in mind. VRML was conceived as a networked VR application using the Internet as its pipeline.

This chapter outlines developments in networked VR systems. The topics it discusses include:

- The principles involved in populating virtual environments
- Networked VR and the games industry
- The DIS standard
- A description of various networked VR applications

Populating Virtual Environments

In *City of Bits*, William J. Mitchell notes that the role of computers "Is to construct cyberspaceóa new kind of place for human interactions and transactions" (*City of Bits: Space, Place, and the Infobahn*. Cambridge, MA: MIT Press, 1995, p. 109). And many VR developers have worked to find ways in which this "new kind of place" can be rendered as a 3D environment. Indeed, one of the attractions of networked VR is its potential as an engine for collaboration and interaction.

But the technical elements needed to bring people together in virtual environments can be daunting. Networked VRóespecially on a global scaleórequires a great deal of computing and networking power. It is not enough to connect people in an artificial environment; you must enable actions and interactions to be transmitted and rendered in real time.

The Technical Side of Networked VR

A major problem for networked VR environments concerns how to keep track of the positions and actions of the different users in the environment, and transmit that information to the different client computers. Currently, there are two models for connecting people in virtual environments. The first, the centralized model, connects users to a central computer that analyzes and distributes information to each client (see Fig. 5.1). The central computer receives the data from the clients' computers and stores it. It then sends the information out to each client, and the client's machine interprets the information and renders the scene.

Figure 5.1

The centralized model for networked VR.

The centralized model allows programmers to develop simple data structures to store and handle the incoming and outgoing data. All the information can be channeled and processed by the central computer before going out to the clients, thereby improving the rendering times. However, this model places limits on the number of users that can be hooked up to the same environment at any given time. As more users are put onto the system, more burden is placed on the central computer's processor and network interface. There will come a point at which the performance of the central computer becomes overtaxed and information is no longer processed and transmitted quickly enough to have real-time performance.

The second model does away with a central computer and has each user's computer carry its own copy of the complete database. In this distributed model, users might be directly connected to each others' computers (see Fig. 5.2) or to a broadcast-capable network. The VR program on a user's computer calculates and stores any changes in its own copy of the database, and then transmits these changes to the other computers. This avoids the kinds of user limits imposed by the centralized model.

Fig. 5.2

The distributed model for networked VR.

However, the distributed model raises other communications problems. For example, let us say that there are one hundred users hooked up to the same VR environment. Theoretically, each computer would have to be connected to 99 other computers. Moreover, each time it made a change, it would have to send the same message 99 times. Individual computers would eventually be unable to cope with the data traffic. Therefore, the main question lies in how to reduce both the number of connections that need to be made and the number of messages that need to be sent.

One approach is to use the broadcast method of communication. When a user's computer makes a change, it sends only one message out to the network. The network transmits the same message to all the other users at the same time. This significantly reduces the number of messages being sent between the users' computers. It also reduces the number of connections needed.

And yet, while the broadcast method can greatly reduce the number of connections and transmissions between computers, it can still cause transmission bottlenecks with large numbers of users. We can reduce the number of transmissions even further if we use the dead-reckoning algorithm. This algorithm reduces the number of messages that need to be sent by recording users' direction of movement and velocity, and estimating their positions. A message is sent only if a user changes his or her direction

and/or speed.

The advantage of the dead reckoning system is that the various computers in the system do not need to receive an update from the network with each change in an object's position. It is the basis of a number of networked VR systems including the Distributed Interactive Simulation (DIS) protocol and its predecessor SIMNET. For more information about DIS and dead reckoning, see Chapter 29.

Looking at Cyberspace

Another technical aspect of networked VR concerns how we will look at cyberspace. There is a faction that believes that VR is VR only when you have a fully immersive 3D environment. This generally means using a head-mounted display (HMD). But the health issues associated with HMDs like visually induced motion sickness (VIMS), also called "virtual nausea," have raised questions about their long-term viability. New developments like the Virtual Retinal Display (VRD) may help to alleviate these problems, but a commercial version is still a number of years away.

The Virtual Retinal Display (VRD) is being developed by the Human Interface Technology Laboratory (HITL) at the University of Washington. Unlike an HMD which displays an image on a screen in front of the user's eye, the VRD projects the image into the user's eyes using low-powered lasers and optics. The resulting image can have a higher resolution than even the most expensive conventional displays. It also has a larger field of view than most HMDs. For more information, see the Human Interface Technology Laboratory's Web page at <http://www.hitl.washington.edu/projects/vrd/>.

Another display technology that is gaining favor with high-end networked VR systems like the Cave Automatic Virtual Environment (CAVE) uses projection televisions linked with LCD shutter glasses. These displays project images onto large screens or walls to provide a sense of immersion. The 3D effect is produced by alternating images in synchronization with the shutter glasses. This alternates the views in the right and left eyes and produces a 3D effect.

However, a much simpler, and in my view perfectly acceptable, form of display is the lowly CRT display. This option can only produce 2D images of 3D environments. However, it is a readily accessible medium; if you have a computer, you have a computer monitor. Moreover, the popularity of games like Doom and the success of 2D networked worlds like Habitat, Worlds Chat, and WorldsAway shows that this option is a viable alternative to a fully immersive display.

The Distributed Interactive Simulation (DIS)

The Distributed Interactive Simulation (DIS) system is a set of protocols created by the U.S. Department of Defense for networking VR simulations. It is currently the basis for the National Simulation Capability program developed by the Federal Aviation Administration and is being examined by the entertainment industry as a standard for simulation applications in theme parks and distributed games.

The DIS mission statement follows:

"The primary mission of DIS is to define an infrastructure for linking simulations of various types at multiple locations to create realistic, complex, virtual 'worlds' for the simulation of highly interactive activities. This infrastructure brings together systems built for separate purposes, technologies from

different areas, products from various vendors, and platforms from various services, and permits them to operate with each other. DIS exercises are intended to support a mixture of virtual entities (human-in-the-loop simulators), live entities (operational platforms and test and evaluation systems), and constructive entities (wargames and other automated simulations).

"The DIS infrastructure provides interface standards, communications architectures, management structures, fidelity indices, technical forums, and other elements necessary to transform heterogeneous simulations into unified seamless synthetic environments. These synthetic environments support design and prototyping, education and training, test and evaluation, emergency preparedness and contingency response, and readiness and warfighting (DIS Steering Committee, The DIS Vision: A Map to the Future of Distributed Simulation, Version 1, May 1994. Available in a variety of file formats including Acrobat and Postscript at <http://www.sc.ist.ucf.edu/~STDS/ftp/docs/vision.>)."

In DIS, a virtual environment is defined as a set of entities that interact with each other through events that they cause. The core of DIS is a set of protocols that transmit information about entities and events over a network to various simulators. The aim of these protocols is to connect simulators thousands of miles apart and still maintain real-time simulations. This can be achieved by ensuring that any lag between simulators is kept below 100 milliseconds.

For more information about DIS, see "What Comes Next?" (Ch. 29). You can also visit the DIS Service Center Webpage at <http://www.tiig.ist.ucf.edu/~DISSC/>.

The Philosophical Side of Networked VR

When all the technical problems of networked VR are finally worked out, other questions remain. What do we want to do in a virtual environment populated by other people? What form should cyberspace take?

Of course, the fundamental premise underlying cyberspace is that it will become a multi-participant, collaborative forum. While it may be nice to wander a 3D space that is simply interesting to look at, one can get much the same effect by looking through a picture book of the same space. If there is a purpose for using the power of computers to create these virtual spaces, it is to be an active participant in that environment—a participant with the space and a participant with the other people in that space.

Even more fundamental questions concerning cyberspace are "who should be in charge?" and "who should be let in?" The easy answer to the first question is "everyone"; the control of cyberspace should not be left to any one person, company, or agency. However, the issue of who is going to pay for its construction and just how much power will accrue to those builders cannot be ignored. If "the information superhighway" is going to be built, it will need the participation of private enterprise to complete it. Should those who build it not have a say in its control?

In some respects, they already will have a big say. As the global cyberspace becomes a reality (as opposed to a virtual reality), the private interests that help to build it will be able to control huge tracts of bandwidth. They will be able to bombard our homes and workplaces with their information sources, their products, and their messages.

On the other hand, we have earned the right to have our say in the new global cyberspace. It should not be forgotten that the research funding that helped develop much of the technology and the government

dollars that helped build its infrastructure came from us. We also have earned a say because we helped to make the Internetóthat first public, global cyberspaceóthe phenomenon that is currently generating so much interest. Cyberspace may be the product of commercial interests, but it is equally the product of our participation in it. Moreover, the time and effort volunteered by those who created the software for the Internet demand that they have a voice in this debate. Therefore, we have as much right to stake our own claim to a place in cyberspace as does private enterprise.

The answer to the second question, "who should be let in," is the easiest to answer but is often the most ignored. In essence, everyone has a right to have access to cyberspace. Admittedly, today's profile of the average Internet user is affluent and male. But that is the average, and the "unaverage" are beginning to rise in number. More and more women are joining. People in developing countries are starting to get access. Everyone is starting to get access.

More fundamental to this question, however, is the issue of population. What good is a world that has no people in it? If we attempt to restrict the voices in cyberspace to the affluent, it will become a sterile utopia. And all utopias are doomed to fade. If it is to survive, the virtual world must be as diverse and have as many voices as the real world. Indeed, what makes the real world exciting, what makes it fun to live in, is its diversity. The more choices we have and the more people we can encounter, the better the chance we will find things that will interest us. For cyberspace to survive, therefore, we need to have as large a diversity of people in the virtual world as in the real one.

Examples of Networked VR

Now that we have looked at the technical and philosophical aspect of networked VR, we need to look at examples of how network VR has actually been implemented. Some of the examples cited here, like Habitat and WorldsAway, do not necessarily fall into the category of VR. However, they provide insights into this subject and so are included.

SIMNET

The military has had a long history in developing simulators for the training of its personnel. And so it should come as no surprise that it should also be a major player in the development of networked VR applications. In the 1980s, the U.S. Army's need to lower the costs of training its tank crews led it and DARPA to develop the SIMNET system.

SIMNET was the first large-scale, networked VR application. It could accommodate up to 1,000 simulators with fully dynamic images. The computer hardware and software was developed by Bolt, Beranek, and Newman of Cambridge, Massachusetts.

As SIMNET grew, issues about standards and hardware compatibility began to arise. The original technology was developed in 1983 using a closed system by Perceptronics. A decade later, the equipment was VAX-based. The need for networking and simulation standards prompted the formation of what eventually became DIS (see above).

While a great deal of money went into the development and implementation of SIMNET, the focus of the project was to network a large number of simulators for a fraction of the original estimated cost for such a project. The original military estimates placed the cost of each simulator at \$20 million each. The

actual cost for a SIMNET simulator came in at \$200 thousand each.

There were a number of limitations to the SIMNET system. It could not render basic terrain features such as hills, wet ground, or mine fields. The graphics were cartoon-like and lacked stereoscopic viewing. External communications such as voice and radar were also not possible. Moreover, the computer simulations of the enemy lacked any artificial intelligence interface and so were essentially limited forms of target practice.

Nevertheless, SIMNET was a major step in the development of networked VR. It showed that it was possible to connect large numbers of simulators in a network and still maintain real-time interaction. It also was instrumental in pointing out the need for an established set of standards for networked VR systems.

Networked VR and Games

When I was a young lad, my favorite arcade game was a submarine game in which you looked through a periscope to target cardboard silhouettes of ships passing in front of you. It was, in a sense, a crude form of VR. You had a simulator that rendered a virtual world with which you interacted.

Among the first commercial applications of networked VR have been the games industry. Borrowing from the simulator technology developed by the military and industry, the games industry has created a number of networked VR applications for the public. Nearly all of these applications overcome the problems of network communications by setting up in large arcades or amusement parks, where high-speed LANS are easy to install. However, they do provide the public with a fully immersive VR experience.

Virtuality

Virtuality's system is a fully immersive VR environment for the arcade market. It was released in 1991 and has grown in popularity ever since. There were originally two models of the Virtuality system: the Virtuality SU1000, in which the user stood, and the Virtuality SD1000, in which the user sat in a cockpit.

Both the SU1000 and the SD1000 could function as a single-player game or they could be networked for multi-player versions. Both types used an HMD called the Visette and a simple input device. The Visette used two color LCDs, each with a resolution of 372 by 276 pixels, to provide stereoscopic vision. Its field of view was adjustable, going from 90 degrees to 120 degrees. Head tracking and position were monitored by Ascension Technology's Bird tracker: a magnetic sensor similar to the Polhemus magnetic positioning system.

The input devices for the SD1000 included either a steering wheel and an accelerator and brake pedal for car racing simulations, or two joysticks for fighter aircraft games. The SU1000 used a hand-held pointing device with a trigger and button. As you moved this device, a virtual hand appeared in the display holding a virtual weapon. Pulling on the trigger fired your weapon. Pushing on the button moved you in the direction you were looking.

To enhance Virtuality's marketability, the designers made it very easy to use. The wires to the HMD and the input device were gathered into a backpack that was strapped onto the player. The Visette used a large, simple adjustment bar at the back of the helmet to fit it to the user's head size. In the stand-up

version, you stood in the middle of a platform surrounded by a safety ring. The ring kept you from walking off the platform and served as a handhold in the event you became disoriented.

As noted above, the resolution of the Visette was only 372 by 276 pixels. This is a much lower resolution than a VGA screen of 640 by 480 pixels. As a result, the images you saw in the Visette lacked detail. Moreover, because the tracking system used a magnetic sensor, it was difficult for the computer running the system to compensate for sudden movements.

Yet, despite these shortcomings, the Virtuality system has grown in popularity. Virtuality can be found in more and more arcades around North America and Europe. Much of this can be attributed to the networked games that are available on the system.

One of the most popular games for the system is Dactyl Nightmare. When you put on the Visette and start the game, you see an area with a main floor and four platforms connected by stairs. Various geometric shapes are scattered around the area to act both as hiding places and as decoration. The object of the game is to hunt down the other players and shoot them before they shoot you.

The competitive aspect of this game is important to its popularity. Players of the game win by successfully shooting their opponents. However, the game does add a cooperative element. As the game progresses, a computer-generated pterodactyl flies above the players and randomly attempts to grab one of them. If it is successful, it flies over the game area with its catch, and drops it to the ground. The unlucky player then loses points. Players must stop their fighting in order to shoot down the pterodactyl before it grabs one of them. Since they cannot know which player it will pick, it is in their interest to join forces to destroy it before it catches its prey.

Another popular game for the Virtuality system is Legend Quest. It is similar to the Dungeons and Dragons role-playing game. Four players are wired to four modified Virtuality SD1000s that are networked together. The players enter the same virtual world which includes a castle, a dungeon with multiple levels, and a forest. They can speak with each other, and a special voice input/output system lets them use the voice of the character they have chosen to play.

Unlike Dactyl Nightmare, Legend Quest is a game based on player cooperation. The premise of the game places the four players on a joint quest. They have more success when they cooperate than if they try to compete against each other. There are eighteen possible combinations of characters using race (Elf, Dwarf, or Human), sex (Male or Female), and occupation (Warrior, Thief, or Wizard). Each attribute has certain advantages and disadvantages, and so the team with the most diversity is more likely to win the game.

The goal of the game is to overcome ten challenges without getting killed. The Legend Quest game program generates monsters that use an artificial intelligence algorithm. This algorithm lets the monsters "learn" from their experience with the players and perform countermoves. Players do not have to complete the game in one sitting. Each player joins the Legend Quest game center and receives a "smart key" with a memory chip that gives the players access to the system. A database stores all the player information including their character construction and previous location. Players can, therefore, start playing where they left off.

An important lesson to be gained from the Virtuality experience is that you do not need high-quality graphics to have a successful VR system. Ease of use and content play a much larger role than graphics in determining the viability of these systems. Moreover, a user's interaction with other people—either in

competition or as a cooperative unitócan enhance his or her experience in a virtual environment.

BATTLETECH

The BATTLETECH arcade centers are the product of Virtual World Entertainment, Inc. It was founded by Jordan Weisman and Ross Babcock, who developed the concept first as a fantasy role-playing board game in 1985 and later licensed it for three computer game versions. In 1990, they opened a prototype BATTLETECH center in Chicago with the assistance of Incredible Technologies, a Chicago arcade games and amusements maker.

In BATTLETECH, players "control" huge attack-robots called Battlemechs from a cockpit. Players are given a mission briefing before entering the game area in which novices are shown a training video while veteran players plan strategy and go over the mission environment. A game lasts ten minutes. They are then divided into teams of four and sent to the game room. If a player's Battlemech is destroyed before the game ends, it is immediately replaced with a new one.

The designers of BATTLETECH have gone to great lengths to give the illusion of reality to their game. In addition to the mission briefing, the cockpits are complex and designed to create a sense of overload. There are over 100 controlsóall of which have some function. Players can learn how to use more controls as they gain experience. The basic movements require only four controls. The displays in the BATTLETECH cockpits provide only 2D viewing. However, this is more than compensated for by the high resolution of the image.

The game play of BATTLETECH is highly engaging. In order to control your Battlemech, you must use foot pedals, body movement and firing controls, monitor numerous displays, and handle radio communications with other team members. You need to watch your enemy's position, keep track of team members (you don't want to blast them by mistake), and watch over your Battlemech's damage.

The success of the BATTLETECH centers is a function of the game's learning curve. While it is relatively easy to start, getting the full experience requires time. Master players have at least 50 games to their credit. Many average over 200 games each. The success of challenging the user with the interface raises interesting questions about the need for simplicity to interact with a virtual environment.

Disney's Aladdin VR Ride

The latest entry in the field of VR entertainment is Disney's Aladdin. A joint development by Walt Disney Imagineering (WDI) and Silicon Graphics, Inc., the ride is scheduled to open in 1996. Their goal is to provide a VR experience with movie quality graphics. The first application is a virtual world based on the Disney movie, Aladdin.

The Aladdin VR system is the most complex and expensive entertainment system designed so far. It uses high resolution HMDs and motion platforms controlled by SGI Onyx graphics supercomputers incorporating SGI's RealityEngine2 graphics subsystem. This system provides fully anti-aliased, textured images with little delay in the translation of a change in a user's viewpoint to a corresponding change in the image displayed in the HMD. Moreover, it is networked to allow users to see other users flying through the virtual world. And when the final version of the Aladdin system is released, users will be able to interact with different characters from the film.

Disney has attempted to address a number of concerns relating to having a fully immersive, public VR system. For example, hygiene problems with HMDs are solved by issuing individual helmet-liners that are put on before fitting the HMD. During the prototype testing stage, Disney has monitored user response by issuing surveys to those who have tried the system. To date, over 20,000 surveys have been processed.

Disney's Aladdin VR is an example of where the quality of the graphics is the paramount concern for a VR entertainment system. The cost of achieving this goal is extremely high: the hardware alone runs in the millions of dollars. However, Disney has not ignored the interactive aspects of VR. Whether this approach to networked VR becomes a success has yet to be determined.

Communication, Collaboration, and VR

A popular slogan for telephone companies a while back was "reach out and touch someone." Today, networked VR is providing us with the means to do just that. It is becoming a new tool that extends the possibilities for communication and collaboration. The following is a small sample of the projects, both past and present, that use a VR model to connect people with each other.

Habitat

Habitat was a project started by Chip Morningstar and Randall Farmer for Lucasfilm. It is a multiparticipant on-line virtual environment that originally ran on a commercial network in the U.S. until Fujitsu purchased it. Fujitsu adapted the program to use a Japanese interface and has run it since 1990 on its commercial network, NiftyServe.

In Habitat, users use their home computer as the interface for communicating over a commercial packet-switching data network to a centralized server. The home computer provides a real-time animated graphic display of what is happening in the virtual world, and sends a user's input back to the server. The server maintains the world model and keeps the users' computers up to date as to all the changes occurring in that world.

The first version of Habitat operated using Commodore 64s. PC versions were developed later.

The Habitat interface is divided into three parts. The largest section is the graphic display. It contains various objects as well as representations of the users who are present in that part of Habitat's world. The users are represented by cartoon-like figures called avatars. Above the graphics screen is the text screen in which the various conversations being conducted by the users present in the virtual environment are shown. The third part is a control panel for controlling a user's avatar in the virtual world.

A user controls the actions of her or his avatar. Avatars can move around, pick up and manipulate objects, and talk and/or gesture to other avatars. Talking is achieved by typing on the keyboard. The text appears in the text area of the screen. Different avatars' "voices" are distinguished by the color of the text.

The Habitat world is made up of discrete locations called regions. Each region occupies a single screen and can be joined to four other regions. To move to the next region, a user simply moves his or her avatar to the left or right edge of the screen. Other connections can be made using doorways and passages. Each region contains objects that an avatar can pick up and/or manipulate.

Although Habitat is a two-dimensional world, its interface uses many of the elements associated with a networked VR environment. The metaphors used to communicate with other users are largely graphic. Users see each other in the virtual world and are then able to "talk" directly with each other. The virtual world creates a simple representation of the real world in order to make the interaction seem more intuitive. And finally, users are represented with figures—sometimes human and sometimes not—that offer a sense of presence for communicating in the world.

A number of articles have been written about Habitat, most notably Chip Morningstar and F. Randall Farmer's article, "The Lessons of Lucasfilm's Habitat" (in Michael Benedikt, ed., *Cyberspace: First Steps*, Cambridge, MA: MIT Press, 1991, pp. 273-301. This article is also available on the Web at <http://www.communities.com/paper/lessons.html>) and Farmer's article, "Social Dimensions of Habitat's Citizenry" (in Carl Eugene Loeffler and Tim Anderson, eds., *The Virtual Reality Casebook*. New York: Van Nostrand Reinhold, 1994, pp. 87-95. This article is also available on the Web at <http://www.communities.com/paper/citizenry.html>). Both Morningstar and Farmer list a number of principles they derived from their experiences with Habitat. They noted that communications bandwidth is a limited resource and that data traffic needs to be kept to a minimum. They also found that an object-oriented data representation was crucial to maintain efficient communications. Another technical principle was the need for data communications standards.

However, Morningstar's and Farmer's insights into the building and running of an artificial world are also important. For one, they note that it is impossible to have a detailed central design for a large virtual environment. The original design for Habitat called for a capacity of 20,000 users, each with his or her own house situated in a virtual town or city. The designers discovered that the sheer volume of individual environments made it impossible for them to plan out all the places users could visit.

Morningstar and Farmer also discovered that users were quick to exploit holes in the system. Habitat was designed with a commercial system for its avatars. Users discovered that they could buy a certain item for less than the virtual pawnshop would offer to buy it. Users began to line up to buy the item, shuttle across to the pawnshop to sell it, and then return to buy another unit of the same item. Users would spend hours doing this process. However, they eventually amassed fortunes that allowed them to buy more exotic items in the virtual world.

The developers of Habitat also discovered that allowing users to drive the world's design proved far more effective than trying to anticipate the latter's needs. The developers became facilitators as well as designers and implementors. Their influence on the world came in the form of suggestions to the users.

One of the key conclusions that Morningstar and Farmer draw from their experience with Habitat is that the defining characteristic of cyberspace is the interactions rather than the display. A virtual world can exist without HMDs and high-resolution graphics as long as users are able to interact with the world and with each other. It is a conclusion that must be remembered in networked VR design.

WorldsAway

WorldsAway is a joint project of CompuServe and Fujitsu Cultural Technologies. It is an animated, on-line community where subscribers to CompuServe can meet and communicate with each other in real-time. Users create and control their own avatars and can participate in planned events as well as simply wander through the virtual environment and communicate with other users.

The software for WorldsAway is a redesigned and reengineered version of the Habitat software. Thus, the environment is two-dimensional. However, users can control the facial expressions of their avatars to add visual cues to the text-based communication.

The object-oriented technology of this system permits the creation of multiple worlds and environments. This will allow the WorldsAway environment to change and expand as the need arises.

Worlds Chat and AlphaWorld

Worlds Inc.'s Worlds Chat and AlphaWorld software are opening networked VR communication to the Internet. They both provide 3D artificial worlds for users to enter and explore. More importantly, however, they do so with the intention of creating a new avenue for communications between people.

Worlds Chat is defined as "a 3D social computing environment" where people can "chat" with others while exploring a virtual space. It uses proprietary software that is distributed for free by Worlds Inc. (see Worlds Inc.'s home page at <http://www.worlds.net/> and go to the Worlds Chat page). The company also sells the server software needed to run Worlds Chat. You can use a pre-made avatar or you can create your own using BMP-formatted pictures or graphics that you make using the software stored in COMPIMGC.ZIP available at the Worlds Inc. Website.

In order to connect to the Worlds Inc. server, you will need to have a TCP/IP connection (see "SLIP and PPP," Ch. 1). The Worlds Chat software will use that connection to link with the server.

When you run Worlds Chat, the first screen that you will see is the Avatar Gallery (see Fig. 5.3). It is a virtual environment that has pictures of the various pre-made avatars that are available. You can wander through the gallery to find the avatar that most appeals to you. If you see one that you like, click once on its image and a rotating 3D model will appear on your screen. You will be asked whether you want to use this avatar or continue looking.

Fig. 5.3

The Worlds Chat Avatar Gallery.

Once you have chosen the avatar you wish to use, you will be asked for a username (see Fig. 5.4).

Fig. 5.4

Choosing your username on the Worlds Chat program.

Once you have connected to the Worlds Chat server, you will appear in the Worlds Chat Hub (see Fig. 5.5). From here, you can explore the virtual environment. You can go through different doors and wander through corridors to reach other areas, or you can "teleport" directly to different areas.

Fig. 5.5

The Worlds Chat Hub area.

More importantly, however, you can converse with other users on the network. If there are other users in the same area, you will be able to see their avatars. You communicate using text typed on your keyboard. You can monitor and contribute to the general conversation being held in the area. However, you can

also have private one-on-one conversations with specific users using the Whisper feature.

AlphaWorld is another networked VR application that allows users to interact with each other. However, whereas Worlds Chat uses a pre-built environment, AlphaWorld allows users to construct buildings and personal spaces themselves. The inhabitants of AlphaWorld must apply for citizenship and a "passport" number that gives them access to the system. They are then invited to stake out their territory and build their own buildings and/or populate their space with objects of their choice.

You can build your "home" on any open piece of "land" in AlphaWorld. The borders of AlphaWorld are marked by the mountains that surround it. It is interesting to note that the virtual space itself is completely flat (see Fig. 5.6)

Fig. 5.6

An open space in AlphaWorld.

The builder becomes the "owner" of any land occupied by his or her objects, and the space above and below them. You can build a house that is as simple or elaborate as you choose (see Fig. 5.7). You will need to write down the coordinates of any spot where you build in order to find your building or buildings later.

Once you have set up your home in AlphaWorld, other users can begin to send messages to you. You should mark your space with a mailbox (mailbox1.rwx) and put a mailto: designation in the destination slot. If someone has a problem with something you have built, they can send you mail or they can just drop a "hi neighbor" e-mail to you.

Fig. 5.7

A building in AlphaWorld.

For more information about AlphaWorld or to download the AlphaWorld software, check the AlphaWorld home page at <http://www.worlds.net/alphaworld/>.

Virtual Polis

The Virtual Polis is a three-dimensional, distributed, computer-generated city that can be accessed by a potentially unlimited number of users. It is a distributed VR application; that is, it uses a distribution network to connect end users. It is a project of the STUDIO for Creative Inquiry at Carnegie Mellon University and was developed to examine the potential for tele-existence in a virtual environment. The prototype for Virtual Polis was first shown at VR Vienna-'93, a conference on virtual reality held in Vienna in December 1993. As well as the participants who were at the conference, the virtual city also was connected to participants in Pittsburgh and Tokyo.

The Virtual Polis is constructed to simulate a contemporary, western city, and it maintains many of the features of a real city. You can navigate through public spaces like parks and museums. You can also develop your own private space like an apartment or a personal office. You can access interfaces for work, entertainment, and home shopping. There are also virtual tools to alter your environment. You can move your avatar's head and limbs using various input devices such as an HMD and dataglove. Later versions will be able to display different facial expressions.

The designers of Virtual Polis world have attempted to render the virtual city in a way that closely simulates the real world. For example, buildings in the city display a variety of states. You can see new, shining buildings. But you can also find ones that appear dilapidated and shoddy. In some buildings, you can see broken plaster and falling plaster. The environment can also exhibit different behaviors such as moving tree branches.

Virtual Polis uses a client-server model to connect its users. They can dial into the server using a modem, over direct-dial lines and through the Internet. Plans call for a network of multiple servers and more advanced distribution models. The system is multiplatform and the performance of the software will depend on the power of the user's computer. Moreover, users do not have to log on using the same kind of computer.

As much as it is a test of the capabilities of networked VR technology, the Virtual Polis is also a social experiment. It has already pointed to a number of important issues in implementing a global cyberspace: the need for private spaces as well as public ones, the importance of body and facial expressions for communication, as well as voice and text, and concerns about security.

Moreover, Carl Eugene Loeffler notes that the makers of cyberspace need to be careful not to create a new kind of western colonialism. The danger is that the makers of the virtual universe will design it to the exclusion of cultures and ideas that do not conform to our ideas and sphere of knowledge. People's tendency is to be rather myopic when it comes to things that do not conform to our own values (that is, North American, European, or Japanese values) and still believe that we are producing a "global" entity.

CAVE

The CAVE Automatic Virtual Environment (CAVE) is a high-end networked VR system for collaborative scientific research. The illusion of immersion is created by projecting 3D graphics into a 10'x10'x9' cube onto screens that completely surround the user. The 3D effect is achieved using LCD shutterglasses that "flicker" in conjunction with the projected images. This approach to stereoscopic viewing is described in more detail in Chapter 4.

The hardware for the CAVE system is extremely expensive. Each CAVE node uses high-end equipment and connections. The viewing area itself uses several Electrohome Marquis 8000 projectors with 2000 linear pixel resolution to put the images on screens surrounding the user. The audio is created using multiple speakers controlled by the computer. Stereographics' LCD stereo shutterglasses are used to separate the alternating images to each eye. Users' head and hand movements are tracked using Ascension tethered electromagnetic sensors.

However, the heart of each node of the CAVE system is its computer and communications link. The computer is a Silicon Graphics Onyx graphics supercomputer with three RealityEngine add-in boards. The communications link is established using high-speed digital lines and OC-3 circuits capable of 155 Mbps. The connections are made using Asynchronous Transfer Mode (ATM) switches.

The CAVE system gives users in remote locations the ability to manipulate objects in a virtual environment. The designers give an example where an automotive company in Detroit will be able to assess design feedback from European and Japanese consultants in real-time. The people in Detroit are able to manipulate the design and the consultants will be able to respond.

The CAVE VR system is a high-end solution to using networked VR for collaboration. So far, the system is in the testing stage. However, the designers envision a time in the not-too-distant future when CAVE to CAVE communication will be widely available.

Some Conclusions

In looking at networked VR and a variety of different applications of it, we can draw some conclusions about how we should be considering VRML. The first thing to consider is the trade-off between interactivity and graphic resolution. It is clear from the success of game systems like Virtuality and networks like Habitat and WorldsAway that graphics quality plays a less important role than the level of interactivity. Thus, we need not be overly concerned about image resolutions in VRML browsers and should instead concentrate on developing ways to interact with other users.

Moreover, the need for immersive VR in VRML is not as vital as one might believe. The use of "fishtank VR" is a perfectly acceptable alternative to using HMDs and 3D input devices. The immersive element tends to raise the cost of networked VR in terms of the equipment needed.

The type of interface is also an important factor. While it may be important to keep the VRML interface relatively simple, it is also desirable to provide users with different levels of control. In some cases, users will gain more out of the experience by having more to do in a simulation.

Finally, we need to consider what kind of place we want cyberspace to become. How much control do we want to exercise in the building of our VRML environments? Do we want to have a wide variety of virtual spaces to explore? Who do we want to have access to a global virtual world? And how are we going to encourage those with visions that are different from ours, or who do not have easy access to the technology to join in?

As we begin to build a global virtual world with VRML, we must keep these issues at the forefront of our thinking. A world made up of nothing but square pegs populated by square peg kinds of inhabitants is an uninteresting world. The real world is more interesting, aggravating, and enriching because of its diversity. We can only improve on the richness of a virtual world made up of VRML environments if we can encourage the same diversity.

From Here__

Applications of networked VR provide us with starting points from which we can begin to build a global virtual world. The Internet and the World Wide Web will be the connectors for that world. And VRML will be its building material. To learn about VRML and the Internet, or to start programming your own VRML environments, see the following chapters and parts:

- i [Chapter 6, "VR and VRML,"](#) describes the connection between VRML and virtual reality.
- i [Chapter 9, "An Overview of VRML Browsers,"](#) describes how VRML browsers work with the World Wide Web and the different types that are available.
- i If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment](#)

[from Scratch.](#)" These chapters provide a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



6-The Mechanics of VRML

This chapter is designed to be a brief "history lesson" on the development of VRML. If history bores you, feel free to skip it; you won't miss anything critical. However, if you're curious about where VRML fits into the ongoing history of virtual reality, read on. In this chapter, we'll explore

- The importance of standards
- What VR was like before VRML
- What VRML offers
- How VRML was developed
- The current status of the VRML specification
- What the future may hold

The Tower of Babel

VRML is a very recent development; there were VR systems long before VRML was designed, and each of them has had its own unique way of doing things.

Among the most popular of the VR software packages are VREAM, Superscape, Sense8 WorldToolKit, Autodesk CDK, and Division's dVS system. VREAM and Superscape are designed as integrated authoring environments, whereas WorldToolKit and CDK are designed as libraries to be used by programmers in order to create virtual worlds. Division covers both, with its dVS and dVISE products.

Each of these systems has its own file format for storing virtual worlds, and no two systems can exchange data. If you build a world using VREAM, you can't bring it into WorldToolKit; if you create a VR application using CDK, you can't bring it into Superscape. In short, there have been no standards.

The only common file format that everyone has been able to import is DXF, a format originally designed by Autodesk for CAD applications. While it's well-suited to that market, DXF has a lot of problems. It is an ugly format, difficult for human beings to read or write. It has a lot of different ways of doing exactly the same thing, and very few of the VR systems actually support the entire DXF specification. Worse yet, DXF doesn't have any way of specifying object hierarchy, material properties, texture maps, lighting, or any of the other things that are important in VR.

Here's an excerpt from a DXF file:

Listing 6.1

```
0  
SECTION  
2  
HEADER  
9  
$ACADVER  
1  
AC1009  
0  
ENDSEC  
0  
SECTION  
0  
SECTION  
2  
BLOCKS  
0  
ENDSEC  
0  
SECTION  
2  
ENTITIES  
0  
POLYLINE  
8  
0
```

66

1

10

0.0

20

0.0

30

0.0

70

48

71

2

72

10

0

VERTEX

8

0

10

9.8231

20

0.0

30

0.0

70

64

0

VERTEX

8

0

10

7.91327

20

5.877853

30

0.0

70

64

0

VERTEX

8

0

10

2.91327

20

9.510565

30

0.0

70

64

0

SEQEND

8

0

0

ENDSEC

0

EOF

Not very readable, is it? To give you some idea of how old the DXF standard is, it's written to be easily parsed by a Fortran program. However, some things should be clear in the example above: it defines a POLYLINE made up of a number of VERTEX entries. The meaning of these things will become clear when we discuss 3D graphics.

VREAM script files are a little more reasonable:

Listing 6.2

VREAM_VERSION,S,0.0802
GRAPHICS_MODE,S,16;
COLORS16,S,0,15,5,59,57,58,10,7,56,54,14,38,20,39,44,63;
COLOR_SHADES,S,
244,255, 208,223, 192,207, 160,175,
176,191, 128,143, 144,159, 228,239,
240,255, 96,111, 112,127, 64,79,
80,95, 32,47, 48,63, 224,239;
BACKGROUND_IMAGE,S,NAME.PCX,0;
VIEWER_POSITION,S,0.0,-10.0,1.5;
VIEWER_ROTATION,S,0,0,.0,0.0;
LINKS,S,0;
INFORMATION,S,0;
SORT_METHOD,S,0;
SORT_WIREFRAME,S,0;
SORT_SECONDARY,S,0,0;
OBJECT_ELIMINATION,S,0;
DETAIL_ELIMINATION,S,0;
STEREOSCOPIC,S,0,0,0.2500;
AUTO_MOTION,S,1;
ALWAYS_SELECT,S,1;

GRID,S,1;

ENVIRONMENT,S,0;

SHADING,S,0;

LIGHT_SOURCE,S,0.0,-1.0,0.0;

SCREEN_SIZE,S,-1,-1,-1,-1;

SOLID,S,1;

WIREFRAME_S,1;

HIDDEN_SURFACE,S,1;

BACKGROUND_COLOR,S,15

FOREGROUND_COLOR,S,0;

HAND,S,0;

HAND_DESCRIPTION,S,13,2,1,1.0,11.0;

GRAVITY,S,0,32.0;

GRAVITY_BASE,S,0.0;

VIEWER_MOVE_SPEED,S,1.0;

VIEWER_ROT_SPEED,S,1.0;

Cube,S,

-2.0,-2.0,2.0,

-3.0,-2.0,2.0,

-3.0,-1.0,2.0,

-2.0,-1.0,2.0,

-2.0,-2.0,3.0,

-3.0,-2.0,3.0,

-3.0,-1.0,3.0,

-2.0,-1.0,3.0,

0.0,0.0,0.0,

6,6,6,6,6,6;

ACTIVATE_SOLID,S,1;

ACTIVATE_WIREFRAME,S,1;

WIREFRAME_SOLID,S,4;

_VISIBILITY_VALUE,S,1;

As you might guess from the last part of the file, this VREAM script creates a cube consisting of eight vertices. VREAM has several new products coming out that will support VRML, including its new VRCreator package and a VRML browser called WIRL. Check its Web site (<http://www.vream.com>) for details.

Sense8 takes a different approach with its WorldToolKit package; it uses the file format only for defining the object geometry, and left behavior, lighting, and other issues up to the application. Here's a sample ".nff" file:

Listing 6.3

nff

version 1.60

viewpos 0.000000 0.000000 0.000000

viewdir 0.000000 0.000000 1.000000

object3

20

210.000000 -30.000000 190.000000

210.000000 -60.000000 200.000000

240.000000 -60.000000 200.000000

240.000000 -30.000000 190.000000

190.000000 -30.000000 180.000000

190.000000 -30.000000 220.000000

210.000000 -30.000000 240.000000

240.000000 -30.000000 240.000000

260.000000 -30.000000 220.000000

260.000000 -30.000000 180.000000

240.000000 -30.000000 160.000000

210.000000 -30.000000 160.000000

210.000000 -20.000000 160.000000

240.000000 -20.000000 160.000000

260.000000 -20.000000 180.000000

260.000000 -20.000000 220.000000

240.000000 -20.000000 240.000000

210.000000 -20.000000 240.000000

190.000000 -20.000000 220.000000

190.000000 -20.000000 180.000000

11

4 3 2 1 0 0xff both _T_earth

8 19 18 17 16 15 14 13 12 0x7ac _V_wood

8 11 10 9 8 7 6 5 4 0x7ac _V_wood

4 4 5 18 19 0x7ac _V_wood

4 5 6 17 18 0x7ac _V_wood

4 6 7 16 17 0x7ac _V_wood

4 7 8 15 16 0x7ac _V_wood

4 8 9 14 15 0x7ac _V_wood

4 9 10 13 14 0x7ac _V_wood

4 10 11 12 13 0x7ac _V_wood

4 11 4 19 12 0x7ac _V_wood

Again, we see the familiar lists of numbers; these are the coordinates of the vertices that make up the object.

REND386 (and AVRIL, VR-386, and Gossamer) use a ".wld" file format that looks like this:

Listing 6.4

options h

loadpath sample

light 10000,10000,10000

worldscale 1.0 # 1.0 mm per unit

hither 10 # anything closer than this gets clipped

yon 200000 # anything farther than this doesn't get _rendered

camera 1 7488,1600,1372 0,-89.5,0 1 # starting location, orientation _and zoom

Surface definitions:

surfacedef plaster 0x1fd8

surfacedef brick 0x11ff

surfacedef wood 0x12df

Surface mappings:

surfacemap single1

surface 1 plaster

surfacemap single2

surface 1 brick

surfacemap single3

surface 1 wood

polyobj 4 carpet 800,0,0 800,0,2800 4000,0,2800 4000,0,0

polyobj 6 carpet 4000,0,2800 4000,0,4800 5600,0,4800 8800,0,4800 _8800,0,0 4000,0,0

polyobj 6 carpet 0,0,4800 0,0,10400 2400,0,10400 5600,0,10400 _5600,0,4800 4000,0,4800

polyobj 4 carpet 5600,0,10400 8800,0,10400 8800,0,4800 5600,0,4800

polyobj 5 carpet 2400,0,12800 8800,0,12800 8800,0,10400 5600,0,10400 _2400,0,10400

object bigwin 1,1,1 0,0,0 8800,0,0 0 wind2

object wall1 80,48,4 0,90,0 8800,0,12800 0 single3 fixed # H

area 200,1500,5000 Office

area 1000,10,1000 Gallery

area 1000,10,3000 Backyard

object rug 10,1,10 0,-90,0 7500,1,7500 0 rugcolors

object art 20,20,1 0,180,0 6500,1200,5 0 piccolors

object picture 20,20,1 0,180,0 4000,1200,4805 0 pic2colors

object vwlogo 6,6,6 0,180,0 4500,1800,10406 0 pic2colors

object shelf 4,2,2 0,-90,0 0,1400,5800 256 . fixed

object tree2 12,12,12 0,0,0 800,1,12000 0

object tree2 8,8,8 0,0,0 -800,1,14000 0

object tree2 14,16,14 0,0,0 -1200,1,19000 0

figure mm=metaldan 1.3,1.5,1.3 0,-90,0 1400,800,1200

Tasks:

task sculspin 25 c6;0,-15,0

task sculspin 25 mh;0,8,0

title memory

End of sample.wld

This file format is slightly different, in that it references other files (which are actually stored in a different format called ".plg").

The important thing to notice about all these file formats is how different they are from one another. The differences are not just syntactic, but conceptual as well, which is what makes it so difficult to convert from one to the other.

With all these different VR systems and their various file formats, there has been a great deal of confusion. Clearly, a standard was needed.

Enter the Internet

At around the same time as these VR systems began to mature, the Internet hit the knee of its acceptance curve and began growing exponentially. The advent of the World Wide Web was the single most important factor in that growth, since it made it possible for people to access information on the Internet through a user-friendly graphical front end. HTML was, of course, the language that allowed this to happen.

It didn't take long for people to realize that the fusion of virtual reality and the Internet had tremendous potential. What was needed was the conceptual equivalent of HTML, a language that could be used to store three-dimensional worlds the same way HTML let us store documents. People could run "helper applications" under their Web browsers, which would allow them to access and view these worlds. It would bring depth to the Internet, in more ways than one.

How It Began

In 1994, Mark Pesce and Tony Parisi sat down and wrote a program called Labyrinth. Mark had access to a software package called RealityLab, from a British company called RenderMorphics (which has since been bought out by Microsoft). RealityLab is a library of routines that does 3D graphics entirely in software, and it runs very fast, indeed. It's one of three packages that came out around the same time, all in England; the other two are RenderWare from Criterion software and BRender from Argonaut.

Labyrinth wasn't really a VRML browser, since VRML hadn't been developed yet; it was more of a

"proof-of-concept." Labyrinth had the ability to retrieve three-dimensional objects over the Web, using the same protocols as regular Web pages used. Mark presented Labyrinth at the First International Conference on the World Wide Web in May of 1994 in Geneva, Switzerland. An electronic mailing list was set up for discussions of VRML, which originally stood for "Virtual Reality Markup Language" (since HTML was the HyperText Markup Language). The "Markup" was later changed to the more appropriate "Modeling," since storing 3D models was what VRML was all about.

After months of discussion and consideration of a number of alternatives the initial VRML specification was designed. It bore a lot of similarity to a format called OpenInventor, originally developed at Silicon Graphics (a maker of computer workstations). Although VRML has since begun to evolve away from OpenInventor, it made sense to use an existing format as a jumping-off point. The original specification was drafted by Gavin Bell of SGI, Mark Pesce, and Tony Parisi.

The draft specification for VRML was presented at the Second International Conference on the World Wide Web, held in October of 1994 in Chicago. A couple of months later, a parser for the language was released into the public domain. That parser (which is an important part of a VRML browser) was called QvLib; it had a number of bugs and proved to be difficult to port to other platforms, but it was an important starting point for anyone writing VRML tools.

Current Status

Since then, there has been considerable work done on improving the standard. As it turns out, there were a number of aspects of the original VRML specification which made it difficult to implement on many systems; it also had some built-in inefficiencies. There were also some ambiguities in the specification, and some areas that needed clarification.

Work began in earnest on the VRML 1.1 specification, with an eye towards an eventual 2.0 spec.

This book is being written during the month of October, 1995. At the moment, it's not clear what the status of VRML 1.1 is, though it should be stable by the time you actually read this. In this book, I've adopted the approach of sticking to the VRML 1.0 spec but not relying on anything that's scheduled to be dropped in 1.1. Chapter 19 will discuss the proposed changes for Version 1.1, and they're summarized in Appendix C. Similarly, Chapter 27 will examine some of the proposed changes for VRML 2.0, which are summarized in Appendix D.

If you want to find the latest VRML specification and related documents, start off at <http://sdsc.edu/vrml> and look under the documentation section; everything you need is there.

The Future

VRML is more than just a file format, of course. It's an idea, a dream, a goal. In order to understand the importance of VRML, you need to have a vision of what will be possible once the specification matures and more powerful hardware and software become available.

Let's consider a possible scenario.

You arrive home after a long day at work. After grabbing a quick bite to eat, you head downstairs and

power up your P6-based personal computer. You decide you want to visit your friend Vicki, so you fire up Netscape and surf over to her home page. You click on her "homeworld" icon, then walk across the room to your VR rig.

You slip on a pair of lightweight gloves, put the glasses on, and step onto the tracking platform; by the time you've done this, a connection has been established and the current state of Vicki's world has been transferred over the Internet into your system (in VRML, of course). Simultaneously, your custom-designed virtual body (your "avatar") appears to Vicki. As each of you moves around, your movements are echoed by your avatars. The two of you wander through cyberspace, exploring strange new worlds together.

Is this science fiction? Today, yes. Tomorrow, no; all the necessary infrastructure (the Internet, fast computers, low-cost HMDs, and of course, VRML) is falling into place. With the tools and techniques we'll be describing in the chapters ahead, there's no reason why you can't be a part of making it all happen.

From Here

Now that you have an overview of where VRML came from and where it's going, you may want to move on to [Chapter 7](#) and explore the mechanics of the language itself. If you prefer, you may want to leap ahead and read more about the future of VRML in Chapters [19](#) and [27](#). For even more esoteric visions of the future, check out Chapter [28](#) on behavior and Chapter [29](#) on distributed VR. If you're feeling more philosophical than technical, jump all the way to [Chapter 30](#) and explore the "Zen" of VRML.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



7-The Mechanics of VRML

I realize the title of this chapter makes it sound like a pulp science fiction novel of the 1950s, but don't let that fool you. This is perhaps the most important chapter in the book, since it gives you the basic foundation you need for creating VRML files. It is essential that you understand the concepts presented here before moving on.

This chapter will:

- Explain the "life cycle" of a world created using VRML
- Describe the basic concepts of three-dimensional graphics
- Introduce important terminology that will be used in later chapters
- Explore in detail the "graphics pipeline" that's inside every VRML browser

Let's start by looking at the birth, transfer, and rendering of a VRML file.

The VRML Life Cycle

Every VRML file starts its life as a vision in the mind of its creator. A plan is drawn up, and various tools are used to create the actual VRML file itself. The file is then made available on a Web server, in exactly the same way that an HTML document is. It's given an extension of ".wrl," so that it will be recognized as a VRML "world" file.

In order for the file to be properly accessed, the Web site must understand VRML files. This is just a one-line addition to a configuration file, done by the Web site administrator (the "webmaster")—a two-minute chore that only needs to be done once. Once it's done, any number of users can make any number of VRML files available over the Internet.

You can place a link to your world from your regular home page, just as you would for an HTML document. Some time after the VRML file is put on the Web site, someone reads your home page and sees a reference to your "home world." They click on the link, and their browser (for example, Netscape) sends out a request to the Web site asking for information about the file. The Web server notices that the file being requested has a ".wrl" extension, so it checks the configuration file that the webmaster set up

and finds out the MIME content type to use; it turns out to be "x-world/x-vrml." It sends this information, along with the file size, date, and other details to the user's browser.

The user's browser in turn checks its local configuration file, and finds that in order to view a file of MIME type "x-world/x-vrml" it must run a specific program called a VRML browser. Fortunately, the user happens to have just such a program; Netscape launches it as a "helper app," and the VRML browser reads in the file.

The first thing the VRML browser does is "parse" the file. Parsing just means reading the file and building a set of data structures in memory that describe the virtual world; the VRML file can be thought of as a kind of cookbook for assembling the world.

Once the data structures are built, the browser uses something called a "rendering engine" to actually draw images of the world on the screen; as the user navigates around, the browser keeps calling on the renderer to do the work of updating the screen. Occasionally, the VRML browser may have to load resources (such as texture maps, or other VRML files) from the Web, but it does this quickly and transparently.

If you click on an anchor-object in a VRML world, it links you to another spot on the Web; it might be another VRML file, or it might be an HTML document. Anchor objects are just like anchor tags in HTML, and they're explained in detail in Chapter 15.

That's all there is to it.

Rendering Engines

You may or may not be familiar with the phrase "rendering engine." Rendering is the process of drawing something; you sometimes hear things like "no photo is available, but here's an artist's rendering of the suspect." An "engine" is just a piece of computer software that performs some specific job, so a "rendering engine" is the piece of software that does the job of drawing the scene on the display.

There are a number of commercially available rendering engines that people are using as the basis for their VRML browsers. Some people have written browsers around the RenderWare engine, others around the Rendermorphics/Microsoft Reality Lab, still others around Intel's 3DR or even OpenGL. Each of these has its strengths and weaknesses; for example, it's easy to write a VRML browser using OpenGL, but the result is usually a very slow browser indeed.

These rendering engines have their own individual kinds of data structures, so one of the most important (and complicated) tasks a browser has to perform is to convert VRML into those data structures. Even though no two rendering engines use the same structures, there are certain basic things they have in common. By examining those common elements, we'll gain a lot of insight into how VRML works.

Data Structures

A data structure is just a collection of information, organized in a specific way and stored in the computer's memory. Most rendering engines treat the virtual world as being made up of a number of objects, each of which consists of a set of vertices and a set of polygons (or "faces"). Some people have a

bad impression of polygon graphics, simply because they've seen a lot of clunky flat-shaded games and demos. When they see a lit, texture-mapped polygon environment, they don't realize they're still looking at polygons.

VRML is all about vertices and polygons. Everything created inside VRML is composed of those two basic elements, so we're going to spend some time getting a handle on just what they are. In order to understand polygons, we need to understand vertices; to understand vertices, we need to know about coordinate systems.

Coordinate Systems

Look at the far left corner of your desk. For a moment, imagine that it's a mathematical point, and that it's the center of the entire universe; everything is measured relative to that specific point. (If you're like me, you probably already feel that way about your desk!)

Now imagine that there are imaginary lines, called axes, that cross through the corner of your desk; let's say the long side of your desk is called the X axis, the short side is the Z axis, and the Y axis points straight up towards the ceiling. Every point in the entire universe can be measured relative to the corner of your desk, using just three numbers: one giving the distance along the X axis, one along the Y axis, and one along the Z axis.

Those three axes form a "coordinate system," and the corner itself (where X, Y, and Z all have the value zero) is called the "origin" of that coordinate system. See Figure 7.1.

Fig. 7.1

The "Desk" coordinate system.

You may already be familiar with X-Y graphs from school; if so, this will be old hat. All we're doing is adding a Z axis, that points towards you from the origin.

Figure 7.1 illustrates a "right-handed" coordinate system. To understand why it's called that, take a fine-point felt marker and draw a big 'X' on your right thumb. Now draw a 'Y' on your index finger (the one you use for dialing a telephone or pushing an elevator button), and a 'Z' on your middle finger (the one you use for making a rude gesture). If you hold your fingers so that the thumb, index and middle fingers are all at right angles to each other (as shown in Fig. 7.2), then they'll form a right-handed coordinate system.

You can, of course, do the same thing on your left hand; the result would be called a "left-handed" coordinate system. It should be obvious that no matter how much you turn your hands around, they'll never be the same; they're mirror images of each other. The same is true of coordinate systems; left-handed and right-handed systems are different, and it's not just a question of looking at it from another angle.

It's completely arbitrary whether you use a left-handed coordinate system or a right-handed one; different graphics software packages do different things. However, when it comes to exchanging virtual worlds, you have to agree on a standard; VRML uses a right-handed system, so that's what we'll use in this book. The X axis points to our right, the Y axis points up, and the Z axis points towards us.

Vertices

A vertex is just a mathematical point; it consists of three numbers (the X, Y, and Z coordinates). If you take your felt marker and put a dot on the tip of your index finger, that point can be thought of as a vertex. No matter how much you move your hand, the location of that vertex can be given by a set of X, Y, and Z coordinates.

If you've read Chapter 4, this should all seem familiar; your finger can move with three degrees of freedom, by "translating" along the X, Y, and Z axes.

A virtual object is made up of a number of these vertices, joined together by lines or faces.

By the way, we sometimes talk about "vectors"; a vector is a direction, specified by three numbers. You can compute a vector between two points by subtracting one from the other; for example, the vector from [2 7 5] to [9 12 18] is [7 5 13].

Lines

Two vertices can be joined together to create a line. If you have a number of vertices joined by lines, you get a "wireframe," as shown in Figure 7.3.

[Fig. 7.3](#)

A wireframe of a cube.

In a cube, there are a total of eight vertices (four on the top, four on the bottom). They are joined by twelve lines.

Faces

A face (also called a "facet" or "polygon") is a flat surface that joins a number of points. A cube is made up of six faces; four sides, plus top and bottom. Each of the faces of a cube is bounded by four lines, and connects four vertices. Faces don't have to be square, and they don't necessarily have to have four sides.

Let's construct a cube, starting with the points:

[Fig. 7.4](#)

The vertices making up a cube, with the X, Y, and Z coordinates marked.

The coordinates of the vertices range from -1 to +1; this means that the origin of the cube's coordinate system (0,0,0) is at the center of the cube. Now, let's assign a number to each vertex:

[Fig. 7.5](#)

Numbering the vertices.

There are eight vertices altogether, numbered from zero to seven (computers always start counting from zero, instead of one). It makes no difference what order we assign these numbers in. We can then write down a table which gives the coordinates of all eight vertices:

Table 7.1 The Coordinates of the Cube's Vertices

X	Y	Z
0	-1	-1 1
1	-1	1 1
2	1	1 1
3	1	-1 1
4	1	-1 -1
5	1	1 -1
6	-1	1 -1
7	-1	-1 -1

What we've done is create a simple data structure to represent the vertices of our cube.

The next step is to define the faces. As shown in Figure 7.6, each face can be specified by listing which vertices it uses; for example, the side of the cube facing to our right uses vertices 3, 2, 5, and 4. We list the vertices of each face in a clockwise order, as seen from outside the cube. We could use counter-clockwise ordering; the only thing that matters is that we're consistent. Let's assign a letter to each face, as shown in Figure 7.6:

[Fig. 7.6](#)

A letter on each face.

Now we can construct a second data structure to represent the faces:

Table 7.2 The Vertices Used by Each Face

A:	0 1 2 3
B:	1 6 5 2
C:	3 2 5 4
D:	4 5 6 7
E:	7 6 1 0
F:	0 3 4 7

The result is a data structure that describes the faces of the cube.

The array of vertices and array of faces are among the most important data structures we'll be examining. When we discuss the syntax of VRML later in the book, we'll be referring back to these two arrays.

Objects and Transforms

The two arrays (vertices and faces), plus some other data, define a virtual object. The example we've been using is a cube, but more complex objects are created in exactly the same way, as shown in Figure 7.7.

[Fig. 7.7](#)

A more complex object.

There are a number of operations you can perform on objects. These operations are called "transformations" or simply "transforms," because they change the object in some way. There are three basic types of transformation: scaling, rotation, and translation. Rotation and translation should be familiar concepts already, if you've read Chapter 4; they correspond to the six degrees of freedom.

Scaling

Scaling simply refers to stretching or shrinking an object, and it can be represented by three numbers: the scale factor in each of the X, Y, and Z axes. For example, if we stretch our cube in the X direction it looks like Figure 7.8:

[Fig. 7.8](#)

The cube, scaled by a factor of 2.0 in the X direction.

And if we stretch it in the Y direction, it looks like Figure 7.9:

[Fig. 7.9](#)

The cube, scaled by a factor of 2.0 in the Y direction.

Rotation

Rotation can be specified in a lot of different ways. For example, you could give three angles which are the rotation of the object around each axis. These are called "Euler angles," and they're usually given in the order Y rotation, X rotation, Z rotation (yaw, pitch, and roll). The order in which the rotations are performed is important, since the end result will be different depending on the sequence.

Another way to specify rotation is in "axis-angle" form. This involves a three-element vector and a rotation angle; the object is simply rotated around that axis by that angle. For example, you might say "rotate an object by 45 degrees around the vector [0 1 0]."

Let's rotate our cube by 45 degrees around the Y axis:

[Fig. 7.10](#)

A 45 degree rotation around Y.

Translation

Translation refers to moving an object. It's specified by a three-element vector that gives the direction along which to move the object (in other words, the amounts to move the object along each of the X, Y, and Z axes). Figure 7.11 shows the cube translated along the vector $[1\ 2\ 3]$.

[Fig. 7.11](#)

Translation.

Order of Operations

The order in which scaling, rotation, and translation are performed is important; scaling and then rotating is not the same as rotating and then scaling. Compare Figures 7.12 and 7.13:

[Fig. 7.12](#)

Scaled by $[2\ 1\ 1]$, then rotated around $[0\ 1\ 0]$ by 45 degrees.

[Fig. 7.13](#)

Rotated around $[0\ 1\ 0]$ by 45 degrees, then scaled by $[2\ 1\ 1]$.

Local vs. Parent vs. Global Coordinate Systems

Transformations can be thought of as a way of converting an object to a different coordinate system.

Let's go back to our "desk" coordinate system. Imagine that you actually build a cube out of chicken wire, and put paper on it to create the faces. If you glue it to your desk, it's fixed to that coordinate system (the one that has its origin at the corner of the desk).

However, the desk's coordinate system isn't anything special (despite what we said earlier). You could easily define another coordinate system, with its origin at the corner of the room down by floor level. The long axis of the room is X, the short axis is Z, and up is Y. A vertex in the cube has a certain displacement from the corner of the desk, but it also has a displacement from the corner of the room. Not only that, you can pick up your desk and move it somewhere else. The desk (with the cube still attached to it) is now in a new location and orientation (having moved with 6 DOF), so the coordinates (relative to the room) of a vertex on the cube are now different. However, the coordinates of that vertex relative to the desk have remained the same.

In other words, if a particular vertex of the cube were located two feet from the left of the desk, one foot from the back of the desk, and a foot and a half above the desk, then it would be located at $[2\ 1.5\ 1]$ in the desk coordinate system. If the desk were four feet from the short side of the room, and three feet from the long side, and two and a half feet high, the desk's origin (its top far left corner) would be located at $[4\ 2.5\ 3]$ in the room coordinate system. The vertex on the cube would be located at $[6\ 4\ 4]$ in room coordinates.

[Fig. 7.14](#)

The basic idea is that transforms not only scale, rotate, and translate objects, they also define a relationship between two coordinate systems. When you build an object (like our cube) you can put its origin anywhere you like; it will get transformed later into world coordinates.

Hierarchies

VR systems organize objects into hierarchies. A hierarchy looks like a tree, or like an organization chart, or a diagram of a DOS or UNIX file system.

Consider the human body; it's made up of a number of different objects (your upper arm, your left foot, your head, and so on). They're arranged into a hierarchy; if we say that your pelvis is the root of the tree, it looks something like Figure 7.15:

[Fig. 7.15](#)

The human body as a hierarchy.

Each object has a "parent," and may also have "children." The pelvis has the chest and the two upper legs as children; the chest, in turn, has the head and two upper arms as children. In a hierarchy, movement of higher-level objects affects lower-level objects; if I bend at the waist, my chest moves and takes my head and arms along for the ride. The motion of lower-level objects has no effect on upper-level ones; if I turn my head, my chest and pelvis can stay where they are.

If you were assembling a human body in VR, you might create an object for each segment; in other words, you'd craft a left thigh, a right forearm, and so on. You'd then assemble the body, using a series of transformations to scale the parts to be the right size, rotate them to the right orientation, and translate them to the right location relative to each other. These transformations can be combined (as described in Appendix A), so each object would have a single, cumulative "transform" that relates it to its parent.

The root of the human body hierarchy is the pelvis, but even that isn't the real root of the tree; if you're driving around in a golf cart, for example, your pelvis is (temporarily) attached to the cart, which itself is attached to the world as a whole. Just as each body part has its own coordinate system, there's a coordinate system for the golf cart and one for the world. To find out the location in world coordinates of a vertex on your right hand (such as the dot you put on your index finger earlier on), you would apply the transforms in the following order:

- transform from fingertip coordinates to hand coordinates
- transform from hand to forearm coordinates
- transform from forearm to upper arm coordinates
- transform from upper arm to chest coordinates
- transform from chest to pelvis coordinates
- transform from pelvis to golf cart coordinates
- transform from golf cart to world coordinates

Each transform consists of a scaling, a rotation, and a translation. Of course, some of the translations can be [0 0 0], some of the rotations can have a zero angle, and some of the scalings can be [1 1 1] (which is to say, no scaling at all). By accumulating these transforms up the hierarchy, you can find out the world-space location of any object-oriented vertex.

Data Structures Revisited

So, to recap, the VRML browser reads the file and builds these data structures. It reads (among many other things) the vertices and faces that describe the geometry of the objects, as well as information about lighting, and more; it also keeps track of the object hierarchy as it's read in, and builds a corresponding hierarchical set of data structures in memory.

Once this is done, it calls the rendering engine to do the dirty work.

The Graphics Pipeline

At first, you might think that the internal workings of the browser at the level of the graphics pipeline aren't relevant to building VRML files. However, much of the discussion in the chapters ahead will be about creating efficient VRML worlds that run at a reasonable speed on widely available hardware. Doing so requires a certain amount of craftsmanship, and a lot of insight. The rest of this chapter is devoted to giving you that insight.

Every time the renderer re-draws the screen, it has to run through something called the "Graphics Pipeline." This is in fact the job of the rendering engine; it executes the graphics pipeline from beginning to end. Let's take a look at what's involved.

Local to World Transforms

This step was outlined above; essentially, the system must traverse the tree and compute the location, scale, and orientation of every object. When you move your chest relative to your pelvis, the updates to the chest transformation must propagate accordingly down the hierarchy. This is performed as a series of accumulating transforms, ending up with every vertex of every object transformed into world coordinates.

Lighting

The next step is to perform lighting calculations for every vertex; this will be described in excruciating detail later in the book, but for now just keep in mind that every vertex will have a certain intensity of light associated with it that's the result of all the lights in the virtual environment.

Viewing Transform

Once the vertices have been converted to world space and had their lighting values calculated, they are transformed one more time. This brings them into the view coordinate system, which has the user's virtual eye (or camera) at the origin; it points into the negative Z direction. At this point, the scene looks something like Figure 7.16.

Fig. 7.16

The viewing frustum.

Object Culling

Notice that the eye can only see objects within a certain viewing angle, both horizontally and vertically. Objects outside this "viewing frustum" (a sideways pyramid with its tip cut off) are not visible, and don't need to be processed any further; determining which objects can be ignored is known as "object culling."

Clipping

The next step is clipping; each individual face is checked to see if it's entirely inside the viewing frustum, entirely outside, or partway in between. Faces that are outside are ignored, faces that are inside are processed as-is, and those that overlap are clipped so that only the parts that are inside the frustum are processed.

Backface Removal

If you consider our cube example, it should be clear that you can only see (at most) three of the cube's six faces at any one time. This will be true of all the objects in our virtual world; roughly half the faces in the scene will be turned away from us at any given time. Backface removal is used to eliminate those faces, instead of wasting time drawing them.

Perspective Projection

For centuries, artists have used a technique called "perspective" to give a sense of depth to their paintings; since we want our virtual worlds to appear to have depth as well, we need to calculate this perspective effect. Technically it's done by dividing the X and Y values by the Z value for each vertex.

Screen Scaling

The resulting X and Y coordinates for each vertex are then scaled and shifted so that they appear in the correct place on the display screen.

Hidden Surface Elimination

This is an important step in the pipeline; it ensures that objects appear to be in their correct depth order. If you wave your hand in front of a virtual object, but the hand is drawn first and the object second, it appears that your hand is behind the object instead of in front of it. It's necessary to do some type of sorting, in order that faces that are hidden behind nearer objects don't get drawn over them.

There are two solutions: draw the faces in the correct order (which is difficult) or draw them in an arbitrary order and use a Z-buffer (which is what most modern rendering engines do).

Rasterization

Rasterizing involves actually drawing the faces of the object on the screen. It's the final stage in the

graphics pipeline, and by far the slowest in a software-based system (which is why 3D graphics accelerators are often used to do this step). It's at this stage that shading is done.

Shading

There are three basic ways to render faces on the screen. The first is wireframe; only the outlines of the faces are drawn, as shown in Figure 7.17.

This is usually very fast, since it involves the least computation (just drawing lines). Unfortunately, it results in objects that look like they're made out of chicken wire.

[Fig. 7.17](#)

Wireframe mode.

The second is flat shading; the faces themselves are drawn, but each face has a single color whose brightness is proportional to the amount of light falling on it. The result looks like Figure 7.18.

[Fig. 7.18](#)

Flat shading mode.

This is reasonably fast, since very little work has to be done on a per-pixel basis. The result of flat shading is that everything appears to be rough and faceted. This is the kind of shading that most people think of when they think of a really cheesy VR system.

The third type of shading is Gouraud, named after the guy who invented it. It involves interpolating the light intensity across each face, to smooth out the rough edges; it's sometimes called "smooth shading." It looks like Figure 7.19:

[Fig. 7.19](#)

Gouraud shading mode.

In addition to these three modes, you can also turn on dithering. Dithering is a technique for eliminating the "banding" that you often see with Gouraud shading; it alters the density of pixels to produce a more seamless effect.

And finally, there's texture mapping. Texture mapping will be described in detail in Chapter 14, but basically it involves taking a flat 2D image and applying it to a face of an object to increase its realism. Figure 7.20 illustrates the effect.

Each higher level of functionality makes the system slower. Wireframe is usually the fastest, with flat shading a close second; Gouraud shading really slows things down, especially if you're dithering, and texture mapping slows it down even more.

[Fig. 7.20](#)

Texture mapping.

Performance

In a software-based system (in other words, one that doesn't have a 3D graphics accelerator) the per-pixel operations are what really kill the speed. Per-pixel operations are Gouraud shading, texture mapping, and Z buffering. The performance also suffers as the resolution of the image increases; at 320 by 200, you might get acceptable performance, but if you go to 1024 by 768 you'll slow to a crawl.

Graphics accelerators deal with that quite effectively; they handle all the per-pixel operations, so the main CPU (i.e., the 486 or Pentium or what have you) only has to handle the "high-level" tasks like transforms. Some graphics accelerators (the more expensive ones, of course) will also do the transforms for you.

Order of Operations

The graphics pipeline described above is just one possible way of doing things. In practice, you'd do the object culling before doing any of the transforms; you'd also do your lighting (and probably your backface removal) in object space instead of world space. Those details shouldn't affect you, though; the main thing is to visualize the pipeline as having two parts: the math and the pixels. The pixel-processing can be speeded up by extra hardware, but the math (transforms and lighting) are generally done by the main CPU.

Bottom line: if you're using a software-only system, keep your image resolution low and turn off features if you don't need them (like texture mapping and Gouraud shading). If you have a graphics accelerator, things'll be much faster; still, keep the number of faces (i.e., polygons) down, to minimize the number of vertices that need to be transformed.

From Here

Appendix A covers the math behind transformation, lighting, hidden surface removal, and more; if you want that kind of grungy detail, I'll meet you there later. For now, you should probably move on to [Chapter 11](#) and start getting familiar with the intricacies of VRML itself.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



8-What Do You Need to Use VRML?

Now that you have decided that you want to view VRML environments and maybe even create your own 3D objects and spaces, you need to think about the machine that makes it all possible: your computer. Not all computers are created equal—especially when it comes to VRML.

This chapter examines what you will need in order to browse and create VRML environments and objects. It focuses on computer hardware—the processor, graphic accelerator boards, and memory—that you must have to use VRML, and the hardware that you can add to make your VRML experience even better. The topics discussed include:

- The types of computers that can use VRML.
- The hardware you need to connect to the Internet.
- The importance of memory and processing speed.
- Ways to improve your computer's performance.

What's Your Computer?

To use VRML, the first thing you need to have is a computer. This is an obvious statement. And you can use any computer to create VRML environments. What's not so obvious is knowing which computers can actually view it. Even though VRML is a platform independent language, there are certain hardware requirements that need to be considered.

PC-Compatible

A PC computer (or simply PC) refers to IBM-compatible computers, but not all PCs can use VRML. Theoretically, you can run a VRML browser on a PC using a 386 processor and 8 MB of memory. However, the browser will run, if it runs at all, at an excruciatingly slow speed—even if you add a math coprocessor. Many browsers for this platform recommend that you have at least a 486 processor running at 50 MHz. However, I have found that to get smooth frame-by-frame movement, you really need to be using a 100 MHz 486. A Pentium would be even better.

Another important element to consider is your display. While you can view a VRML environment using a 16 color VGA screen with no graphics accelerator card, you will find that you have a better

appreciation of what VRML can do if you are using an SVGA screen with 256 colors or more. Indeed, many VRML browsers have a minimum requirement of 256 colors. Moreover, the performance of your computer will increase if your computer has a graphic accelerator card. As well, you will find that a PCI bus will help make the graphics go faster than if you have an ISA or VLB bus.

Therefore, the minimum configuration for using VRML on a PC is a 486 processor running at 100 MHz with no less than 8 MB of RAM. It must be running with at least Windows 3.1 installed. All VRML browsers for this platform require a version of the Windows software in order to run. Some require you to use a 32-bit version of Windows—that is, either Windows 95 or Windows NT. However, a 100 MHz 486 should be considered to be only a minimum configuration.

A more workable configuration for using VRML is a PC with a 100 MHz Pentium processor and a minimum 16 MB of RAM. With this configuration, you will find that VRML environments render at or close to real-time. Moreover, using a 32-bit operating system like Windows 95 or Windows NT requires that you have a more powerful processor and at least 16 MB of memory.

In the best of all worlds, we would be able to afford all the upgrades that will optimize our VRML browsers. But if you are on a tight budget, the first thing you should get is more memory. Most of the delays you will encounter while navigating a VRML environment are caused by the browser running out of memory to store a scene and having to write data to your hard drive. The next consideration would be to get a better processor. This will speed up the calculations needed to render a VRML environment.

Nearly all the VRML browsers designed for PCs run under Windows 3.1, Windows NT, or Windows 95. There is currently no VRML browser that is designed specifically for OS/2. However, most of the Windows browsers will run perfectly well in OS/2's Windows emulator.

Macintosh and PowerMac Computers

The Macintosh and Power Macintosh (or simply PowerMac) computers have long been labeled as graphics-oriented machines. Therefore, it would seem that the Macintosh and PowerMac computers would be an ideal platform to run VRML. However, at the time of the writing of this book, only one VRML browser is available for this platform: the Whirlwind 3D browser and Web Surfer. Other companies such as Papersoft (WebFX) and TGS (WebSpace) have promised Macintosh-compatible versions of their VRML browsers soon, and they are expected some time in early 1996.

Looking at the Whirlwind browser, it is possible to see the kinds of problems that users may discover as more VRML software becomes available for this platform. For one thing, Whirlwind requires you to have the QuickDraw 3D (QD3D) software loaded on your computer. Moreover, it needs 16 MB of RAM in order to operate. This means that anyone buying the Performa computer bundles will have to buy at least 8 MB of extra memory if he or she wants to run VRML software.

All of the VRML software that is coming for the Macintosh platform will require the System 7 or later operating system. Moreover, there will be more software coming out for the PowerMac system than the older 68000 system.

Silicon Graphics and Other Workstations

If you have one of the graphics workstations developed by Silicon Graphics, Inc. (SGI), you probably don't have to worry about running a VRML browser or worldbuilding toolkit. The minimum requirements for using WebSpace and other SGI-based VRML browsers are far exceeded by the system requirements needed to operate many of the popular applications for this platform such as Alias PowerAnimator and Softimage's graphics software.

If you want to use the SGI-oriented VRML browsers, however, you will need to be running the IRIX 5.3 or 6.1 operating system. You will also need a large amount of hard disk space available. For example, WebSpace requires 21.6 MB to load itself and Open Inventor on to an SGI system. You will also need Netscape 1.1S or later.

As with SGI Workstations, IBM RISC workstations, DEC, and Sun Sparcstations will have enough hardware resources to run the VRML browsers designed for their platforms. UNIX-based systems will need X-Windows to run the VRML browsers created for that operating system.

Connecting to the Internet

The hardware you will use to connect to the Internet is going to be just as important as your computer. Modems, and more recently digital phone formats like ISDN, are now becoming standard equipment with most computers. But what kind of communications equipment do you really need to use VRML?

As was mentioned in Chapter 1, "The Internet: Its History and How It Works," the most common method of making this connection is by using a regular phone line and a modem.

Most PCs and Macintosh computers sold these days are bundled with a 14.4 Kbps modem. But while this transfer rate is adequate for downloading VRML files now, the rising level of complexity and sophistication of artificial environments will soon require faster and faster downloading speeds. Therefore, I recommend that you should have a 28.8 Kbps modem and that you use the compression setting in your SLIP/PPP software. Of course, you will need to make sure that your ISP can handle a 28.8 Kbps connection and the compression standard of your modem (see "Connecting to the Internet via a Modem and Regular Telephone Lines," Ch. 1).

However if you are a business or simply have money to burn, you might think about getting a digital line. Digital phone lines permit much higher transmission speeds than conventional telephone lines. One type of digital line service is the Integrated Services Digital Network (ISDN). As the hardware improves, ISDN will eventually allow up to 128 Kbps on a single line. And while the price of an ISDN line is still quite high, recent trends have shown that these prices are beginning to drop rapidly. In some states, you can reportedly get an ISDN line for only \$25 per month. Medium to large-sized businesses should consider getting a T1 or T3 connection. While these connections are expensive, the amount of bandwidth available through them is necessary if you are planning to hook up a high-traffic server to the Internet.

Memory and Processing Power

If there is one thing that you cannot have enough of when using VRML, it's memory. Your computer needs RAM to store its operating system, run the VRML software, store the VRML files, do the calculations and render the environment. If it does not have enough memory to do all these things in RAM, your computer will either not be able to display the VRML environment you have downloaded, or it will have to swap information to your hard disk—a process that may drastically slow down the performance of your browser.

When I was preparing the chapter on VRML Browsers for this book, I used a computer with a 100 MHz Pentium processor and 16 MB of RAM. The operating system I was using was Windows NT. Whenever I tried to load a large VRML file (1 MB or more), I found a number of the browsers simply crashed. When I checked the system, I found that once the operating systems and the browser software were loaded into memory, I had less than 4 MB of RAM left to load the environment, do the rendering calculations, and render the scene. This was with 16 MB of memory. It would be impossible, therefore, to view the larger environments that have been posted on the Internet (which have file sizes of up to 13.3 MB).

The minimum amount of memory required to use VRML effectively is 16 MB. Any less and your computer will have to constantly swap to the hard disk to compensate for the lack of RAM. If possible, you should upgrade to at least 32 MB. This will ensure that you have enough RAM available to render even the largest environments. If you are planning to do a great deal of multitasking, you should consider getting even more RAM.

Processor speed also has an important role to play in the performance of your computer. Rendering a 3D environment in real-time from a list of coordinates and transformation matrices requires a huge amount of calculation. If you have a faster processor, the renderings will run smoother. When I tried to run WebFX with a 25 MHz 386 with a math co-processor, I found that each time I wanted to move, it would take the computer a second to do all the calculations to render the next frame. In contrast, running WebFX on a 100 MHz Pentium computer gave us smooth, real-time movement.

In short, the guiding principle concerning memory and processor power is "more is better." The more memory and processing power you have, the better will be the performance of your VRML software.

Graphics Accelerators Making Your System a Better VRML Machine

So now that you have a faster machine with more memory, what else can you do to use VRML more effectively? The answer is to get a graphics accelerator card. These cards improve your computer's performance by taking on the task of putting the actual image on to your computer's display. They accomplish this feat by having their own memory and processors.

The following is a list of 3D graphics processors. If you want to find out more about these processors and 3D graphics accelerator boards, see <http://www.cs.columbia.edu/~bm/3dcards/3d-cards1.html>.

ARTIST Graphics 3GA Graphics Processor

The 3GA by ARTIST Graphics is a 2D and 3D graphics coprocessor for the PC platform. It includes built-in Windows acceleration, 2D and 3D lines, flat-shading, Gouraud shading, and texture mapping. Software drivers for Windows 3.1, Windows NT, Windows 95, OS/2 Warp, AutoCAD, Hoops, OpenGL, BRender, RenderWare, Reality Lab, and 3DR are also available.

The 3GA coprocessor is capable of 65,000 texture-mapped and 110,000 Gouraud-shaded polygons per second. It supports 1 to 4 MB VRAM memory and 0 to 8 MB of off-screen DRAM for z-buffers, fonts, etcetera.

3GA boards are available for PCI, ISA, and MicroChannel buses. Prices range from \$395 to \$1,095.

NVIDIA/SGS-THOMSON NV1/STG-2000 Multimedia Accelerators

NVIDIA Corporation and SGS-THOMSON have developed the NV1 and STG-2000 Multimedia Accelerator chips. While these processors' primary design is for speeding up multimedia applications, they do provide some 3D graphics support. The chips include support for texture mapping, lighting effects, and a feature the companies call video texturing: texturing full-motion video on to a 3D object. Boards incorporating these chips range from \$250 to \$300.

Rendition VÈritÈ

Rendition's VÈritÈ is a single chip, 3D graphics engine, GUI accelerator, digital video accelerator, and VGA engine for the PC market. It uses an embedded RISC core and low-cost EDO RAM. The chip supports texture mapping with perspective corrected, z-buffering, alpha blending, fog, and other advanced rendering capabilities. It also includes conventional Windows acceleration.

3Dlabs' GLINT and PERMEDIA Chips

3Dlabs Inc. has developed two processor chips for graphics accelerator boards that are specifically designed to work with 3D graphics: the GLINT and PERMEDIA chips. These chips are licensed to a variety of graphics card manufacturers.

The GLINT chip is capable of processing 300,000 shaded, depth-buffered, and anti-aliased polygons per second. It also provides 32-bit color, 2D and 3D acceleration, an on-chip PCI-compliant local bus interface, and integrated LUT-DAC control. It can perform complex rendering operations like Gouraud shading, depth buffering, anti-aliasing, and texture mapping.

According to Jonathan Hirshon, the public relations person for 3Dlabs, the GLINT chip is intended for users wanting to upgrade their PCs to the graphics workstation level. It can be used on both PCs and Macintosh computers. The boards using this chip start at around \$1,000.

A cheaper alternative is the new PERMEDIA chip that was also developed by 3Dlabs. Like the GLINT chip, the PERMEDIA chip will improve the performance of VRML applications. It can process 500,000 textured polygons per second and 25 million pixels per second. It will run at 32-bit color for 2D applications and 16-bit color for 3D applications.

An important aspect of the PERMEDIA chip is its ability to be used on different computer platforms. PC and Macintosh/PowerMac boards are already in development. It will have one of the widest ranges of 3D API support, including OpenGL, Autodesk's Heidi, Apple's QuickDraw 3D, Reality Lab, and Direct 3D.

However, the most important element in the PERMEDIA chip's favor is its price. Boards using this chip will sell for less than \$250. Creative Labs and Televideo are planning to release the first PERMEDIA-based boards in March 1996.

The GLINT chip is more powerful than the PERMEDIA chip. However, most users will find PERMEDIA-based graphics accelerator boards are more than adequate for improving their VRML browser's performance. GLINT-based boards are designed more for users who work with complex computer graphics programs like 3D Studio.

S-MOS SPC1500 Graphics Geometry Processor

The SPC1500 developed by S-MOS Systems was the first commercially available single chip 3D geometry accelerator for PCs.

From Here...

Now that you have bought your ideal VRML machine and set up your Internet connection, you can begin to explore the wide variety of VRML environments. To learn more about the topics discussed here, including browsers, authoring tools, and how to start programming your own VRML environments, see the following chapters and parts:

- ï [Chapter 1, "The Internet: Its History and How It Works,"](#) provides an overview of the Internet and the World Wide Web. It includes a discussion on modems and digital connections.
- ï [Chapter 9, "Overview of VRML Browsers,"](#) describes how VRML browsers work with the World Wide Web and the different types that are available.
- ï [Chapter 32, "Programming Your Own Viewer and Drivers,"](#) describes how you can develop your own VRML utilities.
- ï Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities," describes the different software packages that are available to help you create your own VRML environments.
- ï If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) These chapters provide a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



9-VRML Browsers

The key to using VRML is the VRML browser. In the same way that HTML are viewed using an HTML browser, VRML environments need a VRML browser to turn their collection of coordinates and matrices into a 3D environment that you can wander through. Without this piece of software, your VRML environment is nothing more than a list of numbers and letters.

However, which VRML browser should you use? The answer depends partly on your computer. But it is also a function of the ability of the browser, its interface, and the speed at which it can process the data and render the image. And once you have decided on which browser you will use, how do you use it to connect to the World Wide Web?

This chapter provides an overview of the Web browsers that are available for different computer platforms. It examines the following issues:

- How does a VRML browser work?
- How do I configure my Web browser to use a VRML browser?
- Which browser will work on my system?
- How do I use different browsers?

How Do VRML Browsers Work?

If VRML is a language, then a VRML browser is its interpreter. Indeed, that is the browser's main function: to read a file containing VRML code and attempt to translate it into a graphic image. And whenever it comes across something that it cannot understand, it will give you an error message. In other words, a VRML browser is essentially unilingual—it understands only VRML.

While it is possible to enter the URL of a VRML environment directly into the browser, you are more likely to use a Web browser as your starting point. When your Web browser encounters a file with the .wrl extension, it looks in its configuration file to see if there is an application that is configured for that file type. If you have configured your Web browser correctly, your Web browser will start up your VRML browser.

When your VRML browser is loaded and has downloaded the .wrl file, it will begin to interpret the various nodes and fields. It chooses an initial camera position. It also checks for any links using the WWWAnchor and the WWWInline nodes. If it finds a WWWInline node, the browser will check to see if it needs the child nodes (see ["Puttin Up Walls," Ch. 12](#)). If it does, it will call up those files from the appropriate Web server.

Once it has done all the calculations, the VRML browser renders what it has interpreted on to a computer display. However, many VRML environments are distributed: that is, the information that makes up the environment is spread out over the Web. Some VRML environments may refer to other peoples' worlds. Some may make use of VRML objects that other people have posted on the Web.

In that case, the browser must load the necessary information in stages. In the first stage, the VRML browser loads the basic scene description. It then loads any scenes contained within a scene (or nested scenes). This can result in some delays in loading a VRML world depending on the amount of Internet traffic, the quality of your connection, the power of your computer, and the speed of your connection (see ["Modems, Direct Links, and the Internet," Ch. 1](#)).

As you move through the VRML environment, the browser will be constantly checking to see if it needs to load any new material as your point of view changes. It will also check to see if you activate any WWWAnchor nodes. If you do, the VRML browser will load the corresponding URL and send a request to the appropriate Web server. If the server returns another .wrl file, the browser will start the rendering process again. However, if the server returns a file that the browser cannot interpret, the Web browser on your computer is loaded and it determines what action needs to be taken.

All this rendering requires a fair bit of computer power, however, in terms of both the processor and the amount of memory. A processor that can perform more calculations per second will give a smoother frame rate and, therefore, a more realistic sense of movement in a virtual world. More memory will also help in creating better images in real-time. Moreover, graphics processors like the SGI's RealityEngine2 and 3DLab's GLINT and PERMEDIA chips can remove some of the burden from your computer's processor. For more about the computer hardware considerations for VRML, (see ["What Do You Need to Use VRML?", Ch. 8](#)).

Manipulating VRML

All VRML browsers use the center point of a model as the pivot point of their examination mode. This means that if you want to examine an item at one end of a model, you will not be rotating the object, but the entire environment. Another way of visualizing this kind of manipulation is to imagine that the entire VRML model is contained within a sphere. When you move an object, the entire model moves using the sphere's center as the pivot point (see Fig. 9.1).

How you manipulate objects using a browser will be influenced by their position in the environment. If an object is in the middle of an environment, you will be able to examine the object using the browser's examine mode. However, if the item is off to one side, you will have to use the walk and/or fly modes to reposition it with each change you make in examine mode.

[Fig. 9.1](#)

Rotating around an object.

How Do I Get My Web Browser to Use My VRML Browser?

In order to let your Web browser know when to call up your VRML browser, you need to adjust the former's MIME settings. You begin by calling up your browser's MIME settings dialog box. It may be labelled Helper or Helper Applications. You should get a menu that looks like Figure 9.2.

[Fig. 9.2](#)

The Netscape Web browser's Helper Applications menu.

Look through the menu of file types and look for a line that says "x-world/x-vrml." If you do not find this item, you will need to add a new MIME type. In Netscape, you click on the New Item button. You will get the window for configuring a new MIME type (see Fig. 9.3). In the field for MIME Type, enter x-world. In the field for MIME SubType, enter x-vrml. In Mosaic, you should type VRML World in the Description field and enter x-world/x-vrml in the MIME type field.

When you have filled in the boxes, click on the OK button. The new MIME type will appear in the MIME type window. You then must let your browser know the file extension for VRML. In the File Extensions or Suffixes field, enter .wrl. It should appear in the Extension section of the window.

Once you have configured the MIME type for VRML files, you can then configure your Web browser to run your VRML browser. In the Helper Application window, you will see a section called Action. This is where you tell your Web browser which program to run. In the file type window, select the MIME type for VRML. It should be highlighted when you click on it. In the Action menu, click on the Launch Application radio button. Then in the Launch Application field, type in the drive, directory path, and the executable filename for your VRML browser. For example, if you are using WebSpace 1.0 for Windows NT, you will probably need to type in C:\tgs\WEBSpace.EXE (see Fig. 9.4).

[Fig. 9.3](#)

Configuring the VRML MIME type.

[Fig. 9.4](#)

Setting your VRML browser.

You are now ready to access VRML environments on the Web.

An Overview of VRML Browsers

With the sudden interest in VRML, the number of browsers that are coming on to the market is growing by leaps and bounds. Deciding which browser is best for you can be a daunting task. This section provides an overview of the VRML browsers available at the time of this writing. However, this section

is not meant to be a comprehensive manual for the browsers covered. It includes information about the computer platforms they run on and provides URLs for where you can get more information about them. We have also included details about how to operate some of the browsers to give you an idea of the different interfaces that are being used.

The VRML 1.0 specification was finalized on 26 May 1995. Nearly all of the VRML browsers that we tested were beta or pre-beta versions. A number of them did not have many of the features like LOD and AsciiText nodes implemented. Moreover, their design would often change from week to week. Compounding this problem was the development of the Netscape 2.0 beta version Web browser which included a plug-in feature. Companies like Paper Software and TGS were programming their VRML browsers to use this feature and this resulted in delays in implementing changes to earlier versions. This following overview is as current as possible.

For more information, see <http://www.divelabs.com/vrml.htm>.

VR Scout (Chaco Communications, Inc.)

Supported Platforms: Windows 3.1, Windows NT, and Windows 95.

VR Scout is an external VRML viewer developed by Chaco Communications, Inc. for use with NetManage WebSurfer, Netscape Navigator, Mosaic, and other browsers. It can also be used to view VRML files without a Web browser running. Chaco has also released a plug-in version for Netscape Navigator 2.0. VR Scout supports the entire VRML 1.0 specification, including GIF, JPEG, and BMP textures; the WWWAnchor node; inline objects; point-sets; transformed textures; the LOD node; as well as the AsciiText and Font nodes. VR Scout also uses Intel's 3DR technology, which allows it to use the new 3D-rendering boards to enhance its performance. Figure 9.5 shows a screenshot of the VR Scout browser.

Fig. 9.5

The VR Scout browser.

Troubleshooting

Do your graphics appear dark?

If you find that the graphics on your screen are darker than you think they should be, you can adjust 3DR's "gamma correction" factor. Using Notepad, open the "3DR.INI" file, which should be in your Windows directory. Find the following line

Gamma=1.0

and change it to a higher value. A value between 1.0 and 2.0 is usually correct for PC video hardware.

Navigating a VRML Environment with VR Scout

In VR Scout, you navigate a VRML environment using a click-and-drag interface with your mouse. There are two movement modes: Walk and Fly. You select a mode using the View menu. By selecting

View Examine, you can rotate the entire scene around its center. If you select View Walk, you can move around a scene on a horizontal plane. Choosing View Fly allows you to choose your viewpoint and "fly" through a scene while upside down or tilted. To activate linked objects in the scene, you simply have to click on them once using the left mouse button.

To move using either the Walk or Fly mode, click on and hold the left mouse button and move the cursor around the screen. In Walk mode, move the cursor anywhere on the scene viewing screen. Click and hold the left mouse button. Moving the cursor up from your original point will move you forward; moving the cursor below the center of the screen will move you backwards. Moving the cursor to the left or right of the origin, respectively, will move you left or right in the scene. You can also drag diagonally to combine movements. The further you move away from the original point, the faster you will move through a scene.

You can look up and down by pressing both the Control key and the left mouse button while moving the cursor up and down the screen. When the Shift key and left mouse button are pressed, you can change the position where you are standing up, down, or from side to side without changing your orientation.

You can also control your movement in Walk mode with the keyboard. The following table explains the controls:

	Pressing Direction Key	Direction Key with Shift Key	Direction Key with Control Key
Right Arrow	Turn right	Shift right	
Left Arrow	Turn left	Shift left	
Up Arrow	Walk forward	Shift up	Look up
Down Arrow	Back up	Shift down	Look down

In Fly mode, you move through a VRML environment as if you are in a helicopter. You move in whatever direction you are looking. If you look up and move forward, you start climbing. You can even hover and fly backwards. You also can tilt your view from side to side.

Using the mouse to control your movements in Fly mode is similar to using it in Walk mode. To start moving, click on the scene and move the mouse in the direction you want to fly. Dragging the mouse flies you around in the VRML environment. When the Control key is pressed, you can use the mouse to look up and down as well as tilt left and right. When the Shift key is pressed, you can shift your position without changing your orientation. Dragging diagonally combines movements, and the further you move away from your original clicking point, the faster you will move. The following table illustrates these movements.

	Pressing Direction Key	Direction Key with Shift Key	Direction Key with Control Key
Right Arrow	Turn right	Shift right	Tilt right
Left Arrow	Turn left	Shift left	Tilt left
Up Arrow	Fly forward	Shift up	Look up
Down Arrow	Fly backwards	Shift down	Look down

Troubleshooting

Is the frame rate too slow when you move?

If you find that the frame rate of VR Scout is too slow when you move, you can use the 3D Graphics preferences dialog box to degrade the quality of the image while moving. From the Options menu, choose Preferences, 3D Graphics. In the "When moving use:" field, enter flat shading. You'll move faster. However, objects will not look as good while you're moving.

Examine Mode

Examine mode allows you to look at and manipulate a VRML object. In other words, it is like holding an object in your hand—turning and twisting it to have a better look at it. You can also move the object closer and further away from your eyes. You select Examine mode by selecting View, Examine.

You can rotate the entire scene around its center. You can get a better idea of this by looking at how we can use Examine mode to manipulate a model of a folded piece of paper. Figure 9.6 shows the initial view.

In Examine mode, you can shift the position of the model by clicking and holding the left mouse button on one part of the model and dragging the cursor across the screen. The model shifts its position while rotating around on the center point (see Fig. 9.7). Center point is the center of the VRML environment. Therefore, as long as you keep pressing the left mouse button, the model will move in the direction of the cursor's movement.

You can zoom in or move around using the arrow keys without affecting the center of rotation.

[Fig. 9.6](#)

The initial view of a VRML model of a folded piece of paper in Examine mode.

[Fig. 9.7](#)

The view of a VRML model of a folded piece of paper after manipulation in Examine mode.

You can also manipulate a model using the keyboard, as the following table shows:

	Pressing Direction Key	Direction Key with Shift Key	Direction Key with Control Key
Right Arrow	Turn right	Shift right	Rotate model right
Left Arrow	Turn Left	Shift Left	Rotate model left
Up Arrow	Move towards object	Shift up	Rotate model up
Down Arrow	Move away from object	Shift down	Rotate model down

Headlights

VR Scout uses a headlight feature to illuminate scenes. The default setting is headlights on. Headlights

work the way they sound: they cast light forward from your point of view. You can turn this feature off by selecting Options Headlight. If you turn it off, the scene will be illuminated with only the light that is defined in the model (see Fig. 9.8).

Fig. 9.8

The view of a VRML model with the Headlight feature turned off.

You can also adjust the intensity of the headlight. Choose Options Preferences 3D Graphics. You will get the 3D Graphics page in the Preferences dialog box (see Fig. 9.9). You will see the Headlight Brightness slide bar. Adjust the brightness to the desired level and click on OK. The light intensity of the scene should change.

Fig. 9.9

The 3D Graphics dialog box.

Troubleshooting

How does VR Scout display large textures?

Textures larger than 128 by 128 pixels cannot be displayed correctly using VR Scout because of a limitation in 3DR. VR Scout solves this problem by using texture size reduction. The 3D Graphics page in the Preferences dialog box allows you to turn the feature on and off.

The VR Scout Plug-In

Chaco has released a plug-in version of VR Scout for Netscape Navigator 2.0 (see Fig. 9.10). When a VRML file is called up, the VR Scout plug-in screen will appear in the Netscape viewing area.

You navigate and examine VRML environments in the VR Scout plug-in in the same way as in the stand-alone version. However, the plug-in version adds a toolbar at the bottom of the screen that lets you choose the mode you want to use by clicking on its name. You can also call up the help menu by clicking on the "?".

To call up the menu, click on the right mouse button. The VR Scout menu will appear at the point where you clicked.

For more information, see <http://www.chaco.com/vrscout/>.

Fig. 9.10

The VR Scout Plug-In for Netscape Navigator 2.0.

WebFX Plug-In (Paper Software, Inc.)

Supported Platforms: Windows 3.1, Windows NT (Intel), and Windows 95. Forthcoming: Macintosh versions.

WebFX is a VRML browser developed by Paper Software, Inc. There are plans for two models: a plug-in

version that is incorporated into a Web browser and a stand-alone version called Explorer. At the time of this writing, the WebFX plug-in for Netscape Navigator was the only version of the software to be released. It provides full VRML 1.0 compliance, physics-based navigation, collision detection, viewpoints, gzip support, multiple nested inlines, and support for common Open Inventor nodes. Future versions will support IRC-based multiuser chatting, in-place object-oriented VRML authoring, VRML 1.0 strict checking, and sound and movie extensions.

The plug-in versions are linked to a Web browser like QMosaic and the Netscape Navigator. The WebFX Plug-In lets you view 3D VRML files inside an HTML document in much the same way you view GIF images (see Fig. 9.11). You can view multiple VRML files displayed at once within the same HTML document.

WebFX Explorer is a stand-alone VRML and Web browser that takes full advantage of the Windows 95 interface, providing object-oriented drag and drop, shortcut, and customization capabilities within its interface (see Fig. 9.12). It provides full stand-alone HTTP communications and runs independently of any Web browser.

Fig. 9.11

The WebFX Plug-in VRML browser.

Fig. 9.12

The WebFX Explorer. Figure used with per-mission of Paper Software, Inc.

Using the WebFX Plug-In for Netscape

The WebFX Plug-In for Netscape Navigator version 2.0 was the fastest of the VRML browsers we tested. It also had some of the best graphics. The lighting effects, shading, and texture mapping were among the most realistic. Moreover, by embedding its VRML viewing technology into Netscape, WebFX is able to make use of Netscape's built in multitasking, caching, security, document source views, and bookmark management capabilities. You can even create Webpages that combine an HTML environment with a VRML environment (see Fig. 9.13).

Fig. 9.13

A Webpage combining an HTML document and a VRML environment.

In order to use the WebFX Plug-in for Netscape, you simply need to load a VRML file. Netscape will automatically activate the program whenever it comes across a file with the .wrl extension (see Fig. 9.14).

Fig. 9.14

The WebFX Plug-In for Netscape Navigator 2.0. Figure used with the permission of Paper Software, Inc.

There are three navigation modes in WebFX: walk, fly, and point. The Walk mode lets you navigate an environment along the same plane. The Fly mode lets you move through an environment using all six degrees of freedom. In Point mode, you can left-click on an object and the program "moves you" towards it until you click the left mouse button again. To switch between modes, use the toolbar or right-click on

the viewing screen to bring up the WebFX shortcut menu (see Fig. 9.15).

Fig. 9.15

The WebFX Shortcut menu.

Then choose Navigation. From here, you can select the mode you wish to use. You can also click on the appropriate mode using WebFX's heads-up display (HUD). You can use either the keyboard or mouse to navigate in each mode. The following table lists different mouse and keyboard commands for each mode:

Walking

Up/Down Arrows	Move forward/backward
Left/Right Arrows	Turn left/right
A/Z	Tilt head up/down
J	Jump
Mouse Left Drag	Move forward/backward or turn left/right
Mouse Right Drag	Orbit about the entire scene
Mouse Right Click	Pop-up the WebFX shortcut menu
Mouse Ctrl+Left Click	Automatically walk to the object you clicked on
Shift key (with mouse or keys)	Move faster
Alt key (with mouse or keys)	Pan left/right or up/down
Space	Open doors, manipulate objects

Flying

Up/Down Arrows	Tilt up/down
Left/Right Arrows	Turn left/right
A/Z	Thrust forward/backward
Q/E	Revolve counterclockwise/clockwise
Mouse Left Drag	Tilt up/down or turn left/right
Mouse Right Drag	Orbit about the entire scene
Mouse Right Click	Pop-up the WebFX shortcut menu
Mouse Ctrl+Left Click	Automatically walk to the object you clicked on
Shift key (with mouse or keys)	Move faster
Alt key (with mouse or keys)	Pan left/right or up/down
Space	Open doors, manipulate objects

Pointing

Keyboard	Same as walking
Mouse Left Click	Animate you closer to the point you click on
Mouse Right Click	Pop-up the WebFX shortcut menu

Mouse Left Drag	Move the scene or object
Mouse Right Drag	Rotate the scene or object

For example, if you wanted to move to the left in Walk mode, you would click and hold the left mouse button while moving the cursor to the left side of the screen (see Fig. 9.16).

Fig. 9.16

Moving left in Walk mode.

Troubleshooting

Need to review mouse and keyboard commands?

If you find that you have forgotten how to move or change the direction of your field of view, simply click on the "?" in the toolbar on the bottom of the screen. A list of the mouse and keyboard commands for your current mode will appear on the screen. You can also select the Heads Up Display option from the shortcut menu and select Navigation Help.

If the VRML environment has more than one camera position programmed into it, you can change your position by using the viewpoint feature in the main menu. Click the right mouse button to call up the shortcut menu and choose ViewPoints. You will get a list of the different viewpoints that are available in that environment (see Fig. 9.17).

Fig. 9.17

Choosing a viewpoint in WebFX.

Let's say that we want to go to the Foyer. We simply point the cursor to Foyer and click the left mouse button. WebFX will then immediately move us to the new camera position (see Fig. 9.18).

Fig. 9.18

The view after choosing a new viewpoint.

Optimizing WebFX's Performance

If you are running WebFX on a slow machine or are loading a particularly large environment, you may find that your movement in the virtual space is noticeably slow. You can adjust the settings in WebFX to make things run faster.

One way of speeding up performance is to optimize the window size. The larger you make the WebFX window, the slower will be its performance. Click once on the right mouse to call up the shortcut menu and choose Options, Optimize Window Size. This option will maintain the WebFX window at a reasonable height and width for optimal performance.

Another way of optimizing performance is to change the way the browser handles back-facing polygons. Many models assume that a browser will display both sides of a polygon. While this is the case on high-end graphics workstations, PC-based 3D rendering engines do not draw the back face of a polygon

in order to improve performance. As a result, it is currently necessary for WebFX to assume that every polygon should have a back face even though it may not be visible. This effectively doubles the number of polygons that need to be calculated, thereby reducing performance substantially. As VRML content creators begin to create models which explicitly create a back face only when needed, you can use this option to make WebFX run faster. To turn it on, click the right mouse button to call up the shortcut menu and choose Options, "Always Generate Back Faces" option; you can turn this option off by clicking on it with the left mouse button.

You can also speed up performance by turning off the WebFX heads-up display (HUD). To turn it off, call up the shortcut menu and choose Heads Up Display, All Off. This will not get rid of the toolbar at the bottom of the WebFX screen.

You can also choose to have WebFX optimize for performance. At the time of this writing, this feature will simply switch the program to a faster lighting algorithm. Later versions will automatically set all WebFX options for maximum performance. To turn this option on, call up the shortcut menu and choose Options, Optimize for Performance. The default setting is Optimize for Appearance.

Changing the Rendering Method and Lighting in WebFX

WebFX also allows you to change the rendering method and lighting used to create the scene on your computer. By changing the rendering method, you can speed up the browser. Changing the lighting allows you to switch between using a headlight to illuminate the scene in front of you or using the ambient lighting programmed into the environment.

The default setting in WebFX is to have the headlight setting switched on. Figure 9.11 shows a scene with the headlight turned on. To turn it off, click once on the right mouse button to call up the main menu and choose Lights, Headlight. A checkmark will appear beside the Headlight option when it is on, and will not appear when it is off. Figure 9.19 shows the scene in Figure 9.14 without the headlight feature.

[Fig. 9.19](#)

A scene rendered with the headlight turned off.

You can also change the rendering method used in WebFX. The default setting is smooth shading. Figure 9.19 shows a scene using this method. You can change the method to flat shading by calling up the shortcut menu and choosing Lights, Flat Shading. In Figure 9.20, we see the same scene is rendered using this method.

[Fig. 9.20](#)

A scene rendered with Flat Shading.

You can speed up the rendering process even further by using the wireframe option. Call up the shortcut menu and choose Lights, Wireframe. Figure 9.21 shows how the scene in Figure 9.11 looks when rendered using wireframe.

WebFX also allows you to view a scene using point clouds. In this rendering method, only the points that make up an object are displayed. To choose this rendering method, call up the shortcut menu and choose Lights, Point Cloud. Figure 9.22 shows how the scene in Figure 9.11 looks when rendered using only

point clouds.

To find more information about WebFX or to get the latest version, see <http://www.paperinc.com/>.

Fig. 9.21

A scene rendered with Wireframe.

Fig. 9.22

A scene rendered as Point Clouds.

WebSpace (Silicon Graphics Inc. and Template Graphics Software, Inc.)

Supported Platforms: SGI, Windows NT, Windows 95, Sun Solaris ZX/TZX, and IBM AIX.
Forthcoming: Windows 3.1, Macintosh, Digital UNIX, and HP/UX.

Given that Silicon Graphics, Inc. (SGI) helped to develop VRML, it is only natural that SGI would have its own VRML browser. But SGI's WebSpace is available on more than the SGI platform thanks to an agreement between it and Template Graphics Software, Inc. (TGS). While SGI handles the development of WebSpace for its own computer platform, TGS is developing versions for other computer platforms. This has resulted in a slight lag in WebSpace versions for non-SGI computers. Currently, TGS offers WebSpace version 1.0 for the Windows NT, Windows 95, Sun Solaris ZX/TZX, and IBM AIX platforms (see Fig. 9.23). However, SGI has released WebSpace 1.1 for SGI computers (see Fig. 9.24).

Fig. 9.23

A screenshot of TGS's WebSpace 1.0 for Windows NT.

TGS has also announced that it has developed a WebSpace plug-in version for Netscape Navigator version 2.0.

Moving Around Using WebSpace

Whereas many VRML browsers like WebFX and VR Scout simply use the mouse and/or the keyboard arrow keys to control movement, WebSpace provides a clear navigation interface: the Walker Viewer. Click on the View menu item and choose Walk Viewer. Figure 9.25 shows the Walk Viewer with its navigation console in WebSpace 1.0 for Windows NT.

The console consists of the seek tool, the steering/tilt handle, and the arrow pad. The steering/tilt handle is the tool you will probably use the most to navigate a VRML environment. You can think of it as two navigation mechanisms joined together. On the left side is the steering mechanism. To turn left, you simply place the cursor on the handle, click and hold the left mouse button, and move the cursor left (see Fig. 9.26). To turn right, you push the cursor right.

Fig. 9.24

A screenshot of SGI's WebSpace 1.1.

Fig. 9.25

The navigation console for WebSpace 1.0 for Windows 95 and Windows NT (Intel).

To go forward, place the cursor on the handle, click and hold the left mouse button, and move the cursor forward. And to move back, do the same but move the cursor back.

You will notice a red line on the handle. When you place the cursor on this line, you can use the handle to tilt your point of view up and down. To look up, place the cursor on the red line, click and hold the left mouse button, and push the cursor forward (see Fig. 9.27). To look down, do the same except pull the cursor back.

You can also shift your point of view by using the Arrow pad. You can shift your position up, down, left, or right by clicking and holding on the pad, and moving the mouse in the direction you wish to move. For example, to raise your point of view, click and hold the Arrow pad, and then move your mouse forward. This will raise your point of view (see Fig. 9.28).

Fig. 9.26

Turning left with the steering/tilt handle.

Fig. 9.27

Looking up with the steering/tilt handle.

If you are navigating a large environment, you may find it tiring to hold a mouse button down for any length of time. WebSpace gives you a break by letting you use the Seek tool to move you half the distance towards an object. To use it, click on the tool. The cursor will change to a crosshair. Move the crosshair to the object to which you want to move and click the mouse button again. You will move half the distance between your position and the object (see Fig. 9.29).

Fig. 9.28

Shifting position with the arrow pad.

You can repeat the process until you reach the desired distance.

You can also move to different camera positions that have been defined in the VRML environment. Click on the Viewpoints menu item and a list of the available camera positions will pop up (see Fig. 9.30).

Fig. 9.29

Using the Seek tool.

Fig. 9.30

Using the Viewpoint menu.

Click on your choice and your position will be automatically shifted to the new location. For example, if you want to move to the Foyer in the environment shown in Figure 9.24, click on Viewpoints, Foyer. Your position will be automatically shifted to the Foyer (see Fig. 9.31).

[Fig. 9.31](#)

Moving to the Foyer camera position.

The WebSpace Examine Viewer

WebSpace also lets you examine objects using the Examine Viewer. Click on the View menu and choose Examine Viewer. You will get the Examine Viewer screen (see Fig. 9.32).

[Fig. 9.32](#)

The WebSpace Examine Viewer screen.

You can rotate the model using the on-screen trackball. Move the cursor to the track ball, and then click and hold the left mouse button. As you adjust the trackball, the model will rotate in the same way. For example, let's say you want to examine the model of a folded piece of paper in Figure 9.32. You want to shift the model to the right. To do this using WebSpace's Examine Viewer, click and hold on the trackball and move the cursor to the right. You will see the trackball and the model rotating in the same direction (see Fig. 9.33).

You can also use the Seek Tool to shift the model to bring a particular point to the front of the scene. Let's say that we want to shift our focus on to the top right corner of the paper model. Click on the Seek Tool and the cursor changes to a crosshair. Move the cursor to the top right corner of the model (see Fig. 9.34).

Now click on the model and it will shift position (see Fig. 9.35).

[Fig. 9.33](#)

The model after rotating the trackball to the right.

[Fig. 9.34](#)

Using the Seek Tool cursor to select a new focal point.

The Flywheel moves the model either closer or further away without shifting your point of view. To move the model closer, click and hold on the Flywheel and move the cursor backwards. To move the model away from you, do the same except move the cursor forward. The Arrow Tool operates in the same way as in the Walking Viewer.

[Fig. 9.35](#)

The new view after clicking on the point.

Changing the Headlight Setting in WebSpace

Like other VRML browsers, WebSpace uses a headlight feature to illuminate the objects in the field of view. This is the default setting. Figure 9.36 shows a nighttime street scene when the headlight feature is turned on.

[Fig. 9.36](#)

A nighttime street scene viewed with Headlights turned on.

However, we can turn this feature off to see how the scene is actually illuminated. Select the View menu and click on the Headlights option. We can now see the scene as it was actually designed (see Fig. 9.37).

Fig. 9.37

The same scene viewed with Headlights turned off.

Selecting an URL Programmed Using WWWAnchor

Whenever you come across a WWWAnchor node, WebSpace alerts you by changing the shape of the cursor (see Fig. 9.38).

Fig. 9.38

WebSpace's URL cursor.

When you see this cursor, you will know that there is a WWWAnchor node attached to an object. To activate the node, you simply click once on the object with the left mouse button. For example, there is a WWWAnchor node attached to a bookcase in a VRML environment. The cursor changes shape to alert you that the bookcase has a link (see Fig. 9.39).

If the link is to another VRML object or environment, WebSpace will load the .wrl file and render it. However, if the link is to a Web document, WebSpace will call up your Web browser to display the Web Page (see Fig. 9.40).

Fig. 9.39

Finding a Web link attached to a VRML object.

Fig. 9.40

A Webpage displayed from a WWWAnchor link.

To find more information about WebSpace for the SGI platform, see <http://webpace.sgi.com/WebSpace/WhatsNew/index.html>. To find more information about WebSpace for other platforms, see <http://www.cts.com/~template/WebSpace/monday.html>.

WorldView (InterVista Software, Inc.)

Supported Platforms: Windows 3.1, Windows NT (Intel), and Windows 95. Forthcoming: Macintosh 68000 and Macintosh PowerPC.

InterVista Software, Inc.'s WorldView is a fast, easy-to-use VRML browser designed for the personal computer market. Its makers claim that it can render real-time graphics on a standard PC and 14.4 Kbps modem. It uses Microsoft's RealityLab rendering engine for its Windows versions and can be installed over a network. Figure 9.41 shows the WorldView for Windows screen.

Fig. 9.41

The WorldView for Windows VRML browser.

WorldView can act as a stand-alone browser for exploring VRML environments. It lets you enter an URL, save your own viewpoints, and go back to environments you have previously visited in that session. However, WorldView can also be used as a helper application for a Web browser like Netscape Navigator.

Loading via an URL in WorldView

To load via an URL in WorldView, select the File menu and choose Open Location. You will then get the Open Location dialog box (see Fig. 9.42). Enter the URL in the Open Location field. For example, we can enter the URL for the model of a bird feeder at the University of Arizona's VRML site (<http://amber.rc.arizona.edu/vrml/birdfeeder.wrl>).

This will download the VRML file from the server and display it in WorldView (see Fig. 9.43).

Fig. 9.42

The Open Location dialog box.

Fig. 9.43

The Bird feeder Model (<http://amber.rc.arizona.edu/vrml/birdfeeder.wrl>).

Navigating and Manipulating VRML Models in WorldView

WorldView uses an interface that is one of the most intuitive and non-intrusive available. Like WebSpace, it provides a tool to navigate and manipulate a VRML environment. However, unlike WebSpace, WorldView's control panel is placed outside of the viewing screen.

One of the great features of the WorldView control panel is that all the movement option controls are available on the control panel. You can change from World View mode to Model View mode by simply selecting the appropriate control button. Moreover, all the movement controls are on the panel as well. Thus, you never need to call up a menu when you want to change modes.

There are four movement controls in WorldView: Fly, Translate, Tilt, and the Move crosshair panel. The Move panel works the same way in both the World View and Model View modes. It lets you move on a horizontal plane around an environment. To use it, move the cursor over the panel and click and hold the left mouse button. Moving the cursor to different positions in relation to the crosshair has the following effects on the direction of your movement:

- Above the crosshair means move forward; below means move back.
- Left of the crosshair means rotate left; right of the crosshair means rotate right.
- Using the Shift key with the mouse pans the scene.
- Using the Control key with the mouse does pitch and roll.

You can also use the viewing window in much the same way as the Move panel. Simply imagine that there is a crosshair in the middle of the window. With the cursor in the view screen, use the Shift key and the left mouse button. You can move forward, backwards, left, and right in same way as using the Move

panel.

The Fly, Translate, and Tilt controls vary according to whether you are in World or Model View mode. Table 9.1 shows the actions for each control in both modes:

Table 9.1 WorldView Fly, Translate, and Tilt Controls

	World View	Model View
Fly		
Up Arrow	Will move you forward through an environment	Will move the model away from you
Down Arrow	Will move you out or away from an environment	Will move the model closer to you
Right Arrow	Will shift your camera view to the right	Will turn the model to the right
Left Arrow	Will move your camera view to the left	Will turn the model to the left
Translate		
Up Arrow	Moves you up or gives the appearance of moving the world down	Moves the model up
Down Arrow	Moves you down or gives the appearance of moving you up	Moves the model down
Right Arrow	Will move your camera view to the right	Moves the model to the right
Left Arrow	Will move your camera view to the left	Moves the model to the left
Tilt		
Up Arrow	Tilts the camera angle up	Tilts the top of the model towards you
Down Arrow	Tilts the camera angle down	Tilts the top of the model away from you
Right Arrow	Tilts the camera angle right	Tilts the top of the model to the right
Left Arrow	Tilts the camera angle left	Tilts the top of the model to the left

To use these controls, simply position the cursor over the arrow and click on the left mouse button. For example, we can go closer to the globe in Figure 9.43 by using the Fly up arrow in World View mode (see Fig. 9.44).

For more information on WorldView, see <http://www.webmaster.com/vrml>.

Fig. 9.44

Using the Fly control in World mode.

AmberGL VRML Browser 1.0 (Dive Laboratories Inc.)

Supported Platform: Windows NT (Intel).

The Amber VRML browser is built with DIVE Lab's AmberGL virtual environment toolkit for OpenGL. Figure 9.45 shows a screenshot of this browser. It is currently available only in an alpha version, and it does not support a number of features including the texture nodes, the Cylinder node, the Sphere node, and point sets. Future versions will include improved navigation as well as support for HMDs and datagloves.

Fig. 9.45

The Amber VRML browser.

For more information, see <http://www.divelabs.com/vrml.htm>.

Fountain (Caligari Corporation)

Supported Platforms: Windows 3.1 and Windows 95.

Although Fountain is better known for its ability as a VRML 3D modeling tool (see "[An Overview of Worldbuilding Toolkits](#)," Ch. 25), Caligari also advertises it as a stand-alone VRML browser. Its ability to link to the Internet is especially useful when testing levels of detail, in-lining, and hyperlinks to HTML documents. Microsoft has licensed Fountain's technology for its set of Internet development tools.

The minimum requirements for using Fountain are a fast 486 with 8MB of RAM. Caligari also recommends that if you want to view VRML environments with more than a couple of thousand polygons, you should have a Pentium with a 3D accelerator card.

The Fountain VRML browser supports textures, inlining, and the LOD node. It lets you navigate using Fly mode. Its navigation controls are similar to action games like Doom. As you become more comfortable with the controls, you can select the Auto-Pilot mode. This option lets you control the direction of your movement as in Fly mode, but you no longer have to move the mouse to go forwards or backwards. You can also examine objects using the Look At tool.

For more information, see <http://www.caligari.com/ws/fount.html>.

GLView (Holger Grahm)

Supported Platforms: Windows NT 3.51 and Windows 95.

GLView WWW Edition is a 3D file viewer for Windows NT 3.51 and Windows 95 developed by Holger Grahm. It includes support for VRML 1.0 files, native Windows OpenGL rendering, and integrated WWW Access. The program is capable of texture mapping support, allows animation extensions, and supports the morphing of an object into a sphere/cylinder/box. You can import GLView 3DV, Wavefront OBJ, DXF, RAW, and GEO objects. You can also export GLView 3DV, RAW, GEO, STL, VRML (single object). For more information, see <http://www.snafu.de/~hg>

NAVFlyer (MicronGreen, Inc.)

Supported Platforms: Windows 3.1, Windows NT, and Windows 95.

NAVFlyer is an interactive flying tool for navigating 3D models from VRML, DXF, and 3DS files in various forms including wireframe, shaded, and texture mapped display modes (see Fig. 9.46). It requires a 256-color Windows display as well as WinG, and Win32s for Windows 3.1. LOD support is also included for handling large and more complex models, or to use NAVFlyer on slower machines.

Fig. 9.46

The NAVFlyer browser.

The mouse can be used to navigate through a virtual space or to manipulate an object using 6 degrees of freedom. You have the option of loading the model as a single object, including 2D data, set circle segment size, and setting collision mode on/off. Speeds are variable for translation and rotation while flying. For more information, see <http://www.microngreen.com>

Virtual Explorer (Microsoft Corporation)

Supported Platforms: Windows 95.

Microsoft's Virtual Explorer is a fully integrated VRML plug-in for Microsoft Internet Explorer. Based on the RealityLab 3D rendering engine and the WorldView browser, Virtual Explorer offers support for the VRML 1.0 specification as well as some extensions. Its minimum requirements are a 486/33 Mhz computer with 8MB of memory and a fast video card.

When you open a VRML file, Virtual Explorer displays the environment and a toolbar (see Fig. 9.47).

Fig. 9.47

The Virtual Explorer.

The toolbar provides a menu button, slide, drive and tilt controls, an examine button, a reset button, and a straighten button. The menu button displays the Virtual Explorer menu; you can also call up the menu by clicking on the right mouse button. The slide control lets you shift your viewpoint up, down, left, or right. The drive control moves you through a VRML scene. The tilt control tilts your viewpoint up, down, left, or right. The examine button lets you inspect objects by spinning them around. The reset button returns you to your starting locations and orientation. The straighten button will reset your viewpoint with respect to the scene.

You can navigate through a scene using your mouse, keyboard, or joystick. To move using the mouse, point to the scene or to one of the navigation controls on the toolbar. Press and hold the left mouse button and move the mouse in the direction that you wish to move.

To move forward using the keyboard, press the up arrow key. You can move backward by pressing the down arrow key, turn right by pressing the right arrow key, and turn left by pressing the left arrow key. To tilt your view, hold the Shift key and press the up, down, left, or right arrow keys. To shift your point of view, hold the Control key and press the up, down, left, or right arrow keys.

Virtual Explorer has a collision detection feature that can be turned on or off. Call up the Virtual Explorer menu to see if Prevent Collisions is checked. If it is not, click on the option. This will turn on the collision detection feature.

For more information, see <http://www.microsoft.com/windows/ie/vrml.html>.

Virtus Voyager (Virtus Corporation)

Supported Platforms: Macintosh and Power Macintosh. Forthcoming: Windows 95.

Virtus Voyager is a VRML browser for the Macintosh and the Power Macintosh platforms. A Windows 95 version is planned for release in early 1996. It must be configured as a helper application for HTML browsers such as Mosaic and Netscape. In addition to VRML, Virtus Voyager will be able to import Virtus' own VMDL file format and other 3D file formats.

For more information, see <http://www.virtus.com/voyager.html>.

VRealm (Integrated Data Systems and Portable Graphics, Inc.)

Supported Platforms: Windows 95 and Windows NT. Forthcoming: OS/2 Warp, IBM AIX, Digital Unix, HP-UX, and Power Macintosh.

The VRealm Browser is a fully VRML-compliant browser being codeveloped by Integrated Data Systems (IDS) and Portable Graphics, Inc. It can function as a stand-alone application or as a Helper application for Netscape Navigator. VRealm incorporates advanced image, video, audio, animation, and VR techniques as well as basic Internet functions, such as HTML. Thus, you use this browser to view standard HTML files as well VRML environments.

VRealm will parse Open Inventor, VRML, and HTML files. It uses a highly intuitive interface and includes collision detection and gravity options. You can select different viewing modes and can mark a position in an environment so that you can return to it at any time. VRealm also includes behavior engines that allows you to manipulate 3D models within a scene.

For more information, see <http://www.ids-net.com/vrealm.html>.

VRML Equinox (North Plains Systems Inc.)

Supported Platforms: Power Macintosh. Forthcoming: Macintosh.

VRML Equinox is a VRML browser for the Power Macintosh developed by Phil Smy that supports the full VRML 1.0 specification (see Fig. 9.48). It requires that you install QuickDraw 3D and have System 7.5 or greater.

Fig. 9.48

The VRML Equinox browser. Used with permission of North Plains Systems Inc.

VRML Equinox can be used as a stand-alone browser or as a helper application for Netscape Navigator. The full version is scheduled for release by April 1996.

For more information, see <http://www.ipsystems.com/nps/>.

VRWeb (IICM, NCSA, and the Gopher Team)

Supported Platforms: Windows 3.1, Windows NT, Windows 95, HP-UX, SUN OS, SUN Solaris, SGI IRIX, DEC Alpha, DEC ULTRIX, LINUX, and IBM AIX. Forthcoming: Macintosh Power PC and Macintosh 68000.

VRweb is a 3D graphics viewer jointly developed by IICM (home of Hyper-G), NCSA (home of Mosaic), and the University of Minnesota (home of Gopher). Figure 9.49 shows a screenshot of the VRWeb browser. It supports two file formats: the Hyper-G Scene Description Format (SDF) and VRML. VRML support includes the 1.0 specification using QvLib 1.0 as the parser. The source code for UNIX/X11 and Windows versions are also available. OpenGL versions of VRWeb are available for the SGI, Dec Alpha, and Windows NT platforms.

Fig. 9.49

The VRWeb browser.

For more information, see <http://www.iicm.tu-graz.ac.at/Cvrweb>.

WebOOGL (The Geometry Center)

Supported Platforms: SGI and Sun. Forthcoming: LINUX and other X-Window system platforms.

The WebOOGL software package is a public domain 3D Web browser. It is a "quasi-compliant" VRML browser. In other words, it recognizes some VRML nodes but not others. For example, the VRML texture mapping nodes are not supported by this browser.

The WebOOGL browser is designed to parse files using the Object-Oriented Graphics Language for the World Wide Web (Web OOGL). This language was developed by the Geometry Center at the University of Minnesota. Web OOGL was one of the 3D graphics languages that was considered as a possible standard before the implementation of VRML.

For more information about WebOOGL, see <http://www.geom.umn.edu/software/weboogl>.

WebView (San Diego Supercomputer Center)

Supported Platforms: SGI and UNIX.

SDSC WebView is a publicly available VRML browser for SGI and UNIX computers. It is released with a complete source code and is meant to act as a public platform for developing and testing additions to the VRML specification. Figure 9.42 shows a screenshot of the WebView browser.

To use this software, you will need the following installed on your computer or host:

- C++ and ANSI C compilers
- OpenGL

- OpenInventor
- Motif version 1.1 or later
- X11 r5 or later.

For more information about WebView, see <http://www.sdsc.edu/EnablingTech/Visualization/vrml/webview.html>.

Whirlwind 3D Browser and Web Surfer (Bill Enright and John Louch)

Supported Platforms: Macintosh PowerPC.

The Whirlwind browser developed by Bill Enright and John Louch was the first VRML browser for the Macintosh platform. It currently only works with the Netscape Web browser.

In order to use Whirlwind, you need to have the QuickDraw 3D extensions to MacOS loaded onto your computer. They are available at <http://www.info.apple.com/qd3d/QD3D.HTML>. To run them, you need at least 16 MB of RAM installed on your computer. This means that most of the Performa bundles that are currently being marketed will not be able to use Whirlwind unless you purchase another 8 MB of RAM.

For more information, see <http://www.info.apple.com/qd3d/Viewer.HTML>.

From Here...

Once you have found the browser that best suits your tastes and computer platform, you are ready to explore and create your own VRML environments and objects. To learn how to start building your own VRML environments, see the following:

- If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) These chapters provide a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.
- If you want to look at some VRML environments and models, see [Chapter 23, "A Guide to Other Sample Environments on the Web."](#) It lists some of the VRML sites that are currently available.
- If you want to see what programs are available to help you create VRML environments and objects, see Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities." These chapters provide an overview of the software that can help make it easier to program a VRML world.
- If you feel the need to create your own VRML browser or to understand more about how they work, see [Chapter 32, "Writing VRML Software."](#) This chapter will give you the basic knowledge that you will need to create a VRML browser and utilities from scratch.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



10-Laying the Foundation Planning Your VRML Environment

So you have browsed a number of VRML environments and have looked over its mechanics, and now you feel you want to make your own environment. The first action you need to make is planning. During the initial stages of VRML's development, people created environments mainly for the sake of seeing if they could do it. The language was so new and there were so few resources available that everyone wanted to see what the language could do.

VRML is still relatively new and its full potential has yet to be realized. However, we have gone beyond the "gee whiz" stage of creating environments. We now have to consider the kind of effect we want our environments to have on the viewer. We also need to carefully plot out the design of our VRML objects and spaces to avoid having troubles later.

This chapter looks at the process anyone should follow before creating his or her VRML environment. The topics that are covered include

- Basic design considerations for a VRML environment
- Mapping out your environment
- The Cartesian Coordinate system
- Converting your maps into coordinates

Designing Your Environment What Do You Want It to Do?

Have you ever gone to a Web site and become annoyed when you couldn't figure out what it was trying to do? This is a far too common occurrence as the Web grows. And it may become a too common occurrence as VRML environments begin to proliferate on the Internet. However, this does not have to be the case.

Many Web sites and some VRML environments suffer from a lack of direction. They are created without

a central vision by their creators and, as a result, are constructed haphazardly. And without some central focus to guide its construction, a VRML environment can become a confusing mess of objects.

Therefore, as you begin planning your VRML environment, you must think about its purpose. Do you want to get a particular message across to other users? Are you trying to sell something? Do you want to say something about yourself? What is the point you want to highlight? Even if you simply want to show that you can create a VRML environment, you need to think about what that entails.

Moreover, having a purpose for your VRML project will influence its design. If you want to create a virtual retail space, you will need to avoid cluttering up your environment with too many objects or place your items in spaces that distract your customers from what you want them to buy. If you want to create a space that will be "an expression of your own individuality," you need to think about what kind of environment will accomplish this goal. If you want to show off your VRML skills, you have to think about what will catch everyone's attention and will distinguish your world from those created by other people trying to do the same thing.

For example, let's say that we want to create a VRML world that acts as a VRML tutorial. When people click on an object, it will link with a Web page explaining how that object was created and list its VRML code. So immediately we know that we will have to write some HTML documents. If we want our world to be a tutorial, we will need to have some kind of guide to direct users through our environment. This will entail defining where to set up the opening view, where we put objects, and how we set up the walls, doors, etcetera in our world. The best way of dealing with this task is to make a list of ideas. The following is a preliminary list of things that need to be done or considered for our tutorial environment:

1. Decide which VRML nodes should be covered.
2. 1a. Primitive Shapes
3. 1b. IndexedFaceSet and CoordIndex
4. 1c. Transform
5. 1d. Materials and Texture
6. 1e. Coordinate3
7. Write HTML documents on each subject.
8. Design objects to be linked with each subject.
9. Design world.
10. 4a. Rooms or open space
11. 4b. Do we need a floor?

We can now begin planning our environment in earnest.

Basic Design Considerations A Question of Asking the Right Questions

Now that we have come up with the idea for our VRML environment, we need to consider a number of factors that will influence our final design. As well as thinking about what to put in, we need to remember that there are factors that may limit what we will do. It is all a matter of asking the right questions. How big an environment should we create? How detailed should our worlds be? How should it be shaped? How should everything be laid out?

Environment Size and Detail

The first thing you should consider before drafting your environment is size—not in terms of the space it takes up in the virtual world, but the size of your .wrl file. In the best of all worlds, everyone would have a high-powered graphics workstation and a T3 digital line. Then we could be relatively unconcerned about how long it will take to download and render our VRML environment. But the reality is quite different. Most people are running PCs with a 14.4 Kbps modem over conventional phone lines. If our environment is 10MB, it will take someone at least 93 minutes just to download it. Then it will take the user's computer some time to do all the calculations needed to actually render it. If you are a retailer trying to sell your goods over the Net, it would be like telling your customers that they had to wait in line for an hour and a half just to get into your store.

Therefore, you need to consider how big you are going to make your VRML environment and how detailed it should be. It is a question of compromise. You can have a very large environment, but you will not be able to add every nook and cranny. For example, you may have a virtual sculpture gallery that is very large, but you may not be able to display the full complexity of some of the sculptures. However, you could have a great deal of detail as long as your world was kept relatively small. It is a question of balance: trading off size for detail and vice versa. (Strategies for improving the performance of your VRML environment are discussed in [Ch. 17](#).)

This is why it is so important to come up with a purpose for your world. If you are trying to sell something and you want your customers to understand what they are getting, you probably want to allow more detail in your environment. However, if you want to give people a sense of what it is like to stand next to the Pyramids, rendering it texture-mapped-brick by texture-mapped-brick is not really necessary. If you want to let people tour a model of your entire chemistry laboratory, you may not be able to include every test tube and beaker. But if you want people to see the tools used in a lab, you can limit your environment to a single lab table.

Design and Layout

But while you need to negotiate a balance between environment size and detail, you do not want to forget about how your VRML world will look. Because VRML is a visual medium, you view a VRML environment; hence, you must think about your environment in visual terms. What is the first impression that you want to give someone entering your environment? How are objects to be laid out? Do you want people to follow a particular path through your environment? Or should they be allowed to wander freely?

Again, the important factor in answering these questions is your environment's purpose. For example, if you are creating a virtual world that requires users to follow a particular sequence, then you need to find ways to direct their flow. On the other hand, if you want to create a place for people to meet in cyberspace, you want to have a more open environment in which people can mingle. Even if you are recreating a space that exists in the real world, you need to consider what is necessary and not necessary. Here are three examples that better illustrate the planning process: a music store, a recreation of an art gallery, and a VRML tutorial.

Planning a Music Store

You want to offer your music CD catalogue over the Net, and you have decided to design it as a virtual music store using VRML. Since your catalogue carries a number of music categories, you want to divide your space into four areas: rock, rap, folk, and classical. You also feel it is important to show the covers of the CDs since the users will not be buying them directly from you, but will have to go to a retail outlet and find the titles they want.

Since most people tend to concentrate their buying habits in one category, it is not necessary to let people have access to all the categories at the same time. Thus we can limit the size of the .wrl files by creating separate environments for each category. You simply link them with a connecting room that has an entrance for all four environments and allow users to choose which category best suits their tastes.

Let's say you want to create a simple store with three rooms of the same size connected to a central space. Using Cube nodes to define each room, you would need the following code to render the environment in Figure 10.1:

Listing 10.1

#VRML V1.0 ascii

Separator {

#The Connecting Space

Separator {

Translation {

translation 0 -1.9 0

}

Cube {

width 5

height 0.2

depth 5

}


```
}  
  
#One Show Room  
  
Separator {  
  
Translation {  
  
translation 0 0 -5  
  
}  
  
DEF Room Separator {  
  
#The Entrance  
  
Separator {  
  
Transform {  
  
translation 0 0 2.5  
  
}  
  
Separator {  
  
Transform {  
  
translation -1.5 -1 0  
  
}  
  
DEF Side Cube {  
  
width 2  
  
height 2  
  
depth 0.2  
  
}  
  
}  
  
Separator {  
  
Transform {  
  
translation 1.5 -1 0  
  
}  
  
USE Side  
  
}
```

Separator {

Transform {

translation 0 1 0

}

Cube {

width 5

height 2

depth 0.2

}

}

Separator {

Transform {

translation 0 -1.9 0

}

Cube {

width 1

height 0.2

depth 0.2

}

}

}

#Setting the Walls

Separator {

Transform {

translation 0 0 -1.9

}

DEF Wall Cube {

width 5

height 4

depth 0.2

}

}

#Setting the Floor and Ceiling

Separator {

Translation {

translation 0 -1.9 0

}

DEF Floor Cube {

width 5

height 0.2

depth 5

}

}

Separator {

Transform {

translation 0 1.9 0

}

USE Floor

}

Separator {

Transform {

rotation 0 1 0 1.5708

translation -2.4 0 0

}

USE Wall

}

Separator {
Transform {
rotation 0 1 0 1.5708
translation 2.4 0 0
}
USE Wall
}
}
}
#Another Show Room

Separator {
Transform {
rotation 0 1 0 1.5708
translation -5 0 0
}
USE Room
}

#Another Show Room
Separator {
Transform {
rotation 0 1 0 -1.5708
translation 5 0 0
}
USE Room
}
}

Fig. 10.1

An environment with three rooms connected by a central space.

The code for this environment requires about 1.6K of file space. Moreover, you have not included any of the objects that are to be found in each room. In contrast, you can choose to render only the central space and the entrances to each room (see Fig. 10.2). This environment requires just over 1K of file space.

Fig. 10.2

Rendering only the central xspace and the entrances to each room.

You can set a WWWAnchor node at each entrance that calls up the appropriate room. In this way, you will only need to render the areas that are of interest to each customer. This will save a great deal of downloading time.

Now that you have the basic layout, you need to consider how you are going to display your CDs. This not only entails thinking about how to show the covers, it also requires you to think about the decor of the environment. What will induce someone looking for a rock record to wander through your environment? Should the colors be loud and bright? Do you think a rectangular room is adequate, or should it have an odd shape? Will having posters or other decorative items help make your environment attractive to potential users? Do you want to display your CDs on racks, or would it be more effective to have them floating in midair? Remember, you are not bound by physical laws in cyberspace!

And once you've made your decisions about the rock music room, can those decisions be used in the others? It may work for the rap room, but classical and folk music buyers will not be attracted to the same design.

Planning the layout of your commercial space requires a great deal of planning. You want to maximize the number of items that users who visit your space will want to buy. However, you also want to make sure that the amount of time it takes to download your VRML store is kept as low as possible. You don't want to make potential customers wait more than a minute or two. And your space should have a design that keeps their interest. The key is knowing your customers and designing the layout of your VRML space accordingly.

An Art Gallery

You are the curator of an art gallery, and you want to create a VRML replicate of it for the Internet. It would seem that all your design decisions have been made for you. Just recreate the space and the objects and post them on the Web.

However, you must consider a number of factors even when doing a recreation. First, you must ask yourself how much of the gallery will you present. If you are a large public gallery like the Museum of Modern Art in New York or the National Gallery in London, the size of your gallery is too large to include in a single environment. You may want to divide up the gallery space into different VRML environments even in the case of a small, private gallery. You may also want to avoid having to do a full representation of a gallery in favor of rendering only certain highlights of your gallery. Perhaps it would be enough to have a simple room with a few key pieces in it.

An even more important consideration is how to reproduce the art for a 3D environment. In the case of paintings, it is not simply a matter of creating bitmapped pictures that can be inserted into the environment. You need to balance the resolution needed to let someone look at the painting while minimizing the file size. And what about sculptures and other three-dimensional objects? It is expensive

and it also may not be physically possible to have 3D scans done. Will you be satisfied with a two-dimensional picture taking the place of your 3D sculpture? Will you try to model it using a 3D modeling program? And even if you can do a 3D scan, will the model be so complex that it is not practical to download over the Internet? Just as important, have you secured the permissions needed to re-create these art pieces from the artists and the owners?

Another issue is the use of the LOD node and lighting. If users are able to see objects at a distance, you can combine the LOD node with different lighting to reduce the rendering times needed for your environment. Using the LOD node, you can vary the complexity of a model according to the distance between it and the user. If the user is beyond a certain distance, you can use a much simpler model to re-create an object. As users move closer, you can increase its level of detail. (See [Chapter 14](#).)

And once you have your gallery's sculptures and paintings ready for your environment, will you add people in the scene? After all, galleries are public spaces, and there is nothing lonelier than finding yourself in an empty virtual environment. So the question then becomes, where should you position these "people?" Are there objects that you wish to highlight? And how should these people be constructed? Should they be bitmapped images of people? Can they simply be 3D models made of polygons and shapes?

A VRML Tutorial

Any VRML environment is, in essence, an information source. But when you begin planning to use one as a tutorial, you need to consider how a 3D world can present your material in the most effective way. How will you associate particular pieces of information with an object? Will you use the ASCIIText node to create pointers? Or will you let the objects "speak for themselves": that is, will you simply link an HTML document with a particular object and let users discover that link for themselves?

We also have to consider the kind of space that will house our tutorial. Some tutorials may need to have spaces that have closer ties to the real world. For example, a tutorial about building codes or how to use a library will be better served by a more realistic setting. However, since our tutorial is about VRML, we are free to use a more surrealistic setting. In this case, you need to decide what type of setting will best convey your message. Do you want a small, cluttered space that invites users to explore on their own? Do you want a large, sparsely furnished space that directs users to follow a particular path?

Moreover, any tutorial should reflect something of the personality of the instructor. We can do this with style in text and we can also do this with a virtual environment. A realistic setting with bland colors will focus users' attention to the objects in that room. If you wanted to add a little humor, you could populate it with strange-looking or ridiculous objects. Or you can use the design of the room itself to convey your humor. Would a room with no floor or an open space with a ceiling but no walls appeal to your aesthetic? Would it be useful to have oddly shaped walls? Would you rather have everything floating in space?

Deciding what objects to use to present particular material is also an important concern. Since we are doing a VRML tutorial, some of these associations are relatively straightforward. For example, we can use a cube to represent the Cube node and a sphere to represent the Sphere node. But how do we represent the Coordinate3 node or the IndexedFaceSet node? Should we label some of the links using the description field? How detailed do we want our tutorial to be? Will this entail having more objects? Will we have one object represent more than one topic?

And again, the size of the file is an important consideration. If we have a small environment with a large number of objects, the range of movement in our environment will be small. Is that what we want to have in a tutorial about VRML? Moreover, we do not want to make users wait too long for information to be downloaded. Too long a wait will result in users abandoning the tutorial.

Mapping Out Your Environment

Once you have considered all the factors, you are ready to start mapping out your environment. In other words, you are going to take your ideas and convert them into visual images. Let's take our VRML tutorial as an example.

After carefully weighing our options, we have decided to create a simple tutorial focusing on the primitive nodes (cube, sphere, cylinder, and cone), the IndexedFaceSet and IndexedLineSet, and the Transform node. This means that we do not have to have a large number of objects or link more than one tutorial module with an object. This will also allow us to create a large environment that lets users get a greater sense of movement.

We can start by doodling: trying out different ideas by doing small sketches. You don't have to produce a detailed rendering on paper at this stage. What is important is to get your ideas down on paper so that you can choose a design that appeals to you.

After jotting down some designs, we look at one of them and believe that it has some potential. In our example, we decide that rather than a room or a series of rooms connected with each other, we can have a large environment that has rooms and objects floating within it. Users would then have to "fly" through the environment. Moreover, we can set the rooms and objects in such a way that users would follow a particular sequence (see Fig. 10.3).

[Fig. 10.3](#)

Using different levels to control users' navigation of an environment.

Once we have our design in mind, we need to do a more detailed map in order to work out the coordinates and positions for the various objects. One method is to use two pieces of graph paper: one for the front view and one for the top view. Mark a point at the center of both as the origin (0 0 0). Using each square as one unit, begin placing all your objects. Remember, you can move the initial view using the camera nodes, and so you do not have to place the opening view at the origin. Don't worry about rotations or other transforms at this point. The main thing is to figure where everything should be and not on how it will get there.

You should draw both top and front views of your plan. Remember that the top view's y-coordinate will always be zero. This will be compensated by the front view, which will always have a z-coordinate value of 0. You can then use the x-coordinate to help you put the two views together.

Once you have finished mapping your environment, you are ready to set up the coordinates and transformations.

The Cartesian Coordinate System and Basic Mathematics for VRML

While it is important to have a visual way of thinking to design your VRML environment, the way that it's stored is as a set of coordinates and mathematical transformations. You will need to convert your visual design into these coordinates and transformations. And in order to accomplish this, you need to know a bit about coordinate systems, vectors, and matrices. A more detailed discussion about the mathematics of VRML can be found in Appendix A, "Essential Math for Using VRML."

The Cartesian Coordinate System

I always remember my aunt who was blind, because despite having a house that was full of objects that she couldn't see, she always was able to walk around her house without bumping into things. It was because she remembered the position of different objects relative to certain starting points. She knew where the coffee table was relative to the front door of the house. And she knew where the same table was relative to the entrance from the kitchen.

Like my aunt, VR systems need to have a way of knowing where objects are positioned. Like my aunt, these systems need to know where something must be placed relative to a starting point. VRML uses a right-handed, three-dimensional Cartesian coordinate system.

The term "right-handed" refers to the relative orientation of the axes. Figure 10.4 shows the two types of systems. The right-handed system is used because it is easier to visualize how it works and because it is widely accepted as a standard.

[Fig. 10.4](#)

The (a) left-handed and (b) right-handed coordinate systems.

In a three-dimensional world, you need to provide three coordinates (x, y, and z) to specify the location of a point. In VRML, the size of a single unit represents one meter. Figure 10.5 shows how such a system would look as a graph.

Fig. 10.5

The Cartesian coordinate system.

We can describe a point as being left or right, above or below, and in front of or behind the axis. For example, a point that is one meter to the right, 2 meters above, and one meter in front of the axis would be written as (1 2 1). Conversely, a point that is one meter to the left, 2 meters below, and one meter behind the axis would be written as (-1 -2 -1) (see Fig. 10.6).

Fig. 10.6

Defining points on a right-handed, Cartesian, three-dimensional coordinate system.

Vectors

A point in the Cartesian coordinate system is a vertex. A vertex is simply a location in space represented by three numbers: x, y, and z. You will often see the term vector used in place of vertex. The two terms are, in fact, related to each other. Like a vertex, a vector is represented by x, y, and z. However, whereas a vertex represents a point in space, a vector specifies a direction.

From Maps to Coordinates

Now that we have some understanding of how the coordinate system for VRML works, we can begin to convert our maps into coordinates. We can begin by looking at objects that we want to create. For example, we want to have a six-by-eight-by-six rectangular room in our VRML tutorial. The center point for this room is to be positioned three units to the right of the axis, two units below it, and two units behind it. The origin is at (0 0 0). Therefore, the center point of our room is (3 -2 -2). We can now plot out the coordinates of the eight corner points of our room by adding or subtracting one-half of its length, width, and height from the center point coordinates. The result would be:

0 2 2

6 2 2

6 -6 2

0 -6 2

0 -6 -4

0 2 -4

6 2 -4

6 -6 -4

We can also visualize our room with its coordinates (see Fig. 10.7)

Fig. 10.7

A 6 by 8 rectangular room whose center is (3 -2 -1).

We would repeat this process for all the objects that we want to create in this way.

The next step would be to calculate the various transforms that you will need. The first step is to find out the center point for each object that you want to transform. You then need to figure out which types of transforms you will need. Do you need to scale an object? Does it need to be moved to another location? Does it need to be rotated?

If you want to scale an object, you need to know by what factor you want to stretch or shrink the object on each axis. If you want to move an object to another location, you need to know the difference between the coordinates of the final location and that of the starting location. For example, to translate an object that has a center point of (1 0 1) to the coordinate (10 7 5), you would need a translation of (10-1 7-0 5-1)

or (9 7 4). To rotate an object, you need to calculate the number of radians that you want the object to be rotated. A quick guide is that 45 degrees is equal to 0.7854 radians and 90 degrees is equal to 1.5708 radians.

Once you have recorded all your coordinates and transformation factors, you are ready to begin building your VRML environment.

From Here...

An important first step in programming a VRML environment is planning it out. Once you have conceptualized and plotted out your world, you can move on to the actual coding stage. To start creating your VRML world, or to learn more about the mathematics of VRML, see the following chapters:

ï [Chapter 11, "VRML Standards and Practices,"](#) describes the VRML standard and outlines the conventions used by the language.

ï [Chapter 12, "Putting Up Walls,"](#) describes how to set up a room in a VRML environment.

ï Part VIII, "The Principles of Artificial Environment Design," discusses a number of issues related to the design of virtual environments and objects. It includes a discussion of architecture and industrial design and how these disciplines can help us with our virtual worlds.

ï [Appendix A, "Essential Math for Using VRML,"](#) is a more detailed discussion about the mathematics underlying VRML and how it works.



Notice: This material is excerpted from [Special Edition Using VRML](#), ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



11-VRML Standards and Practices

Well, it's time to get into the nitty-gritty details of VRML itself. In this chapter, you'll learn to:

- Understand the basic syntax of VRML, including nodes and fields
- Recognize the data types used within VRML
- Use the various shape nodes
- Create polygonal objects
- Use grouping nodes to create hierarchies
- Build complete VRML scenes from scratch

We have lots to do, so let's get started...

Basic Syntax of VRML

VRML files are plain ASCII, which means you can create and edit them using an ordinary text editor such as "notepad" under Windows, "edit" under DOS, or "vi" on UNIX. If you use a word processing package (such as Word) you should be sure to save the files as plain ascii text (e.g., Text Only with Line Breaks).

Let's take a look at a simple VRML file:

Listing 11.1

#VRML V1.0 ascii

This file defines a simple red sphere

Separator {

Material { diffuseColor 1 0 0 } # the color red

Sphere { }

}

Everything after a # on any line is a comment, which continues to the end of the line (such as "# the color red"). Other than ignoring comments, the file format is entirely free-form; anywhere where you can have one space, you can have as many as you like. You can also use tabs, or even blank lines.

The first line of the file begins with a #, so it's a comment. However, it's a very special comment: it identifies the file as being VRML, and gives the version number. Browsers require this information to be at the start of any VRML file.

In theory, you can put other text after the word "ascii"; in practice, it's probably better to end the line there and put any additional comments on separate lines. There's no guarantee that extra text on the line won't cause problems for some browsers.

The example above says "V1.0," meaning the file conforms to version 1.0 of the VRML specification. Version 1.1 is in the works, and should be finalized by the time you read this. Except where otherwise noted, everything presented in this book should be the same in both version 1.0 and version 1.1, but be sure to check the actual format specifications for 1.1 once they're available. Version 1.1 files will have the header "#VRML V1.1 utf8." The word "ascii" means that the standard ASCII character set is used in the file; "utf8" means an international character set (based in part on ascii) is being used.

Nodes and Fields

If you happen to be a C programmer, you'll recognize the use of curly braces ("{" and "}") to define blocks of related information. VRML files are made up of things called "nodes," which look like this:

```
NodeType { stuff }
```

The NodeType refers to one of the types of nodes described in the VRML specification, which can be found in Appendix B. At the moment, there are just over 30 different types of nodes; fortunately, we don't need to deal with all of them just yet. The example above uses three different types of nodes: Separator, Material, and Sphere.

The stuff inside the braces consists mostly of things called "fields." In our example, only the Material node has any fields; it has just one, called "diffuseColor." Each field has a name and a value. In this case, the diffuseColor field has the value "1 0 0," a set of three numbers that give the color to use for the Sphere which follows (it lists the color's components in the order Red, Green, and Blue).

Field values can be simple numbers, groups of numbers (like 1 0 0), strings, keywords, images, boolean values, and more. Some fields can have multiple values, in which case the values are separated by commas and surrounded by square brackets. For example, if you had four values for diffuseColor (each consisting of three numbers) they'd be written as

```
Material { diffuseColor [ 1 0 1, 0.5 0.75 0, 0 1 0, 0.3 0.3 0.2 ] }
```

A comprehensive list of the different types of fields and their meanings is given in the VRML specification in Appendix B. Every field has a default value; if you leave out a field, it uses its default. For example, the default diffuseColor is 0.8 0.8 0.8, and the default radius for a sphere is 1 meter.

Naming Nodes

Any node can be assigned a name, by which it can be referred to later. This is done with the DEF prefix, as follows:

DEF Frodo Separator { }

The word DEF is short for "define," so this would define the name "Frodo" to refer to that particular Separator node. We'll be talking about different uses for this name later on.

Objects, Hierarchies, and the Separator Node

As you may recall from Chapter 7, a virtual world can be thought of as a hierarchy of objects. The current version of VRML doesn't have an actual "Object" node; instead, all the information related to an object is stored inside a Separator. In other words, you can think of a "Separator" as a container for an object. Not all Separators necessarily contain any geometry (vertices and faces); some are just used for grouping other objects together as a unit.

You may be wondering about how you specify object hierarchies in VRML. In VRML, the attachment information is specified by placing objects "inside" other objects. In other words, the Separator node has "children" nodes inside it that describe objects that are attached to it. Those children can in turn contain children, and so on.

Confused? Let's take the human body, and represent it as a tree diagram the way we did back in Chapter 7:

Now let's create a VRML file to represent the body:

[Fig. 11.1](#)

The hierarchical structure of the human body.

Listing 11.2

#VRML V1.0 ascii

A simple humanoid body made up of cylinders and spheres

Built (by hand, with a text editor!) by Bernie Roehl, September 1995

DEF Body Separator {

DEF Pelvis Separator {

DEF Chest Separator {

DEF LeftUpperArm Separator {

DEF LowerArm Separator {

}

```
}  
  
DEF RightUpperArm Separator {  
  
DEF LowerArm Separator {  
  
}  
  
}  
  
DEF Head Separator {  
  
DEF Skull Separator {  
  
}  
  
DEF LeftEye Separator {  
  
}  
  
DEF RightEye Separator {  
  
}  
  
DEF Nose Separator {  
  
}  
  
}  
  
}  
  
DEF LeftThigh Separator {  
  
DEF Calf Separator {  
  
}  
  
}  
  
DEF RightThigh Separator {  
  
DEF Calf Separator {  
  
}  
  
}  
  
}
```

The information about the actual shapes and materials in the body has been left out of this example, in order to make the structure clearer. The main thing to notice is that each Separator's children are contained within it.

Only Separator nodes have children (that's not quite true, as we'll see later, but we'll leave that for the time being). The primary purpose of Separator nodes is to define an object hierarchy.

Separators

So Separators correspond loosely to objects in the virtual environment, and the nesting of Separators within each other corresponds to the hierarchical attachment of the objects. A Separator can also contain information about the shape of the object it describes (like the Sphere node in our first example), as well as specifications for the surface properties of the object (like the Material node).

In addition to all of this, a Separator can give the location, orientation, and size of objects relative to their parents. This is done using "transforms" of the type described back in Chapter 7.

So a Separator really looks like this:

```
Separator {  
transforms  
surface properties  
shapes  
children  
}
```

If the surface properties aren't specified, they're "inherited" from the parent object. For example, if you don't have a Material node in the current Separator, the Material from the parent Separator is used. If the parent doesn't have one either, the grandparent's is used, and so on. If a particular surface property isn't specified by the object, or its parent, or any of its ancestors, then certain default values are used. These are listed in detail in the VRML specification.

.

Intermingling Nodes

The VRML 1.0 specification allows the transforms, surface properties, shapes, and children to appear in any order; the transforms and properties apply only to those children and shapes which followed them. For example, you might say

```
Separator {  
Material { diffuseColor 0 1 0 }  
Transform { translation 2 5 7 }  
Cone { }  
Material { diffuseColor 1 0 0 }  
Transform { translation 8 9 10 }
```

```
Sphere { }
```

```
}
```

This would create a green cone, translated to (2, 5, 7). It would also create a red sphere, translated to (10, 14, 17); the transforms would "accumulate," so the location winds up being (2+8, 5+9, 7+10).

There are several proposals for making VRML more "object-oriented" (or to put it another way, to use "higher-level" objects). It seems very likely VRML will move in this direction in the future.

For compatibility with future versions of the specification, you should get used to not intermingling the different types of nodes contained within a Separator. This does not constrain in any way the worlds you can create.

Multiple-shape nodes can appear within one Separator, if necessary. For example, you might have something like this:

```
Separator {
```

```
Material { diffuseColor 1 0 0 }
```

```
Transform { rotation 1 0 0 0 5 }
```

```
Coordinate3 { point [ ... ] }
```

```
IndexedFaceSet { coordIndex [ ... ] }
```

```
IndexedLineSet { coordIndex [ ... ] }
```

```
}
```

The "_" aren't really part of the file, of course; actual coordinate values or coordinate index values would appear there, but they've been left out in order to make things a little more readable. This example illustrates that two shape nodes (the IndexedFaceSet and the IndexedLineSet) can share a material, a transformation, and even a set of coordinates.

If no shape is specified in a Separator, the object has no visual appearance; it's basically just an invisible placeholder, an attachment point for its children. This can be handy for joining a collection of objects together.

There are two other nodes that act like a Separator: Transform Separator and Group. Their use is being discontinued, and they won't be in future versions of the VRML specification; my strong recommendation is that you not use them. If you really need to know about them, see their descriptions in [Appendix B](#).

One very important point that's worth noting is that a VRML file describes exactly one object, which may contain subobjects (i.e., children). That means that after the header line (and other comments), there should be one big Separator node containing everything else.

Now that we know what Separators contain, let's take a closer look at some of their individual pieces.

Transforms

The scaling, rotation, and translation transformations described in [Chapter 7](#) can be specified easily in VRML. The translation and scaling are represented by vectors, and rotations are represented in axis-angle form by four numbers (three for the axis, one for the angle).

The rotation angle is not given in degrees; it's in a unit called "radians." To convert from degrees to radians, multiply by $\text{PI}/180.0$, where PI is 3.14159; this is because there are PI radians or 180 degrees in a half-circle. Translations are measured in meters (one meter is 39.37 inches, or about three feet).

There are several ways to specify transforms in VRML. The simplest is the Transform node, which combines scaling, rotation, and translation:

```
Transform {  
center x y z  
scalingFactor x y z  
scaleOrientation x y z a  
rotation x y z angle  
translation x y z  
}
```

The "center" field specifies a three-dimensional point around which the object is scaled and rotated. It's important to remember that this is a point in the object's coordinate system; for example, if you have a sphere with a radius of 2 units, but specify "center -2 -0 -0," then it will rotate around a point on the surface of the sphere. This will be true regardless of where in the world the sphere is; in other words, the center is given in "object space" not "world space."

The "scaleOrientation" field specifies the way to rotate the object's coordinate system before scaling it; this allows the object to be scaled in unusual ways.

The fields can be listed in any order, but the order in which the operations are performed is always the same: scaling, then rotation, then translation. In other words, the object is first shifted so that its origin is at the given center point, then it's rotated according to its scaleOrientation, then scaled, then rotated back. Then it's rotated by the rotation value, shifted so that its origin is back at its original center point, and then shifted again according to the translation.

That all sounds pretty complicated, but most of the time it turns out to be exactly what you want. The defaults are what you would expect: the center is at 0 0 0, the scaling factors are all 1.0, the rotation angle is zero, and the translation values are all zero. If all you want to do is rotate the object by 45 degrees around the Y axis and then move it right a half a meter, you would do the following:

```
Transform {  
rotation 0 1 0 0.7854
```

translation 0.5 0 0

}

In case you need even more control over location, orientation, and scale than is provided by the Transform node, VRML has nodes that do each transformation as a separate step:

Translation { translation x y z }

Rotation { rotation x y z angle }

Scale { scaleFactor x y z }

You can have as many of these as you like, in any order. You can even mix-and-match them with Transform nodes. The order is "backwards" in that the last transformation you specify gets applied to the object first, and the first one last. In other words,

Separator {

Scale { scaleFactor 2 1 3 }

Rotation { 0 1 0 0.7854 }

Cube { }

}

would first rotate the cube 45 degrees (0.7854 radians) around the Y axis (0 1 0), then stretch the rotated cube by a factor of 2 in the (new) X direction and a factor of 3 in the (new) Z direction. The scale factor for Y is 1, meaning that the Y size doesn't change. This sequence of operations is illustrated in Figure 11.2:

[Fig. 11.2](#)

Rotating and then stretching a cube.

There's one final transformation node, called MatrixTransform. It looks like this:

MatrixTransform {

matrix 16 numbers

}

The sixteen numbers are the elements of a 4-by-4 matrix. If you don't understand what a "matrix" is, don't worry about it; the MatrixTransform node is mostly useful for people who are converting from other formats, and who therefore have access to the transformation matrix for an object. If you're curious about what matrices are, and how they relate to transforms, then check out Appendix A.

.

Unlike most other systems, VRML uses column-oriented matrices; in other words, a vector is multiplied by the first vertical column in order to obtain the X component of the transformed vector (the same is done for Y and Z).

Starting with version 1.1, VRML requires that the matrices be the result of rotation, translation, and scaling only; in other words, the final column should be 0 0 0 1. The matrix must also be invertible; see Appendix A for details.

All the transforms are applied relative to an object's parent; in other words, if you rotate your lower arm relative to your upper arm, then rotate your upper arm at the shoulder, your lower arm maintains its correct relationship to your upper arm.

It's important to remember that transforms apply not only to an object, but also to its children. In particular, the Scale node scales the entire subtree of the object hierarchy, not just the one object.

If you need to scale just the one object, you should make it a separate subtree by putting it inside its own Separator.

The ability to scale, rotate, or translate an entire subtree is useful; you can re-size a human figure and move it somewhere else just by scaling and translating its top-level object (the pelvis, in the case of our human-body example).

Shapes

VRML lets you create a variety of different shapes: spheres, cones, cylinders, and cubes. Let's examine each one of them in detail.

Sphere

The Sphere node is very simple; it has only one field, called "radius," that gives the radius of the sphere in meters:

```
Sphere {  
radius radius  
}
```

The default radius is one meter. Here's what the sphere looks like:

[Fig. 11.3](#)

A standard sphere, displayed in WorldView.

Cone

A cone has two parts to it: the sides and the bottom. Both are optional; by default a cone has both, but the "parts" field can specify either one or the other (either SIDES or BOTTOM). A cone also has a height, and a radius value for its base.

```
Cone {
```

bottomRadius radius

height height

parts parts

}

The default bottomRadius is one meter, and the default height is two meters. The cone's origin is at its geometric center; that is, it's halfway up the height of the cone along the main axis.

Here's a standard cone:

[Fig. 11.4](#)

A standard cone, displayed in WorldView.

Cylinder

A cylinder is similar to a cone, but has three parts: the sides, the bottom, and the top. The parts field can have any combination of the three, joined by vertical bars and surrounded by parentheses; for example, a cylinder that's closed only at the bottom would have a parts field value of (SIDES|BOTTOM). The syntax for a cylinder is

Cylinder {

parts parts

bottomRadius radius

height height

}

The default bottomRadius is one meter, the default height is two meters. The origin of the cylinder is halfway up its length, on its main axis. Here's what the cylinder looks like:

[Fig. 11.5](#)

A standard cylinder, displayed in WorldView.

.

You can use the value ALL for the parts field if you want all the parts of a cone or cylinder to be included, but since that's the default value you don't need to specify it.

.

Remember that the Sphere, Cone, and Cylinder nodes are just a kind of "shorthand"; once the VRML file is parsed, they're converted into vertices and faces by a process called "tessellation." To see how this works, try creating a short VRML file with a Sphere node in it, load it into a browser, and then switch to flat-shading mode to see the individual faces.

Cube

A cube is one of the simplest shapes; it just has a height, width, and depth:

```
Cube {  
  width width  
  height height  
  depth depth  
}
```

All three fields have a default value of 2.0. Since the origin of the cube is at its center, the sides of the cube are all one meter from its origin. Figure 11.6 shows what a cube looks like.

[Fig. 11.6](#)

A standard cube, displayed in WorldView.

AsciiText

In addition to the four basic shapes, there's a node called AsciiText that lets you create flat text for making things like signs. The syntax is as follows:

```
AsciiText {  
  string "string"  
  spacing spacing  
  justification justification  
  width width  
}
```

The string is surrounded by double quotes. There can be multiple strings, separated by commas and surrounded by square brackets; each string is placed on a new line, separated by the vertical spacing (which is a factor by which the height of each letter is multiplied; in other words, 2 means double-spaced). The justification can be LEFT, RIGHT, or CENTER. For example, the following node:

```
AsciiText {  
  string [ "this is", "a test", "of AsciiText" ]  
  justification CENTER  
}
```

would produce the following result:

Fig. 11.7

Some AsciiText seen in WorldView.

It's been proposed that in future versions of the VRML specification, the AsciiText node be renamed to simply Text, since the character set used may not always be ascii. There will likely be other changes to the node as well.

The AsciiText node can be modified by preceding it with a FontStyle node:

```
FontStyle {  
size size  
family family  
style style  
}
```

where size is the height of the text (in meters!), family is one of SERIF, SANS, or TYPEWRITER (usually equivalent to Times, Helvetica, and Courier), and style is either NONE (the default), BOLD, ITALIC, or (BOLD|ITALIC).

For example,

```
FontStyle {  
size 12  
family SANS  
style BOLD|ITALIC  
}
```

would cause the AsciiText to be 12 meters high, in a bold-italic, sans-serif font. The result looks like this:

Fig. 11.8

The FontStyle node changes the typeface of an AsciiText node.

Note that the FontStyle is inherited by children of the current Separator, in the same way that a surface property (like a Material) is.

Here's an example that uses all the primitive geometric shapes we've examined so far:

Listing 11.3

#VRML V1.0 ascii

```
Separator {
```

```

Separator {
Cone { }
}
Separator {
Transform { translation 0 2 0 } # positioned on top of _the cone
Sphere { }
}
Separator {
Transform {
translation 5 0 5
rotation 1 0 0 0.7854 # tip cylinder 45 degrees _over X axis
}
Cylinder { }
}
Separator {
Transform { translation -5 0 0 }
Cube { }
}
}

```

which looks quite nice when it's viewed:

Fig. 11.9

A complete scene in VRML.

Now is a good time to go back to our body example. Here's the human body in all its glory, using cylinders and spheres:

Listing 11.4

```

#VRML V1.0 ascii
# A simple humanoid body made up of cylinders and spheres
# Built (by hand, with a text editor!) by Bernie Roehl, September _1995
DEF Body Separator {

```

DEF Pelvis Separator {

Cylinder { radius .2 height .15 }

DEF Chest Separator {

Transform { translation 0 .35 0 }

Cylinder { radius .2 height .50 }

DEF LeftUpperArm Separator {

Transform { translation -.275 .10 0 }

Cylinder { radius .05 height .30 }

DEF LowerArm Separator {

Transform { translation 0 -.280 0 }

Cylinder { radius .04 height .25 }

}

}

DEF RightUpperArm Separator {

Transform { translation .275 .10 0 }

Cylinder { radius .05 height .30 }

DEF LowerArm Separator {

Transform { translation 0 -.280 0 }

Cylinder { radius .04 height .25 }

}

}

DEF Head Separator {

Transform { translation 0 .425 0 scaleFactor

DEF Skull Separator {

Sphere { radius .1 }

}

DEF LeftEye Separator {

Transform { translation -.04 .025 .0707

Material { diffuseColor 0 0 1 }

Sphere { radius .02 }

}

DEF RightEye Separator {

Transform { translation .04 .025 .0707

Material { diffuseColor 0 0 1 }

Sphere { radius .02 }

}

DEF Nose Separator {

Transform { translation 0 -.01 .1 }

Material { diffuseColor 1 0 0 }

Sphere { radius .02 }

}

}

}

DEF LeftThigh Separator {

Transform { translation -.1 -.275 0 }

Cylinder { radius .075 height .35 }

DEF Calf Separator {

Transform { translation 0 -.33 0 }

Cylinder { radius .06 height .30 }

}

}

DEF RightThigh Separator {

Transform { translation .1 -.275 0 }

Cylinder { radius .075 height .35 }

DEF Calf Separator {

Transform { translation 0 -.33 0 }

```
Cylinder { radius .06 height .30 }
```

```
}
```

```
}
```

```
}
```

```
}
```

Which gives us the following friendly looking figure:

You should study this VRML file closely, to see how the hierarchies, transforms and shapes relate to each other and to Figure 11.10.

[Fig. 11.10](#)

A human body, not anatomically correct.

IndexedFaceSet

The nodes we've looked at so far (Sphere, Cone, Cylinder, Cube, and AsciiText) are useful when you're just starting out, or if you're building simple worlds by hand. However, most VRML files make extensive use of another kind of shape: the IndexedFaceSet.

An IndexedFaceSet is a way of describing an object using a set of vertices that are joined together by faces, exactly the way we discussed back in Chapter 7. For example, you could create a pyramid using five vertices and five faces (four sides and a base):

Listing 11.5

```
#VRML V1.0 ascii
```

```
DEF Pyramid Separator {
```

```
Coordinate3 {
```

```
point [
```

```
-1 0 -1,
```

```
1 0 -1,
```

```
1 0 1,
```

```
-1 0 1,
```

```
0 1 0
```

```
]
```

```
}
```

```
IndexedFaceSet {
```

```
coordIndex [
```

```
0, 4, 1, -1,
```

```
1, 4, 2, -1,
```

```
2, 4, 3, -1,
```

```
3, 4, 0, -1,
```

```
0, 1, 2, 3, -1
```

```
]
```

```
}
```

```
}
```

Here's what it looks like:

[Fig. 11.11](#)

A pyramid, created using an IndexedFaceSet.

The array of vertex coordinates is specified using the Coordinate3 node; it has one field ("point") with multiple values, each of which is a triplet of numbers giving the coordinates of the vertex. Each set of X, Y, and Z values are the coordinates of one vertex. There can be as many coordinates as you like; however, keep in mind that each vertex only has to be listed once, not once for each face it's in. In other words, the peak of the pyramid will only appear in the list once, even though it's part of each of the four side faces.

The actual IndexedFaceSet node contains a field called coordIndex that stores a list of the faces, specified by the indices of vertices in the order they get used to define the faces (as described in [Chapter 7](#)). The sequence "0, 4, 1, -1" is used to create one face made up of the 0th, 4th, and 1st vertices. (Remember that computers start counting from zero, not from one.) The -1 means "that's it for this face, on to the next one!" The next face starts with the next vertex index number.

The pyramid example has four faces that are made up of three vertices each, and one face (the base) that's made up of four vertices.

Here's another example of an IndexedFaceSet, this one created by a three-dimensional modeling program:

Listing 11.6

```
#VRML V1.0 ascii
```

```
Separator {
```

```
Separator {
```

```
MaterialBinding {
```

value OVERALL

}

Material {

ambientColor [

0.100 0.100 0.100,

]

diffuseColor [

0.824 0.824 0.824,

]

specularColor [

0.100 0.100 0.100,

]

emissiveColor [

0.000 0.000 0.000,

]

shininess [

0.000,

]

transparency [

0.000,

]

}

MatrixTransform {

matrix

1.000 0.000 0.000 0

0.000 0.000-1.000 0

0.000 1.000 0.000 0

-1.000 0.000 0.000 1

}

Coordinate3 {

point [

3.201 0.231 -1.406,

3.222 0.195 -2.523,

2.730 1.004 -3.117,

2.217 1.848 -2.594,

2.195 1.884 -1.477,

2.687 1.075 -0.883,

2.000 -0.500 1.000,

1.933 0.616 1.000,

0.933 1.116 1.000,

0.000 0.500 1.000,

0.067 -0.616 1.000,

1.067 -1.116 1.000,

2.131 -0.420 1.574,

2.015 0.666 1.337,

1.071 1.200 1.607,

0.242 0.647 2.114,

0.357 -0.439 2.351,

1.302 -0.973 2.081,

2.495 -0.199 1.985,

2.244 0.805 1.562,

1.453 1.433 2.043,

0.914 1.056 2.947,

1.165 0.052 3.370,

1.955 -0.576 2.889,

3.013 0.116 2.130,

2.569 1.003 1.614,
1.998 1.764 2.201,
1.870 1.637 3.305,
2.314 0.751 3.821,
2.885 -0.010 3.234,
3.572 0.456 1.961,
2.920 1.217 1.464,
2.585 2.121 2.029,
2.901 2.265 3.091,
3.553 1.505 3.589,
3.888 0.600 3.023,
4.050 0.747 1.494,
3.220 1.399 1.124,
3.087 2.427 1.545,
3.784 2.802 2.335,
4.613 2.150 2.704,
4.746 1.122 2.284,
4.343 0.925 0.810,
3.404 1.511 0.649,
3.395 2.614 0.832,
4.325 3.131 1.178,
5.263 2.545 1.339,
5.272 1.442 1.156,
4.387 0.952 0.038,
3.432 1.528 0.120,
3.441 2.642 0.027,
4.406 3.180 -0.148,
5.361 2.604 -0.230,

5.351 1.490 -0.137,
4.172 0.822 -0.674,
3.297 1.446 -0.367,
3.216 2.505 -0.716,
4.010 2.939 -1.373,
4.885 2.315 -1.680,
4.966 1.256 -1.331,
3.746 0.562 -1.188,
3.030 1.283 -0.723,
2.768 2.232 -1.252,
3.223 2.461 -2.248,
3.939 1.740 -2.714,
4.201 0.790 -2.184,

l

}

ShapeHints {

creaseAngle 0.559

vertexOrdering CLOCKWISE

shapeType SOLID

faceType UNKNOWN_FACE_TYPE

}

DEF NoName_1 IndexedFaceSet {

coordIndex [

1, 2, 3, 4,

5, 0, -1, 60,

0, 5, 61, -1,

61, 5, 4, 62,

-1, 62, 4, 3,

63, -1, 63, 3,
2, 64, -1, 64,
2, 1, 65, -1,
65, 1, 0, 60,
-1, 6, 7, 8,
9, 10, 11, -1,
13, 7, 6, 12,
-1, 14, 8, 7,
13, -1, 15, 9,
8, 14, -1, 16,
10, 9, 15, -1,
17, 11, 10, 16,
-1, 12, 6, 11,
17, -1, 19, 13,
12, 18, -1, 20,
14, 13, 19, -1,
21, 15, 14, 20,
-1, 22, 16, 15,
21, -1, 23, 17,
16, 22, -1, 18,
12, 17, 23, -1,
24, 25, 19, -1,
18, 24, 19, -1,
26, 20, 19, 25,
-1, 27, 21, 20,
26, -1, 28, 22,
21, 27, -1, 29,
23, 22, 28, -1,

24, 18, 23, 29,
-1, 30, 31, 25,-1,
24, 30, 25, -1,
32, 26, 25,
31, -1, 33, 27,
26, 32, -1, 34,
28, 27, 33, -1,
35, 29, 28, 34,
-1, 30, 24, 29,
35, -1, 37, 31,
30, 36, -1, 38,
32, 31, 37, -1,
39, 33, 32, 38,
-1, 40, 34, 33,
39, -1, 41, 35,
34, 40, -1, 36,
30, 35, 41, -1,
43, 37, 36, 42,
-1, 44, 38, 37,
43, -1, 45, 39,
38, 44, -1, 46,
40, 39, 45, -1,
47, 41, 40, 46,
-1, 42, 36, 41,
47, -1, 49, 43,
42, 48, -1, 50,
44, 43, 49, -1,
51, 45, 44, 50,

-1, 52, 46, 45,
51, -1, 53, 47,
46, 52, -1, 48,
42, 47, 53, -1,
55, 49, 48, 54,
-1, 56, 50, 49,
55, -1, 57, 51,
50, 56, -1, 58,
52, 51, 57, -1,
59, 53, 52, 58,
-1, 54, 48, 53,
59, -1, 61, 55,
54, 60, -1, 62,
56, 55, 61, -1,
63, 57, 56, 62,
-1, 64, 58, 57,
63, -1, 65, 59,
58, 64, -1, 60,
54, 59, 65, -1,
|
}
}
}

And here's what the object looks like in wireframe mode:

[Fig. 11.12](#)

An object created using an IndexedFaceSet, shown in wireframe mode.

You can give rendering hints to the IndexedFaceSet node by using the ShapeHints node. We'll be talking about ShapeHints again when we discuss lighting, but for now there are two fields you should know about:

```
ShapeHints {
  vertexOrdering ordering
  faceType type
}
```

The vertexOrdering field can be CLOCKWISE, COUNTERCLOCKWISE, or UNKNOWN_ORDERING. As you may remember from Chapter 7, vertices can be listed in either a clockwise or counter-clockwise order around the face; the ShapeHints node tells the browser which order is being used for the following IndexedFaceSet nodes. If you specify UNKNOWN_ORDERING, then the browser will have to draw the face regardless of which side is facing forward.

VRML 1.0 had the vertexOrdering field default set to be UNKNOWN_ORDERING, which put an increased burden on the rendering engine. Starting with version 1.1, the default is COUNTERCLOCKWISE. To be on the safe side, specify the vertex ordering explicitly where needed, and don't rely on the default value. Remember that ShapeHints are inherited, so if all the objects you define have consistent ordering, you only need to specify it once near the top of the file.

The faceType field can be either CONVEX (the default) or UNKNOWN_FACE_TYPE. Most of the time you don't need to worry about this, but if you have faces that have concavities in them, like this:

[Fig. 11.13](#)

A face with a concavity.

- then you should specify faceType UNKNOWN_FACE_TYPE.

The ShapeHints node, like the FontStyle node and all the surface property nodes, is inherited; that is, if you don't specify it, the parent's value is used (or the grandparent's, or great-grandparent's, and so on).

Some Hints on Using IndexedFaceSet

Although IndexedFaceSet is a very powerful tool, there are some things to watch out for when you're using it.

Make sure that all the vertices of a face are coplanar, meaning that the face is flat. If one or more of the vertices are not in the same plane as the others, then the object will look very strange indeed from certain viewing angles.

Avoid T-intersections. Two faces should always meet at a shared edge; if you have a face that touches another face without sharing common vertices, then round-off error will cause peculiar visibility problems.

Avoid using too many faces, since that will slow down the rendering more than any other single factor.

IndexedLineSet

Sometimes you don't want to create polygon-based figures, but are satisfied simply to use lines. For example, the spokes of a bicycle wheel might best be represented by simple lines, rather than by Cylinder nodes; not only is it easier, it will also render much faster.

To create sets of lines, you use the IndexedLineSet node. The syntax is the same as for the IndexedFaceSet, but the results are different; here's the pyramid example we used above, this time as an IndexedLineSet:

[Fig. 11.14](#)

The pyramid constructed as an IndexedLineSet.

PointSet

There are times when you don't need polygons, or even lines; what you really want are individual points. If you're creating stars at night, you should use points rather than lots of tiny spheres. As with the bicycle-wheel example, a set of points is easier to create and a lot faster to render. The syntax for the PointSet node is

```
PointSet {  
startIndex index  
numPoints num  
}
```

The index gives the index of the first vertex in the vertex array that should be used, and num gives the number of consecutive vertices to use. The result is sometimes called a "point cloud."

Here's an example of a PointSet:

```
Separator {  
Coordinate3 {  
point [  
-1 0 -1,  
1 0 -1,  
1 0 1,  
-1 0 1,  
0 1 0  
]  
}
```

```
}  
  
PointSet { }  
  
}
```

Notice that we've left out both the `startIndex` and `numPoints` fields; the `startIndex` defaults to zero, and `numPoints` defaults to 61 (which means to use all the subsequent points in the array of vertices). The vertex coordinates are the same as those from our pyramid example, so the result is a point cloud with vertices (rendered as single pixels) at the corners of the pyramid:

Fig. 11.15

A point cloud, with vertices where the corners of a pyramid would be.

The Info Node

The Info node provides a way of storing textual information in a node. This can be used for whatever the application needs; the browsers generally ignore it, since the information it contains is usually only of interest to humans and has no effect on rendering. The syntax is simply:

```
Info {  
  
  string "some text"  
  
}
```

Some browsers support the use of named Info nodes to give hints as to how they should process the scene. This is a nonstandard practice and will be discouraged in future versions of VRML.

Extensibility

Throughout this chapter, we've mentioned some things that are likely to change in going from VRML 1.0 to VRML 1.1; more changes can be expected in the future. Some of those changes will be described in later chapters; many of them involve adding new types of nodes to the VRML specification.

Fortunately, VRML has a way of handling these kinds of changes, as well as adding features that are needed for a particular application but which aren't in the VRML specification itself.

To create a new node type, you simply make up a name that isn't already the name of a standard node. You then use two special fields to tell the parser how to read the node; the first is the "isA" field, which says what kind of node this should be treated as if the parser doesn't recognize it, and the second is a "fields" field that tells the parser the data type of each field.

For example, let's say we wanted to create a new type of node called a `TaperedCylinder`. The `TaperedCylinder` is just like a regular `Cylinder`, except that it gets narrower towards one end (sort of like a cone with the top chopped off). To create a `TaperedCylinder`, you would do the following:

```
TaperedCylinder {
```

```
isA [ "Cylinder" ]  
fields [ SFFloat topRadius ]  
radius 7  
topRadius 2  
height 2  
}
```

The isA field has a series of strings as its value; in this case, there happens to be just one. The strings are the names of other node types, and the new TaperedCylinder node type is a "special case" of one of those; if the browser doesn't recognize a TaperedCylinder, it goes through the strings listed in the isA field until it finds one it recognizes as a valid node type. Once you can put this type of node into a VRML file, you can then generate a browser that knows how to generate a TaperedCylinder.

A regular VRML browser would look at this node and realize that it doesn't recognize TaperedCylinder as a node type. It would then notice the isA field and decide that this is actually a type of Cylinder. It would ignore the topRadius of course, but at least you'd get something reasonable (in this case, an ordinary cylinder).

The "fields" field has a list of field specifiers, each of which gives the type of data (in this case, SFFloat) and the name of the field. This lets VRML browsers parse fields they don't normally recognize. "SFFloat" just means a floating point number; a complete list of all the field types that VRML knows about can be found in the VRML specification in Appendix B.

Future versions of the VRML spec will advise that node type names be "tagged" for the sake of uniqueness. One way of doing this is to add a suffix to the end of the node type name consisting of your domain name with the periods changed to underscores. For instance, TaperedCylinder_sunee_uwaterloo_ca might be used as the node type name in the example above.

From Here...

In this chapter, we've learned how to create simple VRML files. We've discussed all the shape nodes (Sphere, Cone, Cylinder, Cube, AsciiText, IndexedFaceSet, IndexedLineSet, and PointSet) as well as the Coordinate3, FontStyle, and ShapeHints nodes. We've seen how Separators contain objects, and how they're used to create hierarchies. We've looked at how the Info node is used. We've also examined the transformation nodes (Transform, Scale, Rotation, Translation, and MatrixTransform).

At this point, you should be able to start creating and viewing your own virtual worlds. There's lots more still to learn, but you're on your way!

From here you should move on to [Chapter 14, "Adding Realism to Your VRML Environment,"](#) to learn about lightning, materials, and textures.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



12-Putting Up Walls

Now that you have the basics of VRML programming, we can look at how you can apply them to create your own environments. As is implied by the title, this chapter explores ways to define your virtual space. You can use your layout to guide users through your environment. You can also use it to help establish its underlying philosophy.

And just as you need tools in the real world, you will need tools to build your room in the virtual world. But instead of hammers, saws, and nails, you will use the various primitive shapes, plus the Coordinate3, IndexedFaceSet, and Transform nodes. And if you build something that doesn't suit you, you can change it.

This chapter focuses on different methods for creating walls and other ways to define your VRML environment. As we go through a number of examples, you will be able to come up with ways to adapt what you learn in this chapter to other applications. We will discuss the following areas:

- Using the Coordinate3, IndexedFaceSet, and IndexedLineSet nodes
- Performing transforms on objects
- Creating odd-shaped spaces

Defining Your VRML Environment

As noted in [Chapter 10 \("Laying the Foundation: Planning Your VRML Environment"\)](#), a key element in creating a virtual space is deciding what you want it to do. What is the first impression you want to give to users coming to your environment for the first time? How big do you want to make it? And most important, how are you going to define those parameters?

Since we have already discussed the first two questions in earlier chapters, let's focus on defining parameters. To begin, we need to think about how the purpose of our VRML environment is going to influence our layout. We can put everything in a single space so that users can choose for themselves how to access the material. This provides the user with the greatest degree of freedom with which to explore our VRML environment's themes.

However, we may want to have more control over how users will interact with our virtual world. This

might be achieved by separating different elements into different rooms. Each room can correspond to particular themes and/or subjects. But if we want to create more than one room, we have to consider how we are going to connect them. In some cases, we may want to connect the rooms in a particular sequence. For example if you are using your VRML environment as a tutorial that requires a linear progression, you will probably want to set up your rooms in a row with users starting at one end or the other. However, you may feel that you want to have users starting at a central point and then choosing for themselves what area they would like to try next. This layout is useful in cases where you have a central idea that is to be explored in different ways.

Figure 12.1 shows the different layouts we've discussed so far.

[Fig. 12.1](#)

Layout options for a virtual environment: (a) A single room, (b) A linear layout, and (c) A layout using a central space to connect the other rooms.

Thinking About the Third Dimension

Our planning so far has been based on a two-dimensional plane: forward and backward, right and left. But a VRML environment is a three-dimensional space and we should not ignore its potential. Rather than using walls to define the parameters of our virtual world, we can think about using different levels. Moreover, thinking in three dimensions can show us how we can vary the sizes of our walls to help create a particular effect we want our worlds to make.

Using Levels

Rather than splitting your environment using walls, you can divide it into different levels. Different levels force users to explore your 3D world in a 3D fashion: they have to go forward and backward, left and right, and up and down to navigate your virtual space. Moreover, each level can have its own walls or it can be left as an open space. In some cases, you will find that using levels can be more effective than using walls. You are able to look at all the objects in the environment and see a clear division in their layout. This setup is useful when you want users to have immediate access to everything in your environment but also want to influence how they think about it. Figure 12.2 outlines how such a scheme could work.

[Fig. 12.2](#)

Using different levels to divide a space.

Using VRML to create such a space can be fairly simple. Figure 12.3 shows a VRML environment that uses different levels to define its space. It places the same three objects on three platforms. The platforms are positioned in different areas and at different levels. The following is a listing of the VRML code used to create it:

Listing 12.1

#VRML V1.0 ascii

Separator {

Separator {

Material {

ambientColor 1 1 1

}

DEF Floor Cube {

width 5

height 0.2

depth 5

}

}

DEF Objects Separator {

Separator {

Translation {

translation 0 1 0

}

Material {

ambientColor 0 1 0

}

Cone {

parts ALL

bottomRadius 1

height 2}

}

}

Separator {

Translation {

translation -1 0.5 -1

}

Material {

ambientColor 0 1 0

}

Cube {

width 1

height 1

depth 1

}

}

Separator {

Translation {

translation 1 0.5 -2

}

Material {

ambientColor 0 1 0

}

Sphere {

radius .5

}

}

}

Separator {

Transform {

translation -2.5 4.5 2.5

scaleFactor .75 1 .75

}

Material {

ambientColor 1 1 1

```
}  
  
USE Floor  
  
}  
  
Separator {  
  
Translation {  
  
translation -2.5 4.5 3  
  
}  
  
USE Objects  
  
}  
  
Separator {  
  
Transform {  
  
translation 3 2.5 -2  
  
rotation 0 1 0 .75  
  
scaleFactor 1 1 1  
  
}  
  
USE Objects  
  
}  
  
Separator {  
  
Transform {  
  
translation 2.5 2.5 -2.5  
  
rotation 0 1 0 .75  
  
scaleFactor .75 1 .75  
  
}  
  
Material {  
  
ambientColor 1 1 1  
  
}  
  
USE Floor  
  
}
```

```
}  
.
```

Troubleshooting

Losing track of your brackets?

As you construct your environment, it becomes easy to lose track of the brackets enclosing your VRML nodes and their children. If you discover that your browser doesn't render your environment or only shows a portion of it, look through your .wrl file to make sure that there are a matching number of left and right brackets.

If you find that there is a difference, you are either missing a bracket or have inserted an extra one.

[Fig. 12.3](#)

A VRML environment using different levels to divide a space.

Incorporating different levels is one thing; what to use them for is another. You could make each level represent a particular category or theme, or a level may represent a module in a multimedia tutorial. Once behaviors become integrated with the VRML specification, you will be able to use different levels to represent categories of output or different data sets. The possibilities are limited only by your imagination.

Thinking About Your Walls in 3D

As well as thinking about using different levels in a virtual space, we also need to think about how to use walls in a 3D space to create different kinds of atmosphere. Architects and artists know how different sizes can create different impressions for the viewer. The builders of cathedrals knew that height could create a sense of awe. Using thick walls can convey a sense of strength or foreboding.

We can use the same thinking in designing a VRML environment. We can start with three simple walls. If we set them at a medium height, we can create a rather cozy, friendly space (see Fig. 12.4). The following is the VRML code used to create this space:

Listing 12.2

#VRML V1.0 ascii

Separator {

Separator {

Material {

ambientColor 0 1 0

}

Coordinate3 {

point [

-5 0 -5,

-5 0 5,

5 0 -5,

5 0 5

]

}

IndexedLineSet {

coordIndex [

0, 1, 3, 2, -1

]

}

}

Separator {

Transform {

translation 0 2.5 -3.5

}

Material {

ambientColor 0 1 1

}

DEF Wall Cube {

width 5

height 5

depth 0.2

}

}

Separator {

Transform {

translation 3.5 2.5 0

```

rotation 0 1 0 1.57
}
Material {
ambientColor 0 1 1
}
USE Wall
}
Separator {
Transform {
translation 3.5 2.5 0
rotation 0 1 0 1.57
}
Material {
ambientColor 0 1 1
}
USE Wall
}
}

```

Keeping the walls detached prevents users from feeling that they are trapped. This sense of openness is enhanced by the fact that they can see the entire wall. The walls appear to be rather thin and there is no ceiling to enclose the users. You do not get the impression that the walls are trying to keep something out or to keep you in.

Fig. 12.4

A VRML environment using medium-sized walls.

In contrast, let's see what happens when we double the height of the walls (see Fig. 12.5). The following is a listing of the VRML code.

Listing 12.3

```

#VRML V1.0 ascii
Separator {
Separator {

```

Material {

ambientColor 0 1 0

}

Coordinate3 {

point [

-5 0 -5,

-5 0 5,

5 0 -5,

5 0 5

]

}

IndexedLineSet {

coordIndex [

0, 1, 3, 2, -1

]

}

}

Separator {

Transform {

translation 0 5 -3.5

}

Material {

ambientColor 0 1 1

}

DEF Wall Cube {width 5 height 10 depth 0.2}

}

Separator {

Transform {

translation -3.5 5 0

rotation 0 1 0 1.57

}

Material {

ambientColor 0 1 1

}

USE Wall

}

Separator {

Transform {

translation 3.5 5 0

rotation 0 1 0 1.57

}

Material {

ambientColor 0 1 1

}

USE Wall

}

}

Unlike the environment in Figure 12.4, we cannot easily determine the height of the walls in this new world. At first glance, one gets a sense of being enclosed.

However, when we look up, we can see that there is no ceiling to enclose us (see Fig. 12.6). At the same time, having to look up gives users the impression that they are rather small in comparison to the environment. We can enhance this impression by starting users in a smaller room like the one in Figure 12.4 and then having them move into the larger one.

[Fig. 12.5](#)

A VRML environment using high walls.

[Fig. 12.6](#)

Looking up at the high walls.

Increasing height may create a sense of awe for our VRML environments, but increasing depth can make

them seem more imposing. If we take the walls in Figure 12.4 and increase the Cube node's depth field from 0.2 to 3, the difference becomes striking (see Fig. 12.7). Despite the fact that we can see the top of the walls, there is a certain menace in the way they look. These are objects that look as if they are meant to keep something out or something in. You could even feel that they are threatening. If we added a texture map of large granite blocks, these walls would seem unassailable.

Fig. 12.7

A VRML environment using thick walls.

Of course, we can adjust the height, width and depth of our walls to any size we want. We can even combine different sizes within the same space. But whatever we choose to do with our walls will have a direct impact on how users will react to our VRML environment.

Once you have thought about the layout of your VRML environment, you can start thinking about actually building your walls, floors, and ceiling. There are two ways of creating these objects. One is to make them out of polygons. The other method is to make them using the primitive shape nodes.

Using Polygons

Creating walls, floors, and ceilings out of polygons is relatively simple. There are two things you must do. First, you need to determine the coordinates of your polygon's corner points using the Cartesian coordinate system. You then need to decide how the points are to be connected. Once you know this, you use the Coordinate3 node with either the IndexedFaceSet or the IndexedLineSet node to create your polygons.

Polygon Basics

The VRML 1.0 specification states that the Coordinate3 node defines the set of 3D coordinates that can be used by the IndexedFaceSet, the IndexedLineSet, or the PointSet nodes. By itself, the Coordinate3 node does not create any actual shapes. It simply gives the vertex coordinates that will be referenced by the other nodes. It is defined as:

```
Coordinate3 {  
  point 0 0 0  
}
```

The point field lists the different x, y, z coordinates in the polygon. Coordinates are separated by spaces and each set is separated by a comma.

We'll begin with a simple, flat polygon: a square. If the center of our square is also the center of our Cartesian space, the center's coordinates are 0 0 0. Remember that even though we are creating a 2D polygon, we are doing so in a 3D environment. Therefore, we still have to include the x, y, and z coordinates. Let's make our square 2 units on each side, and make it lie on the ground: that is, we will use it to create a floor for our environment. One corner would then be at -1 0 -1, another at 1 0 -1, another at 1 0 1, and the last would be at -1 0 1. Using the Coordinate3 node, we would write this down as

```
Coordinate3 {
```

```
point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1]
```

```
}
```

.

It is important to note that the Coordinate3 node doesn't care if you keep the point coordinates on the same line or put them into separate lines like this:

```
Coordinate3 {
```

```
point [
```

```
-1 0 -1,
```

```
1 0 -1,
```

```
1 0 1,
```

```
-1 0 1
```

```
]
```

```
}
```

As long as you keep the x, y, and z coordinates together separated with only white space and separate the points with commas, any VRML browser will accept it.

Technically, the order in which the points are listed does not matter. When you go to convert them into a polygon, you will use an index of their positions rather than the coordinates themselves. However, it is far easier to keep track of how your coordinates are related to each other if you follow a consistent plan. For example, in the case of the square, I have started with the rear left corner point and have moved clockwise. In the case of a 3D shape such as a cube, you can think of it as a number of different flat polygons put together. You can record the coordinates of one face and then go on to record those of the next face systematically.

.

Troubleshooting

VRML is case-sensitive.

If you find that your VRML files are not working, check to see that you have typed the node name exactly as it appears in the specification. The VRML language is treated as case-sensitive by the various browsers. Therefore, typing Indexedfaceset is not the same as typing IndexedFaceSet. VRML browsers will only recognize the latter.

Once we have established the coordinates of our polygon, we can define how it is to be constructed. To do this, we can use either the IndexedFaceSet or the IndexedLineSet nodes. Both of these nodes take the vertices from the Coordinate3 node and use them to create a shape. The IndexedFaceSet node creates faces (polygons); the IndexedLineSet node connects the points with lines without filling the faces with a color or texture.

Both the IndexedFaceSet and the IndexedLineSet nodes use an index of the coordinates to perform their functions. This means that instead of using the coordinates themselves, these nodes store the position of the points in the Coordinate3 node. The first vertex in the list is assigned index number zero (0), the next vertex is assigned the number one (1), and so on. In the case of our square, the points have the following index numbers:

```
Coordinate3 {  
point [  
-1 0 -1, #point 0  
1 0 -1, #point 1  
1 0 1, #point 2  
-1 0 1 #point 3  
]  
}
```

With these index numbers, we can specify the vertices of our polygon using the coordIndex field. To create our square (see Fig. 12.8), we can use the sequence "0, 1, 2, 3, -1." Remember that VRML uses -1 to indicate the end of one face of the polygon (see "IndexedFaceSet," Ch. 11). This means that 0 is connected to 1, 1 is connected to 2, 2 is connected to 3, and 3 is the last point in this face. We could also use the sequence "3, 0, 1, 2, -1" to achieve the same effect. We simply have to make sure that our list follows the order in which the points are to be connected. With this sequence, we can write the IndexedLineSet node as follows:

```
IndexedFaceSet {  
coordIndex [  
0, 1, 2, 3, -1  
]  
}
```

This will create the following image:

[Fig. 12.8](#)

A square created using the IndexedFaceSet.

We can also use the IndexedLineSet to create a square that is not filled (see Fig. 12.9). According to the VRML 1.0 specification, this node "represents a 3D shape formed by constructing polylines from vertices located at the current coordinates." In other words, the IndexedLineSet node takes the vertices in the Coordinate3 node and draws lines to connect them. This means that, unlike the IndexedFaceSet node, you are able to look through a polygon constructed with the IndexedLineSet node. The IndexedLineSet node is

defined as:

```
IndexedLineSet {  
coordIndex 0  
materialIndex -1  
normalIndex -1  
textureCoordIndex -1  
}
```

A normal is a vector that is perpendicular to the surface of an object. It is used to calculate the effects of light sources on a scene. See Chapter 14 for details.

Using the same `Coordinate3` setup as for our `IndexedFaceSet` square, we construct our `IndexedLineSet` square as follows:

Listing 12.4

```
Coordinate3 {  
point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1]  
}  
IndexedLineSet {  
coordIndex [  
0, 1, 2, 3, -1  
]  
}
```

This will produce the following image:

[**Fig. 12.9**](#)

A square created using the `IndexedLineSet`.

As well as modeling our square as a single polygon, we can use the same coordinate point set to create one or two right triangles. To create a single triangle (see Fig. 12.10), simply index three of the points instead of all four:

Listing 12.5

```
Coordinate3 {  
point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1]
```

```

}
IndexedFaceSet {
  coordIndex [
    0, 2, 3, -1
  ]
}

```

This will produce the following image:

Fig. 12.10

A right triangle rendered using the IndexedFaceSet.

We can also render a square floor by indexing two right triangles using the same point set (see Fig. 12.11). Moreover, we can use different colors for each section by assigning a separate Material node for each triangle:

Listing 12.6

```

#VRML V1.0 ascii

Separator {Separator {
  Coordinate3 {
    point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1]
  }
  Separator {
    Material {ambientColor 1 1 1}
    IndexedFaceSet {
      coordIndex [0, 2, 3, -1]
    }
  }
  Separator {
    Material {ambientColor 1 0 0}
    IndexedFaceSet {
      coordIndex [0, 1, 2, -1]
    }
  }
}

```

```
}  
  
}  
  
}
```

This produces a two-colored square:

[**Fig. 12.11**](#)

A two-colored square rendered using the IndexedFaceSet.

From 2D to 3D Polygons

Now that we have the basic tools for creating a polygon, we can shift from making a floor out of a simple 2D polygon to creating a room out of a 3D one. For our example, we will make a cube using the `Coordinate3` and `IndexedFaceSet` nodes rather than a `Cube` primitive. This will let us select the faces we want to render rather than having to show the entire cube.

We start by considering how adding the third dimension affects our coordinates and thinking about our room in terms of Cartesian space. First, we know that a cube has eight corners. Therefore, we will need to have eight coordinates. If we take the origin of the Cartesian coordinate system (0 0 0) as our center point, we can create a cube 2 units in height, length and depth by setting all our coordinates one unit from the origin. Figure 12.12 shows such a cube and its coordinates in a Cartesian space.

[**Fig. 12.12**](#)

A cube and its coordinates in Cartesian space.

We can then use these coordinates to set up our `Coordinate3` node:

```
Coordinate3 {  
  
point [  
  
-1 1 1, #point 0  
  
1 1 1, #point 1  
  
1 -1 1, #point 2  
  
-1 -1 1, #point 3  
  
-1 1 -1, #point 4  
  
1 1 -1, #point 5  
  
1 -1 -1, #point 6  
  
-1 -1 -1 #point 7  
  
]  
  
}
```

```
}
```

We now have to turn these coordinates into the faces of our cube. We can visualize the cube as eight points connected by lines (see Fig. 12.13). The first point corresponds to point 0 in our point set (-1 1 1); the next point corresponds to point 1 in our set (1 1 1); and so on. Eventually, we come up with a picture of our cube with the point indices:

[**Fig. 12.13**](#)

A cube and the point indices.

Using this picture, we can plot out the coordIndex field for our cube:

```
IndexedFaceSet {  
  coordIndex [  
    0, 4, 5, 1, -1, #The Ceiling  
    3, 7, 6, 2, -1, #The Floor  
    0, 4, 7, 3, -1, #The West Wall  
    1, 5, 6, 2, -1, #The East Wall  
    0, 1, 2, 3, -1, #The Front Wall  
    4, 5, 6, 7, -1 #The Back Wall  
  ]  
}
```

This code will give us a solid cube (see Fig. 12.14). We can replace the IndexedFaceSet with the IndexedLineSet node to get a better idea of our cube's construction:

[**Fig. 12.14**](#)

A cube created with the IndexedLineSet node.

You can see that all the sides of our cube have been rendered. However, we can choose to use only certain parts of our cube. Let's say that we only want to use the ceiling, floor, the west wall, and the back wall. We would rewrite the IndexedFaceSet node as:

```
IndexedFaceSet {  
  coordIndex [  
    0, 4, 5, 1, -1, #The Ceiling  
    3, 7, 6, 2, -1, #The Floor  
    0, 4, 7, 3, -1 , #The West Wall
```


4, 5, 6, 7, -1 #The Back Wall

1

}

This would give us the image in Figure 12.15.

Starting with VRML 1.1, the default settings on the ShapeHints node will be changed. To be on the safe side, put in an explicit ShapeHints { vertexOrdering UNKNOWN_ORDERING } to guarantee that your walls will be double-sided.

Fig. 12.15

An IndexedFaceSet cube with only the ceiling, floor, west wall, and back wall.

We can add or remove different faces to create the type of room that we want. We can also choose to create only sections of a polygon's face. Instead of rendering a square face on our cube, we can render a right triangle by using three of the four coordinates of one side. If we wanted to show the east wall as a right triangle, we would use the following (the right triangle's coordinates are italicized):

IndexedFaceSet {

coordIndex [

0, 4, 5, 1, -1, #The Ceiling

3, 7, 6, 2, -1, #The Floor

5, 2, 6, -1, #The East Wall

4, 5, 6, 7, -1 #The Back Wall

1

}

The result is shown in Figure 12.16.

You can connect any coordinates as long as they are on the same plane. For example, we could render a slanted wall by using the coordinates 1, 3, 7, and 5. However, you cannot connect 0, 4, 7, and 2. If you are not sure if the points you want to connect are on the same plane, map out your coordinates on a piece of paper and draw a line connecting the ones you want to use. If the result cannot be viewed as a 2D polygon, the points are not on the same plane.

Fig. 12.16

A cube using a right triangle for the east wall.

Defining VRML Spaces with Primitives

Although we can use polygons to define our VRML environments, you will find that using the other shape nodes are a more convenient way of constructing walls. Rather than having to plot out a series of coordinates, you can set the size of your walls by the more familiar parameters of height, width, and depth. Moreover, the size of your file can be greatly reduced by using these primitives.

Making a Wall Using the Cube Node

To create a room using the Cube node, you need to define the size of your room. You also must decide whether you want to have a floor and/or ceiling, and whether you want to join your walls or to separate them from each other.

If we wanted to make a wall that is 5 units wide, 4 units high, and 0.2 units thick, we would use the Cube node in the following way:

```
Cube {  
width 5  
height 4  
depth 0.2  
}
```

The thickness of your walls can be adjusted to suit your needs by changing the value of the depth field. Figure 12.17 shows various walls with depths ranging from 0.1 to 1.0 units:

[Fig. 12.17](#)

Cube nodes with a variety of depth values. Top row (left to right): 0.1, 0.2, 0.3, 0.4, and 0.5. Bottom row (left to right): 0.6, 0.7, 0.8, 0.9, and 1.0.

You need to remember that your wall's center point will be 0 0 0. Therefore, the distance of the edges of your wall from the center of your environment are set at one-half of its width, height, and depth values. This will become important when we begin moving and connecting our walls, floors, and ceilings to build a room.

Creating a wall using the Cube node is only the first step in defining a VRML space. Once you have your wall, you will need to create others and position them in your virtual space. The keys to accomplishing these tasks are DEF, USE, and the Transform node.

Using DEF and USE to Create More Walls

While we can use a separate Cube node to create each wall, floor, and ceiling in our VRML space, it is also possible to use the same Cube node and then adjust it using the Transform node. To do this, we must assign a name to our first Cube node. For example, we can define nodes such as Wall and Floor:

```
DEF Wall Cube {
```

```
width 5
```

```
height 4
```

```
depth 0.2
```

```
}
```

```
DEF Floor Cube {
```

```
width 9
```

```
height 0.2
```

```
depth 9
```

```
}
```

We can then reuse these nodes by using the USE keyword to call them up:

```
Separator {
```

```
USE Wall
```

```
}
```

```
Separator {
```

```
USE Floor
```

```
}
```

```
.
```

You may be wondering why we don't just use the Cube node to create a large cube that can act as our room. Unfortunately, the primitive shape nodes are designed to create objects with only one face. This means that when you go inside a shape, it will disappear. Try entering a large cube created with the Cube node. The moment you pass through the outer face, the cube will disappear.

Using the Transform Node

In Chapter 11, "VRML Standards and Practices," Bernie went over the Transform node. We will now look at how we can use it to position our walls to where we want them. We will look at how we can use rotations to set a wall to the desired angle, translations to position a wall in our environment, and scaling to adjust the size of a wall instead of using another Cube node.

Translating and Rotating Your Walls to Make a Room

We can now begin constructing a room using the named nodes we have just created. We can begin by moving our first wall using the Transform node. As noted earlier, the center point of our wall using the Cube node is at 0 0 0. However, we want to make that point the center of our virtual space. Therefore, we

need to move the first wall back.

Before we begin translating the wall to a new position, we need to think about whether or not we want our walls, floor and ceiling joined or separated from each other. This decision will dictate the values we will use to translate these objects from their original position. If we want to join the objects, we need to move a wall by one-half the width of the adjoining object less the former's depth. For example, we have decided to use the same dimension for all our walls. To move the rear wall into the proper position, we must shift it along the z-axis by 2.4 units ($(\text{width}/2 - \text{depth}/2)$ or $(5/2 - 0.2/2)$). Moreover, since we want to move it to the rear, we have to give this setting a negative value. The Transform node for this action would look like this:

Separator {

Transform {

translation 0 0 -2.4

}

DEF Wall Cube {

width 5

height 4

depth 0.2

}

}

Deciding which axis or axes to use when translating an object is fairly straightforward. Moving it left or right involves the x-axis. Moving it up or down uses the y-axis. And moving it forward or back uses the z-axis. When you need to determine whether a value needs to be negative or positive, it is easier to think in terms of the direction you are moving it. If you want to move it to the right, forward, or up, then its value will be positive. If you want to move it to the left, back, or down, then its value will be negative. "Forward" in this case means "towards the user."

You can also use the Translation node to perform the same function. We can move our floor by moving it along the y-axis. Since our walls are 4 units high and the floor is 0.2 units in height, we need to move it 1.9 units down. The Translation node for this would be:

Translation {

translation 0 -1.9 0

}

DEF Floor Cube {

width 5

height 0.2

depth 5

}

.

You may have already realized this, but the Scale, Rotation, and Translation nodes are really just a special case of the Transform node.

Now that we have put our rear wall and floor in place, we can begin adding the other walls and a ceiling. Adding a ceiling is simple (see Fig. 12.18). We simply call up the floor and use the Transform node to place it at the top edge of the rear wall:

Separator {

Transform {

translation 0 1.9 0

}

Use Floor

}

This produces the following environment:

[Fig. 12.18](#)

A ceiling, floor and rear wall created using the Cube node.

We can also use the Transform node to create our east and west walls. As we have already seen, you can rotate an object using the Rotation node or the Transform node. You must specify the axis and the angle in radians (see Fig. 12.19). You can use both positive and negative values for the number of radians to control the direction of the change. A quick guide for thinking about radians is to remember that 45 degrees is equal to .7854 radians, 90 degrees is 1.5708 radians, and 180 degrees is 3.1416 radians. Figure 12.18 shows an arrow adjusted at both .7854 and -.7854 radians (or 45 and -45 degrees) on different axes.

[Fig. 12.19](#)

An arrow rotated at .7854 and -.7854 radians on different axes. Top row: no rotation. Middle row (left to right): .7854 radians on the x-axis, on the y-axis, and on the z-axis. Bottom row (left to right): -.7854 radians on the x-axis, on the y-axis, and on the z-axis.

In the case of a wall, you are more likely to rotate it on the y-axis. But you may want to adjust its angle using the x- and/or z-axis to create a certain effect. If we wanted to rotate our wall 90 degrees on the y-axis, we would use the Transform node as follows:

Separator {

Transform {

rotation 0 1 0 1.5708

}

USE Wall

}

Now we can add the translation field to move our wall into position. To move it into the west wall position, we need to move it -2.4 units ($5/2 \times 0.2/2$) along the x-axis:

Separator {

Transform {

rotation 0 1 0 1.5708

translation -2.4 0 0

}

USE Wall

}

.

Remember that the order in which the fields appear inside the Transform is irrelevant; the scaling is always done first, followed by the rotation, followed by the translation.

If we include the east wall as well, the full listing of our VRML environment would look like the following:

Listing 12.7

#VRML V1.0 ascii

Separator {

Separator {

Transform {

translation 0 0 -2.4

}

DEF Wall Cube {

width 5

height 4

depth 0.2

```
}  
}  
Separator {  
Translation {  
translation 0 -1.9 0  
}  
DEF Floor Cube {  
width 5  
height 0.2  
depth 5  
}  
}  
Separator {  
Transform {  
translation 0 1.9 0  
}  
USE Floor  
}  
Separator {  
Transform {  
rotation 0 1 0 1.5708  
translation -2.4 0 0  
}  
USE Wall  
}  
Separator {  
Transform {  
rotation 0 1 0 1.5708
```

translation 2.4 0 0

}

USE Wall

}

}

This will produce a room as shown in Figure 12.20.

Fig. 12.20

A room constructed using rotations and translations.

We can also adjust the size of a wall by using the Transform node's scaleFactor field. For example, instead of creating a separate Cube node for our named node called Floor, we could scale the node labeled Wall, rotate it 90 degrees on the x-axis, and then translate it. We begin by determining how much we need to scale Wall. The dimensions of Floor are width 5, height 0.2, and depth 5; the dimensions of Wall are width 5, height 4, and depth 0.2. Since we can rotate Wall, we only need to adjust the height.

To get the scaling factor, we divide the destination size by the current size. In the case of the wall, we divide 4 into 5 to get 1.25. This is our scaling value. Since we do not want to change the width (x-scale) or depth (z-scale), we set their values at 1. We then replace the floor and ceiling portions of our code with the following:

Separator {

Transform {

translation 0 -1.9 0

rotation 1 0 0 1.5708

scaleFactor 1 1.25 1

}

USE Wall

}

Separator {

Transform {

translation 0 1.9 0

rotation 1 0 0 1.5708

scaleFactor 1 1.25 1

}

USE Wall

}

This replacement code, along with the code for the three walls, will produce exactly the same model as in Figure 12.20.

If you want to construct a room in which the walls, floor, and ceiling are not joined (see Fig. 12.21), you simply have to increase the translation value. For example, we can increase the value of each translation by 1 unit to create the following:

[Fig. 12.21](#)

The same room constructed by increasing the translation values by one unit.

Following is a listing of the code used to create Figure 12.21:

Listing 12.8

#VRML V1.0 ascii

Separator {

Separator {

Transform {

translation 0 0 -3.4

}

DEF Wall Cube {

width 5

height 4

depth 0.2

}

}

Separator {

Translation {

translation 0 -2.9 0

}

DEF Floor Cube {

width 5

```

height 0.2
depth 5
}
}
Separator {
Transform {
translation 0 2.9 0
}
USE Floor
}
Separator {
Transform {
rotation 0 1 0 1.5708
translation -3.4 0 0
}
USE Wall
}
Separator {
Transform {
rotation 0 1 0 1.5708
translation 3.4 0 0
}
USE Wall
}
}

```

Using the Cylinder Node Instead of the Cube Node

There may be times when you want to define your virtual space with columns instead of walls. This setup provides a more open space while still establishing definite boundaries. We can use the same floor and ceiling created for Figure 12.20 and use the Cylinder node to create the columns.

We start by designating our cylinder node as Column, and setting the height as 4 and the radius as .5:

```
DEF Column Cylinder {  
parts ALL #The Default Value  
radius .5  
height 4  
}
```

This node will create a cylinder at the center of our room. We must then move it to the desired location using the Translation node. We'll put it in the left front corner. This means we must move the cylinder half the value of the depth less the radius of the cylinder, and half the value of the width less the radius of the cylinder. (Remember that in the Cylinder node, we use the radius and not the diameter. Therefore, we do not have to divide the radius in half as we do with the values of the Cube node.) Thus we get $2.5 - 0.5$, or 2 for both the x- and the z-axes. We then write the Translation as

```
Translation {  
translation -2 0 2  
}
```

Repeating this procedure for the other columns, we end up with the following code:

Listing 12.9

```
#VRML V1.0 ascii
```

```
Separator {
```

```
Separator {
```

```
Translation {
```

```
translation 0 -1.9 0
```

```
}
```

```
DEF Floor Cube {
```

```
width 5
```

```
height 0.2
```

```
depth 5
```

```
}
```

```
}
```

```
Separator {
```

Transform {
translation 0 1.9 0
}

USE Floor
}

Separator {
Translation {
translation -2 0 2
}

DEF Column Cylinder {
parts ALL
radius .5
height 4
}
}

Separator {
Translation {
translation -2 0 -2
}

USE Column
}

Separator {
Translation {
translation 2 0 -2
}

USE Column
}

Separator {

```
Translation {  
translation 2 0 2  
}
```

```
USE Column  
}  
}
```

This code will render a space consisting of a ceiling, floor, and column at each of the corners (see Fig. 12.22).

[Fig. 12.22](#)

A room constructed using the Cylinder and Cube nodes.

Creating Odd-Shaped Rooms

The first thing that needs to be said about creating odd-shaped walls and rooms using VRML is that this task is usually best left for world-building software (see "An Overview of Worldbuilding Toolkits," Ch. 25). Trying to plot out a room using unusual angles or curved walls is difficult when trying to program a VRML environment by hand. However, it can be done if you are patient and willing to compromise in its construction.

An easy way of creating a curved wall is to use the Cylinder node. Of course, you may not want to use the entire cylinder in your environment, but there is no way to remove any of its segments. We can, however, adjust its curve by scaling it using the Transform node. The following shows a VRML environment that uses a scaled Cylinder node. Figure 12.23 is followed by the code used to create this environment.

[Fig. 12.23](#)

A room constructed using a scaled Cylinder node and Cube nodes.

Listing 12.10

```
#VRML V1.0 ascii
```

```
Separator {
```

```
Separator {
```

```
Transform {
```

```
translation -4 0 -4
```

```
rotation 0 1 0 .7854
```

```
scaleFactor 1 1 0.5
```

scaleOrientation 0 0 1 .7854

}

Material {ambientColor 0 1 0}

Cylinder {

parts ALL

radius 4

height 4

}

}

Separator {

Transform {

translation 2.3 0 -6.7

}

Material {

ambientColor 0 1 1

}

Cube {

width 7

height 4

depth 0.2

}

}

Separator {

Transform {

translation -4.5 0 2.5

rotation 0 1 0 2.1

}

Material {

ambientColor 0 1 1

}

Cube {

width 10

height 4

depth 0.2

}

}

Separator {

Translation {

translation 3 0 6.9

}

Material {

ambientColor 0 1 1

}

Cube {

width 10

height 4

depth 0.2

}

}

}

As you look through the code, you will notice that many of the translation fields use values with origins that are not immediately apparent. For example, the short wall connected to the east side of the cylinder uses a translation value of 2.3 on the x-axis and -6.7 on the z-axis. The way that I arrived at this value was to estimate the cylinder's position before scaling it. With a translation value of -4 0 -4 and a radius of 4, I figured that the joint could be made at 0 0 -4. I then translated the cube according to this estimate (3.5 0 -6).

However, when I viewed the environment in a VRML browser, I saw that my estimate was way off. I had to make several adjustments before the two objects were joined. In fact, creating this environment took several tries because some of the changes were made using 0.1 unit increments.

The key to creating odd-shaped rooms by hand is experimentation. You can always approximate where different objects will be joined, but you must be aware that using different angles and curves will require minor adjustments after you first code it up. The benefit is that you will have created an interesting layout using a minimum of file space.

From Here...

Building walls, ceilings, and floors is only a preliminary step in creating your VRML environment. You will also need to think about adding things for users to look at and explore. You can also produce a greater element of realism by adding lights and texture maps.

To learn about how to add to your VRML environment, see the following:

- [Chapter 13, "Furnishing the Room,"](#) demonstrates how you can create objects for your virtual world. It goes through the steps for building a table and chair, and it discusses ways that you can begin designing your own objects. It also points to Internet sites where you can download objects that you can incorporate into your VRML world.
- [Chapter 14, "Adding Realism to Your VRML Environment,"](#) outlines the process of using lighting, materials, and texture maps to add more realism to your VRML world. It goes over the various lighting and texture nodes.
- [Chapter 15, "Adding Links to Your VRML Environment,"](#) outlines how to use the WWWAnchor and WWWInline nodes to connect your environment to the World Wide Web.
- If you want to look at the experiences of others who have created their own VRML environments and models, see Part V, "Learning from the Pros: Sample VRML Environments."
- If you want to see what programs are available to help you create VRML environments, see Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities." These chapters discuss how these programs can help you build and furnish your VRML world.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



13-Furnishing Your Room

Once you have built your VRML space, you will want to put things into it. You may need conventional-looking furniture: maybe just a few simple chairs and a table, or you may want more futuristic-looking furnishings. You may want to put a fish-tank, a bird-cage full of flying toasters, or even an entire planet in your room. You could even have a need for a two-headed llama. Whatever your needs, you have a great deal of freedom when creating furniture and other objects in VRML. You will find that the same tools you used to create walls, floors, and ceilings in VRML can be used to make chairs, tables, and other furnishings.

This chapter focuses on different methods for creating objects with which to furnish your VRML environment. As we go through some examples, you will be able to adapt what you learn in this chapter to design your own objects. We will examine the following areas:

- Using the Shape nodes to create objects for your VRML environment
- Placing objects in your virtual space
- Using the transform node to alter your objects

Creating Objects for Your VRML Environment

Creating objects for your VRML environment uses the same techniques as building walls and floors. You can use either polygons or shapes to construct your models, and you can put the different components together using the Transform node. However, you must consider a number of factors before you begin.

The first thing you must think about is your model's position in terms of Cartesian space. Whereas the rooms we constructed in Chapter 12 used the origin (0 0 0) as their center point, we do not necessarily want to use the origin as the center point for our objects. The reason is that the transforms that we will use to position items in our VRML environments are easier if the origin is not at the center. We will need a standard reference point when we begin to import objects from other sites on the Internet using the WWWInline node (see ["Linking Your Environment to the Internet: The WWWInline and WWWAnchor Nodes," Ch. 15](#)).

The most logical approach to use in a conventional environment is to center your object on the x- and

z-axes, and to set its bottom edge on the y-axis (see Fig. 13.1). By always having the bottom edge at the zero point on the y-axis, the object defaults to sitting on the floor. The calculations for our transforms will then become much easier to handle.

[Fig. 13.1](#)

Positioning objects so they can be easily imported into a VRML environment.

For example, we can use the room we created in Chapter 12 (see [Fig. 12.18](#)). We have created a chair and have put its bottom edge on the y-axis. We know that the room is 4 units high, the floor is 0.2 units thick, and the room's center point is 0 0 0. This means that we must move the chair a distance equal to one-half the height of the room minus the thickness of the floor. This gives us a translation value of 1.8 ((4/2)-0.2). We can then use the following Translation node to shift our chair to the desired position:

```
Translation {  
translation 0 -1.8 0  
}
```

The chair is now in the correct position (see Fig. 13.2).

[Fig. 13.2](#)

A chair placed in a VRML room.

In contrast, creating the chair with its center set at 0 0 0 would make our calculation more complicated. We would first have to figure out the translation value from the room's center point to the surface of the floor. We would then have to calculate a translation value to shift the chair's bottom edge to zero on the y-axis. Then we would have to subtract the second value from the first one to come up with our final translation value.

While we can make these calculations using models we create ourselves, it is more difficult when using ones created by others. This task becomes even more difficult when referencing models from other World Wide Web sites using the WWWInline node. Determining the alignment for a VRML model that you did not create yourself is difficult enough when you have the code in front of you; trying to determine its alignment when the file is located at some remote Web site would be an extremely tedious job.

We also have to consider the size of our objects. While we can always use the Transform node to scale an object, it would be easier to work from a standard. In this way, we can import objects created by other VRML developers knowing that their size will be in proportion to the other items in our virtual space. The VRML standard says that all dimensions are in meters, and so that is the reference we will always use.

Keeping the notion of a standard alignment and dimension in mind, we can start creating some objects to see how the process works.

Making a Chair and a Table Using the Cube Node

The most efficient way of creating objects is to use the shape nodes. They require less file space than objects constructed by combining the Coordinate3, IndexedFaceSet, or IndexedLineSet nodes. It should be noted that VRML models created using the primitive shape nodes are simpler than the ones made from polygons. However, you will find that making objects using the primitive shape nodes will often be perfectly adequate.

Constructing a Chair

We can make a simple chair using the Cube node. To begin, we have to decide on its dimensions. Looking at the chairs in my apartment, I found that the legs were approximately 0.4 meters high and the depth of the seat varied from 0.4 to 0.5 meters. The height of the back and thickness of the chairs varied.

The Seat and Legs

Having these measurements, we can start by building the seat and legs. The seat will be 0.5 units wide, 0.05 units high, and 0.5 units deep. The Cube node would, therefore, look like this:

```
Separator {
```

```
Cube {
```

```
width 0.5
```

```
height 0.05
```

```
depth 0.5
```

```
}
```

```
}
```

We then have to move the seat up along the y-axis so that the bottom of the legs are on zero on the y-axis. Since our legs are 0.4 units high and the width of the seat is 0.05 units, we need to move it 0.425 units ($\text{leg\_height} + \text{seat\_height}/2$). The translation value, therefore, is

```
Translation {
```

```
translation 0 0.425 0
```

```
}
```

We can now create the legs for our chair. We'll use the DEF node to name the Cube node as Leg (not the most exciting name, but it is descriptive). After some experimentation, I found that a width and depth of 0.1 units gave the best proportion. And as noted above, the height will be 0.4 units. Thus, our Cube node looks like this:

```
Separator {
```

```
Translation {  
translation 0.2 0.2 0.2  
}
```

```
DEF Leg Cube {  
width 0.05  
height 0.4  
depth 0.05  
}  
}
```

The Translation node raises the leg on the y-axis so the bottom edge is resting at zero on the same axis (leg_height/2). It also shifts it along the x- and z-axes so that it is positioned on the right front corner of our seat. We can then add the other three legs by calling up the node labeled Leg and translating them into their proper positions:

```
Separator {  
Translation {  
translation -0.20 0.20 0.20  
}
```

```
USE Leg  
}
```

```
Separator {  
Translation {  
translation 0.20 0.20 -0.25  
}
```

```
USE Leg  
}
```

```
Separator {  
Translation {  
translation -0.20 0.20 -0.25  
}
```

USE Leg

}

This produces a stool with four legs.

Adding a Back to Our Chair

Once we have the bottom part of our chair, we can give it a back. Of course, the backs of chairs come in all sizes and shapes. We'll make ours as wide as the seat and 0.75 units in height:

Cube {

width 0.5

height .75

depth 0.05

}

We then need to align the back with the rest of the chair. Since the bottom edge of the chair is now resting on the zero point of the y-axis, we need to raise the back along the y-axis by the value of its height (0.75) plus one-half the height of the seat (.05/2 or .025). Moreover, we must push the back along the z-axis by a value equal to one-half the seat's depth (.5/2 or .25). Thus, our translation node is:

Translation {

translation 0 0.775 -.25

}

Fig. 13.3

A VRML stool.

We can now see the entire listing for our chair. Note that I have included a Material node that will color it white.

#VRML V1.0 ascii

Separator {

Material {

ambientColor 1 1 1

}

Separator {

Translation {

translation 0 0.775 -.25

}

Cube {

width 0.5

height .75

depth 0.05

}

}

Separator {

Translation {

translation 0 0.425 0

}

Cube {

width 0.5

height 0.05

depth 0.5

}

}

Separator {

Translation {

translation 0.20 0.20 0.20

}

DEF Leg Cube {

width 0.05

height 0.40

depth 0.05

}

}

Separator {

```

Translation {
translation -0.20 0.20 0.20
}

USE Leg
}

Separator {

Translation {
translation 0.20 0.20 -0.25
}

USE Leg
}

Separator {

Translation {
translation -0.20 0.20 -0.25
}

USE Leg
}
}

```

Figure 13.4 shows our chair when viewed with a VRML browser.

[Fig. 13.4](#)

A VRML chair viewed using the WebFX browser.

Constructing a Table

Making a table using the Cube node is even simpler than building a chair. Measuring the tables in my apartment, I found that the legs averaged 0.75 meters in height. For the top, we'll use a Cube node with a width and depth of 1 unit, and a height of 0.1 units.

To construct the legs, we can employ the same technique as building the legs for our chair. We'll use the DEF node to name the Cube node as "Leg." We will also use a width and depth of 0.1 units as well as a height of 0.75 units.

.

Note that you can reuse names; both the chair and the table can have a "Leg." This may or may not be

acceptable in future versions of VRML, so you might be better using "Chairleg" and "Tableleg."

We use the Translation node to raise the leg so that the bottom edge is resting at zero on the same axis (.75/2 or 0.375) and shift it along the x- and z-axes so that it is positioned on the right front corner of our seat. If we want to have the leg flush against the corner of the table top, we will need to shift the leg by one-half the width and depth of the top (1/2 or 0.5) less the width and depth of the leg (0.05). Thus, we will translate the leg 0.45 units along the x-axis, 0.375 units along the y-axis, and 0.45 units along the z-axis. As with our chair, we add the other three legs by calling up the node labeled Leg and translating them into the proper positions. The following shows the code used to create and position the legs:

Separator {

Translation {

translation 0.45 0.375 0.45

}

DEF Leg Cube {

width 0.05

height 0.75

depth 0.05

}

}

Separator {

Translation {

translation -0.45 0.375 0.45

}

USE Leg

}

Separator {

Translation {

translation -0.45 0.375 -0.45

}

USE Leg

}

Separator {


```
Translation {
```

```
translation 0.45 0.375 -0.45
```

```
}
```

```
USE Leg
```

```
}
```

```
}
```

Once you have written the first implementation of the named node, you can save some typing by using the copy and paste features of your editor to copy that part of the code. You then simply have to change the field values.

Creating and positioning the table top is an even simpler task. Using the Cube node to create the shape, we simply have to raise it along the y-axis. Since the height of our table's legs is 0.75 units, we simply have to translate the top the same distance plus one-half of the top's height (.1/2 or .05). This gives us a translation value of 0.8. The following is the code for creating and translating our table top:

```
Separator {
```

```
Translation {
```

```
translation 0 0.8 0
```

```
}
```

```
Cube {
```

```
width 1
```

```
height .1
```

```
depth 1
```

```
}
```

```
}
```

Figure 13.5 shows our table when viewed with a VRML browser.

[Fig. 13.5](#)

Our VRML table viewed using the WebFX browser.

Creating a Table Using the Cylinder Node

We can also build a table using the Cylinder node. This method has two distinct advantages. First, we can use a cylinder as the base for our table. This means that we will not have to calculate any x or z translations to position it. Second, we can use the Transform node to scale our cylindrical table to

generate more interesting shapes.

Making a Cylindrical Table

To build the base of our table, we will use the same height as the legs for our square table (0.75) and make it 0.5 units wide. This means that the radius will be 0.375 units. We will also need to raise the base to make its bottom edge rest on the zero point of the y-axis. As with the leg of our square table, this value is 0.375 ($.75/2$). The code to create the base, therefore, is as follows:

```
Separator {  
Translation {  
translation 0 .375 0  
}  
Material {  
ambientColor 0 0 1  
}  
Cylinder {  
parts ALL  
height .75  
radius .375  
}  
}
```

Creating the table top for our cylindrical table follows much the same process as the top for our square table. We can set the width of the top as 1.5 units, and so the radius will be 0.75 units. The height of the top will be 0.1 units. Since the height of our base is 0.75 units, we translate the top the same distance plus one-half of the top's height: a translation value of 0.8. The following is the code for creating and translating our table top:

```
Separator {  
Translation {  
translation 0 .75 0  
}  
Material {  
ambientColor 1 1 1
```

```

}
Cylinder {
  parts ALL
  height 0.1
  radius .75
}
}

```

Note that the color of the base is different from that of the table top. I found that when the table is rendered, using different colors made it easier to distinguish the two components. The following shows how our cylindrical table actually looks:

[Fig. 13.6](#)

Our cylindrical table viewed using WebFX.

Scaling the Table

Now that we have our table, we can use the Transform node to scale it. Scaling a cylinder does not change the number of sides that VRML uses to construct it. However, we can produce interesting shapes if we scale our table along a single axis.

To begin, we will scale our table along the z-axis by a factor of 2. We join the code for the base and the table top using a Separator node since they can be thought of as a single unit. The VRML code would be as follows:

```

#VRML V1.0 ascii

```

```

Separator {
  Transform {
    scaleFactor 1 1 2
  }
  Separator {
    Separator {
      Translation {
        translation 0 .75 0
      }
    }
    Material {

```

ambientColor 1 1 1

}

Cylinder {

parts ALL

height 0.1

radius .75

}

}

Separator {

Translation {

translation 0 .375 0

}

Material {

ambientColor 0 0 1

}

Cylinder {

parts ALL

height .75

radius .375

}

}

}

}

If we place a chair beside the table, we find that our cylindrical table has become a modern-looking desk (see Fig. 13.7). And if we increase the z value in our `scaleFactor` field to 3, we create a modern-looking conference table (see Fig. 13.8).

[Fig. 13.7](#)

Creating a modern desk using a `scaleFactor` of 2.

[Fig. 13.8](#)

Creating a modern conference table using a scaleFactor of 3.

Positioning Your Objects in a VRML Environment

Once you have built your objects, you will need to position them in your VRML environment. There are two elements you must take into consideration. The first is the object's position in relation to your virtual space. The second is the object's position in relation to other objects.

Transforming Your Objects in Relation to Your Virtual Space

There are no two ways about it: positioning your objects correctly in your VRML environment is as important to its success as the design of the environment itself. If you have everything in just the right place, then you can feel confident that what you are trying to convey through your VRML world is going to be understood. But if something doesn't look right—if something is out of place—then your ideas will be compromised.

An important step in this process is to align your object's bottom edge with the surface of your environment's floor. We'll assume that your models have their bottom edges set at the zero point of the y-axis. We then have to look at where the floor is positioned in relation to a Cartesian space and the manner in which it was constructed.

In the case of a floor built with the `Coordinate3` and `IndexedFaceSet` nodes, the thickness of the floor is not an issue. We can figure out the translation need for our object by looking at the coordinates. For example, let's use the room we created in Figure 12.15. The `Coordinate3` node for that object was:

```
Coordinate3 {  
point [  
-1 1 1, # point 0  
1 1 1, # point 1  
1 -1 1, # point 2  
-1 -1 1, # point 3  
-1 1 -1, # point 4  
1 1 -1, # point 5  
1 -1 -1, # point 6  
-1 -1 -1 # point 7  
]  
}
```

The floor in this room is set at -1 on the y-axis. Therefore, the translation factor for an object will be -1.

If we wanted to place our chair in this room, the Transform needed would be:

```
Separator{  
Transform {  
translation 0 -1 0  
}  
Separator {  
Material {  
ambientColor 1 1 1  
}  
Separator {  
Translation {  
translation 0 0.775 -.25  
}  
Cube {  
width 0.5  
height .75  
depth 0.05  
}  
}  
...  
}
```

Using this transform, our chair would be resting on the surface of the floor (see Fig. 13.9).

You may wonder why we don't just change the position of the floor instead of changing individual objects. We should consider the position of the walls, floors, and ceiling of our environments when we are creating them. However, your environment may have multiple levels, and so you will need to adjust the position of your furnishings rather than your floors.

[Fig. 13.9](#)

Positioning a chair in a room constructed with polygons.

But if we wanted to position the same chair in a room made up of cube nodes, we would have to adjust the translations according to the thickness of the floor. Let's look at how we positioned the chair in the

room we used for Figure 13.1. That room was constructed using Cube nodes. The following is the VRML code:

```
#VRML V1.0 ascii
```

```
Separator }Separator {
```

```
Transform {
```

```
translation 0 0 -2.4
```

```
}
```

```
DEF Wall Cube {
```

```
width 5
```

```
height 4
```

```
depth 0.2
```

```
}
```

```
}
```

```
Separator {
```

```
Translation {
```

```
translation 0 -1.9 0
```

```
}
```

```
DEF Floor Cube {
```

```
width 5
```

```
height 0.2
```

```
depth 5
```

```
}
```

```
}
```

```
Separator {
```

```
Transform {
```

```
translation 0 1.9 0
```

```
}
```

```
USE Floor
```

```
}
```

```
}
```

The height of the walls is 4 units and the room is centered on the origin. However, if we simply use one-half the walls' height as the value for our translation field, our chair would look as if it had melted into the floor (see Fig. 13.10).

Fig. 13.10

A chair positioned in a room before adjusting for the floor's thickness.

To correct this, we need to take into account the height of the floor. Since the floor is 0.2 units in height, we must subtract this value from our original value. This gives us a translation value of 1.8 ($(4/2 - 0.2)$). Therefore if we use the following Transform node, we will get the same result as in Figure 13.2 (which is what we want).

```
Transform {  
  translation 0 -1.8 0  
}
```

Once we have established the correct value for our y-axis translation, we can use other transforms to position our objects along the x- and z-axes. We can also perform rotations to turn or tilt our objects in the desired direction. We can, for example, move our chair to the left rear corner and rotate it 45 degrees so that it faces the opposite corner.

To move our chair to the left rear corner, we need to figure out how far along the x- and z-axes to move it. The room measures 5 units by 5 units along these axes. We could, therefore, move it 2.5 units along each axis. We must remember, however, that our chair is 0.5 units by 0.5 units along the same axes. Moreover, the rear wall is 0.2 units in depth. We don't want to embed our chair in the wall. If we wanted the chair at every corner of the room, we would have to translate it along the z-axis by one-half the depth of the room, less the depth of the rear wall, less one-half the depth of the chair. This would give us a value of 2.05 ($((5/2) - 0.2 - (0.5/2))$ or $2.5 - 0.2 - 0.25$). Using the same formula, we would get a translation value of 2.25 for the x-axis ($((5/2) - 0 - (0.5/2))$). We should put some space (say 0.1 units) between the chair, the rear wall and the edge of the floor as well. When we add our rotation field, we get the following Transform:

```
Transform {  
  translation -2.15 -1.8 -1.95  
  rotation 0 1 0 .7854  
}
```

Figure 13.11 shows how this Transform shifts our chair.

Fig. 13.11

The VRML scene after the Transform.

Transforming Your Objects in Relation to Other Objects

While it is important to think about your objects in relation to their position in your virtual space, you also need to think about their position in relation to other objects in the same environment. Translating them using the same values will result in chaos. When you add another item, you will need to remember where you put the ones already there.

Let's say that you want to add our square table to the room in Figure 13.10. The translation along the y-axis would be the same. And we can adjust the translations along the x- and z-axes using the above method. This gives us the following Transform node:

```
Transform {  
translation -2 -1.8 -1.8  
rotation 0 1 0 .7854  
}
```

However, when the scene is rendered, we discover that our chair has become fused with the table (see Fig. 13.12). We forgot that we put our chair in the same corner.

[Fig. 13.12](#)

Adding a table without considering the position of the chair.

What we need to do here is adjust the table in relation to the chair. Adding 0.25 to the x and z values of the translation field will move the table forward so that it only slightly overhangs the front edge of the chair (see Fig. 13.13)óthis is a much more satisfactory result.

[Fig. 13.13](#)

The same scene after adjusting the table's translation field.

A Word About Collision Detection and Manipulating Objects in a VRML Environment

One of the things that makes a virtual environment more realistic is being able to "bump" into walls and objects, or collision detection. This ability is programmed into a browser. As you move through a virtual world, the browser keeps track of the coordinates for the walls and objects. If your path crosses any of these coordinates, the browser stops your movement. As of this writing, WebFX, WorldView, and Microsoft's Internet Explorer are the only browsers that have this feature.

Unfortunately, most of the other VRML browsers that we tested had not yet implemented collision detection. A number of developers plan to include it in future versions, but it is not clear when this will happen. Until then, you will be able to move through walls and objects as if you are a ghost.

Another element that adds a sense of realism to virtual reality is being able to manipulate objects in the environment. This ability falls into the realm of behaviors (see "Behaviors," Ch. 28). The VRML 1.0

specification does not include behaviors, and so, you cannot directly manipulate objects in a VRML environment. However, you can use the WWWAnchor node to set up a link to a model of a specific object. (See "[Linking Your VRML Environment to the Internet: The WWWInline and WWWAnchor Nodes](#)," Ch. 15).

From Here...

Once you have built and placed the objects for your VRML environment, you can begin the process of fine-tuning. You will want to think about adding texture maps, adjusting the Material node, or adding links to the World Wide Web. You should also think about using lighting to help convey the atmosphere that you want your virtual world to convey.

To learn more about how to add these features to your VRML environment, see the following:

- [Chapter 14, "Adding Realism to Your VRML Environment,"](#) outlines the process of using lighting, materials, and texture maps to add more realism to your VRML world. It goes over the various lighting and texture nodes. It also points to Internet sites where you can download texture maps.
- [Chapter 15, "Adding Links to Your VRML Environment,"](#) outlines how to use the WWWAnchor and WWWInline nodes to connect your environment to the World Wide Web.
- If you want to look at the experiences of others who have created their own VRML environments and models, see Part V, "Learning from the Pros: Sample VRML Environments."
- If you want to see what programs are available to help you create VRML environments, see Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities." These chapters discuss how these programs can help you build and furnish your VRML world.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



14-Adding Realism to Your VRML Environment

Now that we know how to build some simple worlds, it's time to start sprucing them up a bit. In order to make a world look really good, you need to understand how light, materials, and textures work.

In this chapter, we will:

- Look at the four different types of light sources
- Examine the various material properties supported by VRML
- Discuss "normal vectors"
- Explain how "binding" of materials and normal vectors works
- Learn how to use texture mapping

Quite a list. We'll delve into all this neat stuff in a moment, but first let's deal with a sensitive issue.

Compatibility

VRML is designed to be a standard. However, it's still relatively new and there are a lot of rough spots to work out. We'll be looking at some of them later in the book, when we discuss the proposed changes for version 1.1; however, many of these issues have a significant impact on the subject matter of this chapter.

The cold reality is that not all browsers implement the myriad features of VRML. The rendering engines upon which many of them are built are designed for speed, and the generality of VRML's approach to materials and lighting is not a perfect match. Right now, there are over half a dozen rendering engines being used in VRML browsers; all of them have different features. In all cases, there is the same fundamental trade-off between speed and generality.

If you opt for speed when choosing a browser (as most people will), then certain types of lighting and material properties may not be available.

More to the point: most people will use the faster browsers. If you're building a VRML world that you

want to be seen by lots of people, you must design it to work well with the faster renderers; that means you can't rely on the entire specification being implemented. If you avoid those features which are not widely supported, your world will look the same in everyone's browsers; if you use some of the more subtle and esoteric features of VRML, then all bets are off as to who will actually see your world the way you intended it to look. Eventually, many of the VRML browsers will support a large subset of the full specification; in the meantime, stick to the simplest features if you want your world to be viewable by a lot of users.

Throughout this chapter (and elsewhere in this book) I'll be mentioning specific features to avoid if you want your worlds to be widely viewable.

You should be aware that many of the translators, authoring tools, CAD packages, and other VRML-creation programs don't allow you much (if any) control over many of the things we'll be talking about in this chapter.

And now, without further ado, let's discuss the wonderful world of light and color....

How We See

The basic idea behind lighting is simple: light is given off by some source, strikes a surface, and is reflected to our eyes. Of course, if it were really that simple, we wouldn't need a whole chapter to discuss it, now would we?

In the real world, lighting is a very complicated thing. Simulating it accurately in software has been the subject of considerable research for a great many years; several approaches have been found that do a remarkably good job. The best-known is "ray tracing," a technique which simulates light bouncing off a number of objects made up of different materials. Its distinguishing feature is the realism of the images it produces; a good ray-traced image is often indistinguishable from a photograph.

The problem with ray tracing is that it's very computationally expensive; even the fastest machines are much too slow to do it in real time. It's good for rendering a high-quality image over a period of several minutes or several hours, but those kinds of speeds just aren't acceptable in a VRML browser.

VRML browsers are built around rendering engines that use much simpler (and therefore much faster) techniques. As a result, the images are never completely realistic; you can almost always tell that you're looking at a computer-generated scene. However, you can improve the realism of a scene considerably through clever use of light, color, and material properties.

A Rainbow of Possibilities

There are a lot of different ways of representing color. The one used by almost all computer graphics systems is called RGB—a color scheme based on combinations of Red, Green, and Blue. This is the same way that a computer monitor or television set works; there are phosphors that emit red light when struck by the scanning electron beam, there are phosphors that emit green light, and phosphors that emit blue. Similarly, LCD displays have triangular sets of red, green, and blue emitters.

Any color can be represented by some combination of red, green, and blue values. These three values are in the range of 0.0 to 1.0, where 0.0 is no contribution at all from that color component and 1.0 is the maximum contribution. For example,

```
Material { diffuseColor 1 0 0 }
```

would specify a bright red. The first number listed (in this case, 1) is the intensity of red, the second (0) is the intensity of green, and the third (also 0) is the intensity of blue. With a full red component and no green or blue, the result is a pure red. Similarly,

```
Material { diffuseColor 1 1 0 }
```

would produce yellow—a combination of red and green with no blue component.

By using different combinations of values, you can produce shades of any color in the rainbow. Table 14.1 gives a list of some typical color values:

Table 14.1

Black	0	0	0
Blue	0	0	1
Green	0	1	0
Light Blue	0	1	1
Red	1	0	0
Purple	1	0	1
Yellow	1	1	0
White	1	1	1

Note that each component can be any value between 0 and 1, not just those two values.

The examples we've been using so far have been Material nodes, but the basic idea of expressing a color as some combination of red, green, and blue values is the same for lights as it is for materials.

Sources of Light

There are four basic types of light source: directional, point, spot, and ambient. VRML currently supports three of these, but support for the fourth is scheduled for a future release of the specification. All of the lighting nodes in VRML have three fields in common: on, intensity, and color.

The on field can be either TRUE or FALSE; the default is TRUE. Why would you want to create a light that isn't on? You wouldn't, with VRML 1.0; however, once behaviors are added (see Chapter 28) you'll be able to turn lights on and off. The on field just gives the initial state of the light.

The intensity ranges from 0.0 to 1.0, and it determines how bright the light appears to be. If you think of the light as being a dimmer switch, then altering the intensity field is like twisting the knob; changing the on field is like pushing the knob in to turn it on or off.

The color field specifies the red, green, and blue intensities for the light. These are effectively scaled (i.e., multiplied) by the intensity field.

Not all browsers support lights of different colors; some will ignore the color field and treat all lights as white. This may or may not change in future releases of the browsers; in the meantime, do not rely on colored lights if you want your worlds to look more or less the same in all browsers.

Now let's look at what makes each type of light source unique.

Directional Lights

Directional light is emitted by a source so far away that its rays appear to be parallel to each other. The most obvious example of a directional source is the sun; it doesn't matter much where you move to, sunlight seems to be coming from the same direction. Figure 14.1 illustrates how a directional light source works.

[Fig. 14.1](#)

Sunlight, a natural source of directional light.

To create a directional light in VRML, you just use the DirectionalLight node:

```
DirectionalLight {  
on value  
intensity intensity  
color r g b  
direction x y z  
}
```

The direction field gives the direction in which the light source is pointing; for example, if the direction is 0 1 0 then all the light would be shining straight up (in other words, along the positive Y axis) from infinitely far away in the opposite direction.

Point Lights

A point light source emits light uniformly in all directions from a single mathematical point in space. A light bulb or candle flame can be thought of as a point light source, though in reality the filament and the flame are not perfect mathematical points. Figure 14.2 shows how a point light source works.

[Fig. 14.2](#)

A light bulb, acting like a point source of light.

To create a point light in VRML, you use the PointLight node:

```
PointLight {  
  on value  
  intensity intensity  
  color r g b  
  location x y z  
}
```

Notice that a `PointLight` and a `DirectionalLight` have the same syntax, but differ in that a `PointLight` has a location and a `DirectionalLight` has a direction.

Spotlights

A spotlight is, in a sense, a combination of the features of a point source and a directional light. It has both a location and a direction; that is, the light seems to come from a specific point, but it doesn't radiate equally in all directions. Instead, it radiates primarily in a particular direction; that direction is the axis of a kind of "cone of light" that's emitted by the source.

Figure 14.3 shows a spotlight, including its cone.

In addition to the location, direction, and standard lighting fields (`on`, `intensity`, `color`), a spotlight has two other parameters: the `cutOffAngle` and the `dropOffRate`. The `cutOffAngle` is simply the angle of the cone; any objects outside that cone are not affected by the light. The `dropOffRate` determines how fast the intensity of the light diminishes as you move off its axis.

Very few browsers support `dropOffRate`, and the VRML specification is unclear as to just what the `dropOffRate` measures. For maximum compatibility, you should not rely upon this feature.

[Fig. 14.3](#)

A spotlight, with its cone.

The complete `SpotLight` node looks like this:

```
SpotLight {  
  on value  
  intensity intensity  
  color r g b  
  location x y z  
  direction x y z  
  cutOffAngle angle
```

dropOffRate value

}

Ambient Light

Ambient light is unique, in that it's not emitted from any specific source or direction. In reality, it's the result of light from all the various sources, bouncing off objects in the room and contributing to the overall "light level" of the space.

At the moment, ambient light is not supported in VRML. However, the 1.1 version of the specification will include an Environment node which will specify (among other things) the color and intensity of the ambient light. The Environment node is described in Chapter 27.

Number of Lights

Some VRML browsers impose a limit on the total number of lights in the scene. In some cases, the value is as low as eight lights; any more than that, and some lights will be ignored.

This isn't too serious a problem, since performance is adversely affected by each additional light anyway. You want to keep the total number of lights down as much as possible, so you probably won't run into the built-in limit all that often.

.

For maximum compatibility, do not use more than eight lights in your world.

Light and Separators

One serious issue is the question of lighting and Separator nodes.

The current version of the VRML specification says that light stops at Separators; that is, if you have a light as a child of a Separator, it won't illuminate anything outside the scope of the Separator. This is a very peculiar design; no other VR system has a concept of lights that shine only on some objects, but not on others.

Many rendering engines simply don't support such a "feature," and it's quite possible that they never will. Therefore, the browsers that use those engines can't support the "light stops at Separators" feature; this includes most, if not all, of the fastest browsers.

But more fundamentally, the use of a Separator node to limit the influence of a light is a poor design; it uses one language feature (the Separator) to do two completely different and unrelated things: defining an attachment hierarchy and limiting the effect of light sources. This inevitably leads to problems.

For example, it makes it difficult to create something as simple as a flashlight. You want to make the light source a child of the flashlight, positioned inside the flashlight's bulb, but you can't; the Separator around the flashlight will prevent the flashlight's light from shining on anything.

What's more, the 1.0 specification says that lights shine only on objects that appear after them in the

"scene graph"; that again means that some objects are only lit by some lights, which is impossible to implement in most rendering engines.

There's been a lot of discussion of this problem on the VRML mailing list, and it's expected that future versions of the VRML specification will relax or remove this requirement.

For maximum compatibility, you should not make any assumptions about whether or not light stops at Separator nodes. Place all your light sources at the start of your file, ahead of any shape nodes.

Attenuation

In the real world, the intensity of light drops off as the square of the distance between the light source and the surface it's shining on. The presence of fog or mist can further reduce light intensity with distance.

At the moment, VRML provides no means of implementing this feature; however, the proposed Environment node (see Chapter 27) will include attenuation coefficients that will do the job nicely for those renderers which support it.

Materials

When light strikes a surface, two things happen. Some of the light is reflected, and some is absorbed. Of the light that's reflected, some is scattered in all directions and some is reflected in a specific direction. How much reflection/absorption/scattering occurs is determined by the properties of the material itself, as well as the wavelength (i.e., color) of the light.

For example, when white light shines on a red ball, the ball appears red. White light is a combination of all the colors, so it includes red light within it; when we say a material is "red," what we really mean is that it reflects the red part of the light and absorbs the green and the blue. If the ball were white, but had a green light shining on it, it would appear green. If the ball were blue, and had a green light shining on it, it would appear black (since there's no blue in the light for the ball to reflect).

Each color of light interacts separately with a material; that is, you can figure out the red component of an object's overall color by considering only the red component of its material properties and the red components of all the lights shining on it. However, the overall color of an object is determined by the ratio of the red, green, and blue components.

Material Properties

Each material used in a VRML file has a set of properties. Four of these are colors, and they determine how the material will respond to light sources; they are called the `diffuseColor`, `specularColor`, `ambientColor`, and `emissiveColor`. In addition, there's a "shininess" factor and a transparency. Other properties may be added in future versions of VRML.

Diffuse Reflection

When light from a source (directional, point, or spot) shines on a material, some of it is scattered off in all directions. This uniform scattering is called "diffuse reflection." The intensity of the diffusely reflected light is proportional to the angle at which the light is striking the surface; the more directly it strikes, the brighter the reflected light. A shallow angle of incidence will cause less light to be reflected. The intensity of the reflected light is not dependent on the angle at which the viewer is looking at the surface.

Diffuse reflection is represented by a coefficient that scales the intensity of the incident light. In other words, the amount of light striking the surface is computed (taking into account the angle of incidence); that value is then scaled by the diffuse reflection coefficient, to determine what fraction of it is reflected.

In general, the diffuse reflection coefficient is a function of wavelength. In practice, it's usually adequate to simply give three coefficients: one for red light, one for green light, and one for blue light. These are the three values specified in the `diffuseColor` field of a VRML Material node.

In other words... the red component of the incoming light is scaled depending on the angle at which it strikes the surface at a particular vertex, then scaled again by the red component of the `diffuseColor` value. This gives the red component of the diffuse reflection, and these values are added up for all the lights shining on the object; the result is the color of the object at that vertex. For information about how the angle of incidence is used to scale the light intensity, see the mathematical explanation in Appendix A. In Figure 14.4 we see an illustration of diffuse reflection.

[Fig. 14.4](#)

Diffuse reflection.

Specular Reflection

Some of the light that strikes a surface is not scattered in all directions; instead, it's reflected off in a very specific direction that's equal to the angle of incidence. This is called specular reflection, and it's the property that makes a material appear "shiny." A matte-finish photograph has a lot of diffuse reflection and much less specular, while a glossy photograph has lots of specular reflection but much less diffuse.

Since the light is being reflected in a specific direction, the intensity of the reflected light that reaches your eye will depend on the angle at which you're viewing the surface. If the angle that you're looking at a vertex from is very close to the angle at which light is striking the surface, the fraction of the specular reflection that you will see will be greater than if the two angles are far apart.

As with `diffuseColor`, the value of each component of the `specularColor` field for a material is used as a coefficient to scale the corresponding color component of the incident light. However, the viewer's location (and hence the angle towards his or her eye) is used to scale the intensity of the specularly reflected light. The values are added up for all the lights in the scene, and the result is the overall effect of specular lighting.

Figure 14.5 shows how specular reflection works.

[Fig. 14.5](#)

Specular reflection.

Specular reflection is also affected by the "shininess" value of a material. Also called the "Phong exponent," a value that ranges between 0.0 and 1.0, and which determines the size of the specular highlights on a material. The higher the value, the more tightly focused the highlights are. This is independent of the values of the specularColor coefficients.

.

Most of the real-time renderers do not support a separate specularColor value for each of the red, green, and blue color components. Instead, they have a single number representing the overall specular reflection. For maximum compatibility, set all three specularColor values to be the same.

Ambient Reflection

Ambient reflection is similar to diffuse, except that instead of relying upon the angle that the light strikes the surface, it uses the ambient light in the room. Since that information is stored in the Environment node, which isn't officially part of VRML yet, you should probably not use ambient reflection. Again, each color is computed separately; in the case of ambient reflection, the red color component of the light is scaled by the red color component of the ambientColor field.

.

As you might expect, not all renderers support separate color components for the ambient reflection coefficient. For maximum compatibility, leave the ambient light setting at its default value.

The effects of ambient and diffuse light are different. If you turn off all the light sources in the environment, but leave the ambient light on, then it will affect every object in the room the same way. If you turn off the ambient light, and turn on a single light source, it will affect objects differently depending on where they are and which way they're oriented relative to that light source.

Emission

In addition to reflecting light, an object can also emit light. This is not the same as being a light source! A light source by itself is invisible (since it has no geometry), but it can affect other objects in the environment. The emissiveColor, on the other hand, is a property of a material that makes it appear to give off its own light; however, that light does not affect other objects.

For example, if you create a flashlight you would position a SpotLight inside it in order to affect other objects. You might also assign a material with an emissiveColor value to the face of the flashlight, so when you look at it there appears to be light coming out.

Even if there are no light sources in the room (including ambient ones!), a surface whose material has an emissiveColor will still appear to be lit.

.

Starting with VRML 1.1, you can create "pre-lit" objects and scenes by giving a Material node one or more emissiveColor values and no ambientColor, diffuseColor, or specularColor.

Transparency

The final material property we need to look at is transparency. Transparency determines whether you can see through a surface or not. It's a single number in the range from 0.0 to 1.0, where 0.0 means the material is completely opaque (you can't see through it at all) and 1.0 means it's completely transparent (you can't tell it's there). By default, the transparency value is zero.

Not all renderers support transparency values other than 0.0 or 1.0, and those that do may not always do so correctly (since that would involve depth-sorting the objects in the scene, which z-buffering renderers rarely do). Starting with VRML 1.1, renderers that don't support partial transparency will treat values greater than or equal to 0.5 as being the same as 1.0 (fully transparent), and values less than 0.5 as being 0.0 (fully opaque).

How It All Fits Together

If you're sitting at home in your living room at high noon with all the lights out, then what you're seeing is "ambient" light. The light from the sun is bouncing off objects outside, in through your windows, and off of the objects in the room. There's no obvious light source, but you can see everything just fine.

If you're sitting in the same spot 12 hours later (you should try to get a life...), and you turn on a single light in the room, then the light that's reflecting from your couch is diffuse.

If you're in that same situation, but with plastic on your furniture (you really need to get a life...), then the light that's reflecting off the plastic is specular.

And finally, if your couch has some color to it even in total darkness, then it's "emitting" light (check to see if it came from Three Mile Island!).

In general, all four colors will have an effect on the resulting color of the couch. In practice, most of your objects will have a diffuse color and maybe a little bit of ambient color. If you don't specify an `emissiveColor` or a `specularColor`, they default to being zero. The `ambientColor` defaults to "0.2 0.2 0.2" and the `diffuseColor` defaults to "0.8 0.8 0.8." The shininess defaults to 0.2 and the transparency to zero.

Hints on Creating Materials

The `diffuseColor` is what we normally think of as the color of the material. In fact, many renderers use it exactly that way, since they only support white light and treat the ambient and specular color values as simple coefficients rather than actual colors.

You almost always want to set the `ambientColor` to be similar to the `diffuseColor`; they may be scaled relative to each other, but the ratios of their red, green, and blue components should probably be the same. This is because in real life, an object's color doesn't vary depending on whether the light striking it is from a specific source or not.

The color components of the `specularColor` are generally all the same; red equals green equals blue. That way, the specular reflection is for an "all-white" object; in other words, the color of the specularly

reflected light has the color of the light source, rather than the color of the material.

Don't use `emissiveColor` unless the object is supposed to look like it's giving off light (e.g., an indicator panel bulb, a stove element, that kind of thing). You can also use `emissiveColor` for scenes with precomputed lighting.

Nine times out of ten, the only thing you want to set is the `diffuseColor`. The defaults for everything else make good sense.

Systems will generally "clamp" light colors, light intensities, material colors, and computed lighting to be in the range 0.0 to 1.0; if you have too much light in the scene, there's a limit as to how bright it will actually get.

The Material Node

We've already seen the Material node in previous chapters. So far, we've only specified a single diffuse color; Material nodes are much more powerful than that, since they can define all the material properties for a number of different materials at once.

```
Material {
```

```
ambientColor r g b
```

```
diffuseColor r g b
```

```
specularColor r g b
```

```
emissiveColor r g b
```

```
shininess value
```

```
transparency value
```

```
}
```

So far, we've only used the Material node to create a single material. However, it's useful to be able to define an entire array (or table) of materials at once. To do this, each of the fields becomes a "multiple-value field." For example:

```
Material {
```

```
diffuseColor [ 1 0 0, 0 1 0, 1 1 0, 0 0 1, 1 0 1, 0 1 1 ]
```

```
transparency [ 0, 1, 1, 0, 1, 0 ]
```

```
}
```

would define an array of six materials (for example, to apply to the six faces of a cube). The first material would be red and opaque, the second would be green and transparent, and so on.

Notice that when multiple values are specified, the "value" part of the field is surrounded by square brackets and the individual values are separated by commas.

Material Binding

Back in Chapter 11 we discussed how object geometry is created, and now we know how to create materials. The question is, how do we associate materials with geometry? The answer is the MaterialBinding node.

Let's start off with a simple object: a cylinder. As you may recall, a cylinder has three parts: the sides, the top, and the bottom. It turns out that each of these parts can be assigned a different material; for example, you might decide to make the sides of the cylinder blue, the top yellow, and the bottom bright red with a metallic shine to it.

In VRML, the object would look like this:

#VRML V1.0 ascii

Separator {

Material {

diffuseColor [0 0 1, 1 1 0, 1 0 0]

specularColor [0 0 0, 0 0 0, .75 .75 .75]

}

MaterialBinding { value PER_PART }

Cylinder { }

}

The first diffuseColor (0 0 1) is used for the sides; the red and green components are zero, and the blue component is 1, so the sides will be blue. The second diffuseColor (1 1 0) is used for the top; red and green produces yellow. The third diffuseColor is red.

The specularColor for the sides and the top are both (0 0 0), but the bottom has a high specular component. The red, green, and blue values are equal, so the specular highlight will be the color of the light source (even though the surface itself appears red, because of the diffuseColor value).

We used a node called MaterialBinding to indicate that the materials should be "bound" to the parts of the cylinder on a PER_PART basis; that is, the first part (the sides) gets the first material, the second part (the top) gets the second material, and the third part (the bottom) gets the third material.

If we had instead used

MaterialBinding { value OVERALL }

then all the parts of the cone would have used the first material in the list.

A cone is handled exactly the same way, except that only the first two materials would be used since it has no top.

A sphere is also similar, except it only uses the first material (since it only has one surface).

Notice that even though the cylinders, spheres, and cones are tessellated into faces, there's no way to bind materials to the faces themselves. This makes sense; after all, the browser can use as many or as few faces for those objects as it chooses, so there's no way to know how many materials to specify.

A cube is handled similarly, with each face being one "part." You can also use PER_FACE binding; in the case of a cube, it's the same as PER_PART.

So for the simple geometric primitives, things are straightforward. What about IndexedFaceSets? We might want to give each face its own material. It turns out that's easy to do; let's use our pyramid example again:

```
#VRML V1.0 ascii
```

```
DEF Pyramid Separator {
```

```
Material {
```

```
diffuseColor [ 0 0 1, 0 1 0, 0 1 1, 1 0 0, 1 0 1 ]
```

```
}
```

```
MaterialBinding { value PER_FACE }
```

```
Coordinate3 {
```

```
point [
```

```
-1 0 -1,
```

```
1 0 -1,
```

```
1 0 1,
```

```
-1 0 1,
```

```
0 1 0
```

```
]
```

```
}
```

```
IndexedFaceSet {
```

```
coordIndex [
```

```
0, 4, 1, -1,
```

```
1, 4, 2, -1,
```

```
2, 4, 3, -1,
```

```
3, 4, 0, -1,
```

```
0, 1, 2, 3, -1
```

```
]
}
}
```

We've added definitions for five different materials; to keep things simple, we're only using the `diffuseColor` for each one. The `PER_FACE` material binding means that the first material will be applied to the first face (the one that uses vertices 0, 4, and 1), the second material will be applied to the second face (1, 4, 2), and so on. We could also have used `PER_PART`, since it's the same as `PER_FACE` for an `IndexedFaceSet`.

We could, of course, have used `OVERALL` material binding. That would mean that the entire pyramid would have used the first material.

Now, what if we know that two sides of the pyramid are green, two are blue, and the base is gray. Do we still have to specify five different materials, even though only three are being used? It turns out we don't; we can use a single material for multiple faces by using `PER_FACE_INDEXED` material binding.

#VRML V1.0 ascii

DEF Pyramid Separator {

Material {

diffuseColor [0 0 1, 0 1 0, .5 .5 .5] # only _materials are specified

}

MaterialBinding { value PER_FACE_INDEXED }

Coordinate3 {

point [

-1 0 -1,

1 0 -1,

1 0 1,

-1 0 1,

0 1 0

]

}

IndexedFaceSet {

coordIndex [

0, 4, 1, -1,

1, 4, 2, -1,

2, 4, 3, -1,

3, 4, 0, -1,

0, 1, 2, 3, -1

]

materialIndex [0, 1, 0, 1, 2]

}

}

The materialIndex field contains a value for each of the five faces. The first face uses material 0, the second uses material 1, the third uses material 0 again, the fourth uses material 1, and the fifth (the base, whose vertices are 0, 1, 2, and 3) uses material 2. Notice that we start counting from zero; the first material is material number 0, the second is 1, and so on.

For a simple shape like a pyramid, there may not seem to be much of an advantage to doing things this way. However, for a complex shape (such as a human being, for example) the ability to reuse materials can save a lot of memory and a lot of downloading time. It also makes it really easy to make changes; if you define a material for skin tone, and decide you want your character to have a tan, you just change one material entry instead of one for each face.

The bindings for an IndexedLineSet work the same way as for IndexedFaceSet, except that the PER_PART binding is not treated the same as PER_FACE; instead, each line segment is considered to be a separate "part." Each sequence of lines, ending with a -1, is treated as a "face" when the binding is PER_FACE. You can also use PER_PART_INDEXED for an IndexedLineSet, in which case the materialIndex field gives the index into the material array for each line segment's color.

For a PointSet, OVERALL binding uses one material for all the points; any other MaterialBinding value will assign a different material to each point. The _INDEXED versions of the bindings are not used since the PointSet node doesn't have index fields.

More MaterialBinding Values

The MaterialBinding node's value field can be any of the following:

DEFAULT	Uses whatever the default binding is for a particular shape
OVERALL	Uses the first material for the entire shape
PER_PART	Uses one material for each "part" of an object
PER_FACE	Uses one material for each face of an object
PER_FACE_INDEXED	Uses the materialIndex field to find out which material to use for each face
PER_PART_INDEXED	Uses the materialIndex field to find out which material to use for each part

The DEFAULT binding for each type of shape is listed in the VRML specification in Appendix B.

In addition to these bindings, there are two others:

PER_VERTEX Uses one material for each vertex

PER_VERTEX_INDEXED Uses the `materialIndex` field to find out which material to use for corner of each face; see the note on **PER_VERTEX_INDEXED** bindings later in this chapter.

Very few browsers actually support **PER_VERTEX** and **PER_VERTEX_INDEXED** material bindings. This isn't surprising; there are very few cases where you would want a single face to be made of wood at one corner, plastic at a second, and gold at a third. The 1.0 version of the VRML specification doesn't make it clear how browsers should deal with **PER_VERTEX** materials if they can't use them; more likely than not, the materials at all the vertices of the face will be averaged to produce a single material for the face itself, but that's not something you should count on. In general, avoid **PER_VERTEX** material binding.

Pre-lit Scenes

There is one situation when **PER_VERTEX** material binding is useful: precomputed lighting. You can use a technique called "radiosity" to compute a color at each vertex, and then do smooth shading at runtime. The result is very realistic lighting that actually runs faster than a regular, lit scene would (since no lighting calculations need to be done at runtime). Of course, this really only works well for static scenes; if objects or lights move around, then all the values would theoretically have to be recomputed.

In any case, the combination of unlit materials (with multiple `emissiveColor` values, and nothing else) with **PER_VERTEX** `MaterialBinding` can be useful for rendering pre-lit, radiosity-computed scenes using VRML browsers. Assuming, of course, that the browser supports this type of binding.

Normals

In order to compute the lighting for an object, you need more than just light sources and materials; you also need to have things called "normal vectors." As you may remember from the discussion of diffuse reflection earlier in this chapter, the intensity of the light hitting a surface depends on the angle at which the light is coming in. For example, if you have a flat surface (like a sidewalk) it will be brighter at high noon than at 4 p.m.; at high noon, the light is striking the surface straight-on, whereas in the late afternoon it's striking it at a much shallower angle.

A normal vector can be thought of as an imaginary arrow, sticking out from the surface of an object. In general, normal vectors are perpendicular to a surface; in other words, for a flat face of a cube, the normal vector points out at right angles to the surface. Figure 14.6 shows the normal vectors for a cube.

Fig. 14.6

The normal vectors for a cube.

For a curved surface, the normal will vary continuously over the surface. At the top of a hill, the normal would point straight up; as you descend the side of the hill¹, the normal would point more horizontally. Figure 14.7 illustrates this.

Fig. 14.7

How the normal varies across a surface.

For a sphere, the normal vectors generally point outwards, like an extended radius. Figure 14.8 illustrates this.

Fig. 14.8

The normals for a sphere.

VRML gives you a way to specify these normal vectors; it's called the Normal node:

```
Normal {  
vector [ x y z ]  
}
```

This is just like the Coordinate3 field, except that instead of giving vertex locations, it gives normal vectors.

Computing the normal vectors at each vertex by hand is tedious. Fortunately, VRML assumes that all objects will have reasonable normals even if you don't specify any; it will generate a normal for each face (see Appendix A for the math), and then (if necessary) at each vertex it will compute the average of the normals of all the faces sharing that vertex. Therefore, each vertex is assigned a normal vector, which gives a direction perpendicular to the surface at that point.

This works pretty well, most of the time. However, there are odd times where you need to be able to explicitly specify normals. If you have an object that you know is very heavily curved, you'll find that converting it to faces and then computing normal vectors won't be very accurate. It's better to generate the normal vectors from the original, mathematically perfect, smooth shape before converting to faces.

If you're creating objects with pencil and paper, then they'll be starting out as faces; in most such cases you can let the VRML browser work out the average normals for you, since that's all you'd be doing anyway. If you're using a 3D modeling package of some sort, it will generate the normals for you automatically, if it needs to (it may also choose to let the browser do it).

Letting the browser do it is not just taking the easy way out; it also reduces the size of the VRML file, since the normals don't have to be stored or transmitted.

Normal Binding

Just as you can "bind" a material to a face (or part) of an object, you can also bind a normal vector. The way the normals are "bound" to the object determines what kind of shading is done; ultimately, a normal value per vertex (per face) is computed, but how it's computed determines whether the object looks smooth or faceted.

If the NormalBinding is PER_VERTEX, then one normal from the Normal node is associated with each

vertex. The lighting calculations can be done once for each vertex, and the colors are interpolated; the result is a smooth-shaded object.

If the NormalBinding is PER_FACE, then all the vertices of each face will share a common normal vector. One normal per face will be specified in the Normal node. (If no normals are specified, a normal is computed using the techniques described in Appendix A.) Since all the vertices share the same normal, the lighting at each vertex will be the same; the colors will therefore also be the same, and no interpolation has to be done across the face. The entire face will be a single color, whose brightness depends on the angle of the light reaching it (relative to its normal vector). The result is an object that looks faceted instead of smooth; the effect looks something like the "Bizarro Superman" from DC comics.

If the NormalBinding is PER_FACE_INDEXED, then the normalIndex field of the IndexedFaceSet or IndexedLineSet is used; it stores indices into the array of normals the same way that the materialIndex field stores indices into the array of materials.

PER_PART and PER_PART_INDEXED work the same way, except that for an IndexedLineSet the normal is per line segment. Why would you want to do this? Well, in general you wouldn't. You can also specify OVERALL normal binding, but it doesn't make much sense in most cases.

In general, you won't have much use for PER_PART or PER_PART_INDEXED normals, or for OVERALL binding. If you want an IndexedFaceSet to appear smooth, bind the normals PER_VERTEX; if you want it to be faceted, bind them PER_FACE.

PER_VERTEX_INDEXED Binding

There has been considerable confusion about the subject of PER_VERTEX_INDEXED binding. Surprisingly, "PER_VERTEX_INDEXED" does not bind a normal (or material) to each vertex! It actually binds one to each corner of each face. That means a particular vertex could theoretically have a different material (or normal) for each face it's associated with.

This actually turns out to be useful, since there will be cases where you want a single vertex to have two different normals. For example, consider a circular coffee table; the faces around the perimeter will have their vertex normals pointing horizontally outwards, but the faces making up the top surface will have their vertex normals pointing straight up. A single vertex (in a Coordinate3 node) that lies on the edge between the top and the sides of the table would have a different normal vector in each face; the vertex coordinates are being reused, but the normals aren't.

When PER_VERTEX_INDEXED binding is used, there is a one-to-one correspondance between entries in the coordIndex array and the normalIndex array. Entry number 5 in the coordIndex array will specify a particular vertex to use for a point in a face, and entry number 5 in the normalIndex array will specify which normal vector to use for that same point.

While at first this may seem confusing (since PER_VERTEX_INDEXED may seem to be completely different from PER_VERTEX), it actually makes sense. Think of PER_VERTEX binding as a special case of PER_VERTEX_INDEXED binding, in which the values in the coordIndex array are used instead of the values in the normalIndex array; that means that there is in fact a one-to-one relationship between

entries in the Coordinate3 node and entries in the Normal node.

Note that texture coordinates are always bound PER_VERTEX_INDEXED.

ShapeHints Revisited

You may remember the ShapeHints node that was described back in Chapter 11. Well, it has a couple of additional fields that are relevant here.

The creaseAngle field of the ShapeHints node controls the threshold for merging generated normals. What does that mean? Well, remember that if you don't specify a Normal node, then normals are generated automatically. The procedure is to compute a normal for each face, then average them to get a normal for each vertex.

Well, once the face normals are computed they are compared with each other. If two adjacent faces have a shallow enough angle between them, then their normals will be treated as being identical; this means that the crease between them won't be visible (hence the name creaseAngle). If the angle between the normals is less than the creaseAngle, then there'll be no seam and the surfaces will appear to be smoothly joined. If the angle is greater than the creaseAngle, then there will be a seam and the two surfaces will appear to be joined along a visible edge. The creaseAngle is specified in radians; the default value is 0.5 radians, or about 29 degrees.

The other ShapeHints field is shapeType (not to be confused with faceType). The shapeType field will be either SOLID or UNKNOWN_SHAPE_TYPE. A SOLID shape is one that has no openings in it; you can't see any of the inside faces. This lets the browser do some optimizations; if it knows the vertex ordering (i.e., vertexOrdering is not UNKNOWN_ORDERING) and knows that the inside faces are never visible, it doesn't have to light the interior of the object and can also do backface removal. If the vertexOrdering is known but the shapeType is UNKNOWN_SHAPE_TYPE, then the inside faces must be lit, but only the visible ones need to be rendered.

If neither the shapeType nor the vertexOrdering is known, then the browser has no choice but to light and render all the faces, even ones that can't be seen.

Starting in VRML 1.1, the default setting for shapeType will be SOLID and the default setting for vertexOrdering will be COUNTERCLOCKWISE. To be on the safe side, always include a ShapeHints node that explicitly states the ordering of your vertices (even if it's UNKNOWN_ORDERING) and whether your shapes are SOLID or UNKNOWN_SHAPE_TYPE.

Textures

In addition to the material specifications and normals, it's also possible to assign a "texture map" to an object. A texture map is a two-dimensional bitmapped image (i.e., a regular picture) that can be applied to the surface of an object. For example, you could take a picture of a brick wall, then scan it and use it as a texture map for the walls of your virtual world. You could also take a picture of yourself, and apply it as a texture to a flat surface inside a picture frame in your virtual home.

Loading Textures

A texture map can be loaded from anywhere on the Internet, using the `Texture2` node:

```
Texture2 {  
filename URL  
wrapS REPEAT (or CLAMP)  
wrapT REPEAT (or CLAMP)  
}
```

The URL is a Universal Resource Locator, which you should be familiar with if you've ever used the World Wide Web. The "wrap" values will be described below.

The URL refers to a file in some bitmap format. There's considerable confusion about which bitmap formats should be supported (there are literally dozens of them) but the proposed 1.1 specification requires JPEG and suggests PNG. The JPEG format stores compressed full-color images (which start out as 24 bits per pixel). The compression is "lossy," which means some picture information is thrown away to achieve higher compression ratios. The PNG format is lossless, but gets lower compression; it's a replacement for GIF, which is no longer being used due to licensing restrictions.

There's another way to specify the bitmap, but it's not likely to be used much. It's the "image" field, and it's described in the complete VRML specification in Appendix B. It stores the actual pixel values in hexadecimal. The reason it's not generally used is that it's incredibly inefficient; not only is there no compression, there's actually expansion since the data must be represented in ascii instead of binary. This not only takes up a lot of disk space, it chews up a tremendous amount of bandwidth. Worse yet, you can't reuse such a texture in multiple worlds, the way you can if the texture is stored in a separate file. Unless you're creating a tiny, tiny texture, you're much better off using JPEG.

One reason for using the PNG format is that it stores a distinct "alpha channel." An alpha channel stores a value for each pixel, which corresponds to that pixel's transparency. This allows you to create textures that have "holes" in them through which you can see objects on the far side of the surface. For example, you might create a chain-link fence as a single polygon, with the image of the chain links as the texture and an alpha value of 1.0 where there's no chain.

One interesting use of transparent texture maps is to produce trees. You take a photo of a tree and use it as a texture map; you create a transparency map (with alpha set to 1.0 wherever you can see through the leaves and branches of the tree) and apply the texture to two intersecting polygons as shown in Figure 14.9; the result looks good from almost any angle, and is much less of a computational burden than a tree modeled in 3D. You can have the best of both worlds by using LOD to replace this simple intersecting-poly version with a 3D tree when the user gets close enough.

For all this to work properly, the material used for the polygons should have a transparency value of 1.0, so that not only is the texture transparent but the underlying polygon as well.

[Fig. 14.9](#)

Two intersecting polygons with transparent texture maps are useful for making trees.

Applying Textures

For the geometric shapes, there are standard ways of applying the texture; for example, the texture is automatically applied to each face of a Cube. However, for a complex shape described by an IndexedFaceSet, it's not obvious how the texture should be applied. The solution to this is the TextureCoordinate2 node, and the textureCoordIndex field of an IndexedFaceSet.

For each point on each face we have a pair of numbers in the range of 0.0 to 1.0. These two numbers represent a point in the two-dimensional image that corresponds to that corner of the face. For example, let's say we want to apply a texture to each face of our pyramid:

Listing 11.5

#VRML V1.0 ascii

Separator {

Texture2 { filename "bridge.bmp" }

Coordinate3 { point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1, 0 1 0] }

TextureCoordinate2 { point [0 0, .5 1, 1 0, 0 1, 1 1] }

IndexedFaceSet {

coordIndex [

0, 4, 1, -1,

1, 4, 2, -1,

2, 4, 3, -1,

3, 4, 0, -1,

0, 1, 2, 3, -1

]

textureCoordIndex [

0, 1, 2, -1,

0, 1, 2, -1,

0, 1, 2, -1,

0, 1, 2, -1,

0, 3, 4, 2, -1

]

```
}
```

```
}
```

Figure 14.10 shows the resulting texture-mapped pyramid.

Fig. 14.10

A texture-mapped pyramid.

The textureCoordIndex field is similar to the coordIndex field (or the materialIndex, or the normalIndex), and it consists of indices into the TextureCoordinate2 array. There is a one-to-one correspondence between entries in the textureCoordIndex array and the coordIndex array. Each value in the coordIndex array refers to an entry in the Coordinate3 array that gives the X, Y, and Z values for the corresponding point on a face; the textureCoordIndex array refers to an entry in the Texture2Coordinate array that gives the texture map coordinates for that same point on the face.

For example, the second face in the pyramid above uses the vertices at locations 1, 4, and 2 in the Coordinate3 array. The first of those three points (the one using entry 1 in the Coordinate3 array) uses entry number 0 in the Texture2Coordinate array. The second of the three points (the one using entry 4 in the Coordinate3 array) uses entry number 1 in the array. The third (coordinate 2) uses entry 3 in the Texture2Coordinate array.

Complicated? You bet! And none of this is clear from the VRML 1.0 specification itself. As a result, not all VRML software handles this correctly.

At the moment, it's almost impossible to create texture-mapped worlds that look the same in every browser. The only thing that's guaranteed to work (at least in browsers that do texture mapping!) is to only texture-map simple, rectangular polygons. The default is to map the four corners of the texture map to the four corners of the face, which is generally what you want.

Transforming Textures

Texture coordinates can be transformed using the Texture2Transform node:

```
Texture2Transform {
```

```
translation x y
```

```
rotation angle
```

```
scaleFactor x y
```

```
center x y
```

```
}
```

This is like a two-dimensional version of the Transform node that was described in Chapter 11. Like the Transform node, the order of operations is scaling, followed by rotation, followed by translation. The texture coordinates are first stretched (scaled), then rotated, and finally shifted (translated) before being used to index into the texture map. The center field specifies the location around which scaling and

rotation are carried out; it defaults to (0 0) and can be left out.

The texture transform is applied to the coordinates, not the image! That means that if you increase the scale value (for example), the actual texture gets smaller on the object. It's also likely to be repeated across the object (assuming you've left the wrapS and wrapT values set to their default; see below).

A lot of browsers have not yet implemented texture mapping, so don't be surprised if it doesn't work for you. The ones that have implemented it haven't always done it correctly.

The Wrapping Fields

The texture coordinates in the example above are all in the range of 0.0 to 1.0; that's great, but what if they go outside of that range as a result of a texture transform? There are two choices: either the texture can be "tiled" (repeated as often as necessary to fill the surface) or "clamped" (values less than 0.0 become 0.0, and values greater than 1.0 become 1.0). The default value is REPEAT (in other words, tile). In Chapters 17 and 22 we'll see examples of the use of tiled textures.

The current (1.0) version of the VRML specification only allows one texture per object. There's no way to put different textures on each face, or to have some faces texture-mapped but not others. That's a serious limitation, and it's expected that future versions of the VRML specification will fix this problem. For now, you can put different textures on each face by splitting your object up into a number of individual IndexedFaceSets and using a different texture map for each. Yes, it's a royal pain in the rear.

Inheritance

Textures, texture transforms, materials, and bindings are all inherited from parent nodes to children nodes (and to grandchildren nodes, and so on).

For example, if you specify a texture map using a Texture2 node, that texture will be used for every subsequent object lower down in the hierarchy unless it's overridden with another texture. You don't need to keep respecifying the texture for every object; this saves time and space. To turn off a texture, use a Texture2 node with a filename of " ".

Rules of Thumb

We've covered a lot of territory in this chapter, and a lot of it's pretty complex. If you're just starting out, here are some simple rules to follow to make use of lights, materials, and textures.

These are not hard-and-fast rules; they're really just guidelines that will help you create worlds that can be viewed on almost any browser.

Lights

Don't bother setting the on field, since it defaults to being TRUE (which is what you want).

Don't bother setting the color field, since the default is white light (which is what you want most of the time). Also remember that some rendering engines only support white light.

Don't bother setting the dropOffRate, since many renderers don't support it yet.

Materials

Decide on what the basic color of the material should be, and set the diffuseColor to that value. A color cube would be useful for this. Remember, the ratio of red to green to blue determines the color, and the overall magnitude of the values determines the brightness of the color.

Set all three components of the specularColor to the same value; this will cause the highlights to take on the color of the light source. The magnitude of those values determines how much specular reflection there is. The shininess value determines the size of the specular highlights (larger values produce smaller areas of highlight).

Leave the ambientColor at its default value.

Only use emissiveColor for things that are intended to glow or give off light.

If you want a surface to be transparent, give it a transparency value greater than or equal to 0.5; this will ensure that the surface will be transparent on all renderers.

Textures

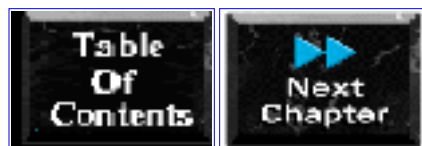
Use JPEG for textures, unless you need transparency; if you do, then use PNG. Be aware that neither is universally supported yet, but they will be eventually. For maximum compatibility, stick to simple rectangular faces.

From Here...

Well, now you know how to create good-looking, professional worlds with lights, materials, and textures. In [Chapter 16](#), you'll learn how to set up attractive viewpoints into our virtual worlds.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



15-Linking Your VRML Environment to the Internet the WWWInline and WWWAnchor Nodes

You have created a VRML space, furnished it with objects for your users to see, and added an element of realism using lighting and texture maps. But say that you want to connect your space to other VRML environments on the Internet. Or maybe incorporate objects located at other Web sites. Or, perhaps you want people to be able to go to a bookshelf, click on a volume and get a Web page dealing with a certain topic.

One of the main features built into VRML is the ability to connect your 3D model or environment to the World Wide Web. There are two specific nodes that implement this property: the WWWAnchor and WWWInline nodes. The WWWAnchor node lets you set up a link to another VRML environment or to a service on the World Wide Web. The WWWInline node lets you incorporate VRML objects from other Web sites into your environment. Both these nodes will play key roles in the success of the VRML standard.

This chapter examines both the WWWAnchor and WWWInline nodes. It outlines how they are used and how to incorporate them into your worlds. The chapter also examines the following areas:

- Using the WWWAnchor node to connect different rooms in your VRML environment.
- Why you should use the bboxSize and bboxCenter fields.
- Sending information using the WWWAnchor node.

The WWWInline Node

The WWWInline node is one of the most important elements in the VRML language. This node lets you call up models stored as individual files from some other locations on the Internet. This means that you can store your own objects in separate files so that you can easily reuse them in other environments.

The definition of the WWWInline node states that it is a group node that can read its children from

anywhere on the World Wide Web. Moreover, the reading can be delayed. The format for this node is as follows:

```
WWWInline {  
name ""  
bboxSize 0 0 0  
bboxCenter 0 0 0  
}
```

The name field lists the URL of the child node and is always surrounded by quotation marks. For example, if you wanted to bring in a model from another site on the Internet (for example, the Polynesian Platform Bird Feeder at the University of Arizona), you would write:

```
WWWInline {  
name "http://amber.rc.arizona.edu/vrml/birdfeeder.wrl"  
}
```

However if you are testing inlines on your own computer, you would write:

```
WWWInline {  
name "file:///d:/direct/FILENAME.WRL"  
}
```

where d is your drive name, direct is the directory where your file is located and FILENAME.WRL is the name of the file you want to use. For example, you could write

```
WWWInline {  
name "file:///c:/webfx/vrml/chair.WRL"  
}
```

to mean that you are calling a file named CHAIR.WRL from drive C in the subdirectory WEBFX\VRML.

The bboxSize and bboxCenter fields are optional; they let you define a bounding box around the space that will be occupied by the object being called up by the node. A bounding box is simply an imaginary cube that outlines where the model will appear. The bboxSize field defines the size of the cube; the bboxCenter field lets you set the x, y, and z coordinates of the cube's center point. The bounding box fields do not affect the object itself. They only provide a hint to the browser.

For example, let's say that we are using a WWWInline node to call up our VRML chair model. We are placing it in a space defined by a flat, square polygon, rotating 45 degrees and translating it 2 units on both the x- and z-axes. We know that both our chair's width and depth value is 0.5 units. Its height is the sum of the height of its legs, seat, and back. This equals 1.2 units (0.4+0.05+0.75). We do not have to

adjust the rotation or the x or z coordinates of the bounding box because the WWWInline node will be governed by the Transform node in this file.

However, we will have to adjust the y-coordinate of the bounding box because our chair uses a translation in its file to set the bottom edge of the chair at the zero mark of the y-axis. We will have to adjust the y coordinate in the bboxCenter field to match this adjustment. Since the height of the chair is 1.2 units, the y coordinate in the bboxCenter field has to be 0.6 (the_height_of_the_chair/2). We would then write our .wrl as follows:

#VRML V1.0 ascii

Separator {

#The Floor

Separator {

Material {

ambientColor 0 0 1

}

Coordinate3 {

point [

-3 0 -3, -3 0 3, 3 0 -3, 3 0 3

]

}

IndexedFaceSet {

coordIndex [

0, 1, 3, 2, -1

]

}

}

#Inlining a Simple Chair

Separator {

Transform {

translation -.5 0 -.5

rotation 0 1 0 .75

```

}
WWWInline {
  name "file:///C:/WEBFX/VRML/CHAIR.WRL"
  bboxSize .5 1.2 .5
  bboxCenter 0 .6 0
}
}

```

When this file is called up in a VRML browser, the area where the chair will appear is outlined with a wireframe cube (see Fig. 15.1).

[Fig. 15.1](#)

A bounding box for an object called up by the WWWInline node.

After the WWWInline node retrieves the model, however, the bounding box disappears, and the object is visible, as shown in Figure 15.2.

[Fig. 15.2](#)

A chair called up by the WWWInline node.

The bounding box is particularly important when using the WWWInline node to access objects over the Internet. Internet traffic, server backlogs, VRML environment size, and user hardware can delay the retrieval of an inlined node. The box lets users start interacting with the environment before everything has been downloaded. It is also used for deciding when (and whether) to download the object.

The WWWAnchor Node

The WWWAnchor node functions like an HTML link. When you click on an object that is a child of this node, you activate a call through your VRML browser. You can use it to establish a link with another VRML scene (this type of link is sometimes called a teleport or portal), or to another type of file like an HTML document. In the case of the latter, most VRML browsers will delegate the task of handling this call to your Web browser.

The WWWAnchor node acts just like a Separator. This node's format is:

```

WWWAnchor {
  name ""
  description ""
  map NONE
}

```

The name field in the WWWAnchor node functions in the same way as in the WWWInline node. The

field lists the URL of the child node and is always surrounded by quotation marks. If you wanted to call up an environment from another site on the Internet (for example, the Modzelewskiego Street environment), you would write:

```
WWWAnchor {  
name "http://andante.iss.uw.edu.pl/cgi-bin/modzel"  
}
```

However if you are testing anchors on your own computer, you would write:

```
WWWAnchor {  
name "file:///d:/direct/FILENAME.WRL"  
}
```

where d is your drive name, direct is the directory your file is located, and FILENAME.WRL is the name of the file you want to use. For example, say you wanted to test an anchor with an HTML file called stephen.htm that is in the directory htmled\files. The directory is on your computer's C: drive. You would use the following:

```
WWWAnchor {  
name "file:///c:/htmled/files/stephen.HTM"  
}
```

This will instruct your browser to find the file called STEPHEN.HTM in the subdirectory HTMLEL\FILES on Drive C.

Both Netscape and Microsoft's Internet Explorer do not require that you type-in the protocol specifier (for example, http://) in most situations. However, this is not the case with all VRML browsers. Plug-in versions like WebFX and Virtual Explorer operate through the Web browser, and so will not always need the protocol specifier. However, stand-alone VRML browsers may or may not require that you include it with the URL. You can experiment with your browser to see if it will let you omit the protocol specifier.

The description field lets you provide users with some text to identify the link. You can enter any string of text you like, but it must always be surrounded by quotation marks. Therefore, you should not include any quotation marks in your string unless you "escape" them using a backslash (\). The description field doesn't change the behavior of the WWWAnchor node. However, a VRML browser will display the text string whenever you move the cursor over the child of that WWWAnchor.

At the time of this writing, not all VRML browsers display what is in the description field. However, they are still able to use the WWWAnchor node.

The map field allows you to pass on the coordinates of the point on the object that you clicked along with the URL in the name field. This can be used in conjunction with a script to change the contents of the incoming file. To turn this feature on, you must change the field's value from NONE (which is the

default) to POINT. When you click on the anchored object, your browser will add the coordinates preceded with a question mark ("?",) to the end of the URL. For example, say your WWWAnchor uses the following URL in its name field:

<http://www.yorku.ca/~smatsuba/VRML/chair.wrl>

A user decides to click on a child of that WWWAnchor group at the coordinates 1.7 2 2. The HTTP server at yorku.ca would then receive a request with the following URL:

<http://www.yorku.ca/~smatsuba/VRML/chair.wrl?1.7,2,2>

Using the WWWAnchor Node

Any object or group of objects can be linked using the WWWAnchor node. Think about using this node in the same way as you would the Separator node. To group objects together in VRML, like the parts of a table or chair, we surround the different components with a Separator node. Well, in order to set an anchor, we need to surround an object or group of objects with a WWWAnchor node.

For example, we have created a VRML model of a table, and we have decided to put three objects on it: a cone, a sphere, and a cube. We have decided to make a link using those three shapes to an HTML document about this book. In order to do this, we need to have the URL for that document. Since I am doing this on my computer, the URL is file:///c:/htmled/vrmlbook. HTM. Also, we can use the description field to indicate that this link is related to this book. To assign the same anchor to all three objects, I would write the code as follows:

[#VRML V1.0 ascii](#)

[Separator {](#)

[#This is the Table](#)

[Separator {](#)

[Separator {](#)

[Translation {](#)

[translation 0 .75 0](#)

[}](#)

[Material {](#)

[ambientColor 1 1 1](#)

[}](#)

[Cylinder {](#)

[parts ALL](#)

[height 0.1](#)

radius .75

}

}

}

Separator {

Translation {

translation 0 .375 0

}

Material {

ambientColor 0 0 1

}

Cylinder {

parts ALL

height .75

radius .375

}

}

}

Separator {

WWWAnchor {

name "file:///c:/htmled/vrmlbook.htm"

description "Special Edition Using VRML"

Separator {

Translation {

translation .35 .8 0

}

#The Cone

Separator {

Translation {

translation 0 .1 0

}

Material {

ambientColor 0 0 1

}

Cone {

parts ALL

bottomRadius 0.05

height 0.2

}

}

#The Cube

Separator {

Translation {

translation -.1 0.05 -.1

}

Material {

ambientColor 0 1 0

}

Cube {

width .1 height .1 depth .1}

}

#The Sphere

Separator {

Translation {

translation .1 0.05 -.2

}

```

Material {
  ambientColor 1 0 0
}

Sphere {
  radius .05
}

}

}

}

} # This is the curly brace that ends the WWWAnchor node
} # This is the curly brace for the Separator Node

```

Note that while we place the WWWAnchor node before the shape nodes, the curly brace that marks the end of the WWWAnchor node comes after the shape nodes. In other words, we mark the cube, sphere, and cone as links by making them children of the WWWAnchor node. Now when we move our cursor over any of the three objects, we can see that they are links to something else (see Fig. 15.3).

[Fig. 15.3](#)

A VRML scene with linked objects using the WWWAnchor node.

And when we click on any of the objects, our HTML document will be called up (see Fig. 15.4).

[Fig. 15.4](#)

An HTML document with linked objects using the WWWAnchor node.

If, however, we wanted to use the WWWAnchor node to link our HTML document with only the cube, we would make that shape the only child of the WWWAnchor. Thus, the relevant portion of our .wrl file would look like the following:

```

#The Cone

Separator {
  Translation {
    translation 0 .1 0
  }

  Material {
    ambientColor 0 0 1
  }
}

```

```
Cone {  
parts ALL  
bottomRadius 0.05  
height 0.2  
}  
}  
#The Cube  
Separator {  
WWWAnchor {  
name "file:///c:/htmled/vrmlbook.htm"  
description "Special Edition Using VRML"  
Separator {  
Translation {  
translation -.1 0.05 -.1  
}  
Material {  
ambientColor 0 1 0  
}  
Cube {  
width .1 height .1 depth .1}  
}  
}  
} # This is the curly brace that ends the WWWAnchor node  
}  
#The Sphere  
Separator {  
Translation {  
translation .1 0.05 -.2
```

```

}
Material {
  ambientColor 1 0 0
}
Sphere {
  radius .05
}
}
}

```

Now the cursor will indicate a link only when we place it over the cube.

Using the WWWAnchor and the WWWInline Nodes Together

We can place each object in a separate file, place it in a VRML environment using the WWWInline node, and set up a link to the model file with the WWWAnchor node.

For example, we can display a chair and a table together in a VRML environment and let users examine the individual items. First, we create VRML models of the chair and the table storing each model in a separate file. We then place them in a virtual environment using the WWWInline node.

The next step is to set up links to the two models' .wrl files using the WWWAnchor node. We make each WWWInline node a child to the corresponding inline for each object. To set up an anchor to the chair inline, we construct the following code:

```

WWWAnchor {
  name "file:///C:/WEBFX/VRML/CHAIR.WRL"
  description "A Simple Chair"
  WWWInline {
    name "file:///C:/WEBFX/VRML/CHAIR.WRL"
    bboxSize .5 1.2 .5
    bboxCenter 0 .6 0
  }
}

```

This will call up the CHAIR.WRL file and insert the object in our VRML environment. We can also add transforms to the inline. We can do the same for our model of the table. When we have finished, we would have a .wrl file that looks like the following:

#VRML V1.0 ascii

Separator {

#The Floor

Separator {

Material {

ambientColor 0 0 1

}

Coordinate3 {

point [

-3 0 -3, -3 0 3, 3 0 -3, 3 0 3

]

}

IndexedFaceSet {

coordIndex [

0, 1, 3, 2, -1

]

}

}

#Inlining and Linking a Simple Chair

Separator {

WWWAnchor {

name "file:///C:/WEBFX/VRML/CHAIR.WRL"

description "A Simple Chair"

Transform {

translation -.5 0 -.5

rotation 0 1 0 .75

}

WWWInline {

name "file:///C:/WEBFX/VRML/CHAIR.WRL"

bboxSize .5 1.2 .5

bboxCenter 0 .6 0

}

}

}

#Inlining and Linking a Square Table

Separator {

WWWAnchor {

name "file:///C:/WEBFX/VRML/TABLE.WRL"

description "A Square Table"

Transform {

translation 0 0 1

rotation 0 1 0 -.75

}

WWWInline {

name "file:///C:/WEBFX/VRML/TABLE.WRL"

bboxSize 1 .85 1

bboxCenter 0 .425 0

}

}

}

This puts our chair and table models together in a VRML environment (see Fig. 15.5). When we click on one of them, that model's .wrl file will be called up by our VRML browser (see Fig. 15.6). This lets us examine it by itself.

[Fig. 15.5](#)

A VRML scene using inlined objects linked to the WWWAnchor node.

[Fig. 15.6](#)

A VRML scene with linked objects using the WWWAnchor node.

From Here...

The WWWInline and WWWAnchor nodes are key components of the VRML 1.0 specification. They link your VRML environment to the Internet, letting you add models from other Internet sites to your environment or link to other VRML and Web sites. You can also combine these nodes to let users examine individual items in your world.

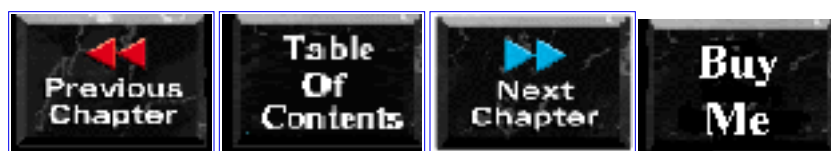
Now that you have set up all the main elements of your VRML environment, you can begin fine-tuning it. You can add different camera views and improve its performance. Most importantly, you will need to put your environment on the Web.

To learn more about VRML development, see the following:

- i [Chapter 16, "Setting Up the Camera,"](#) discusses how to use the camera nodes to establish multiple viewpoints in your VRML world.
- i If you want to make your VRML environment run more efficiently, see [Chapter 17, "Improving the Performance of Your VRML Environment."](#) It shows ways you can make your world load and render faster. It shows you how to use the LOD node.
- i [Chapter 18, "Opening Your House on the World Wide Web,"](#) looks at how to put your VRML files on the Internet. It outlines some of the tools and procedures you will use when transferring your .wrl files from your computer to your Web server.
- i [Chapter 19, "Proposed Additions and Changes to VRML 1.0,"](#) looks at how the upcoming VRML 1.1 and 2.0 specifications may affect the way you create your virtual worlds.
- i If you want to look at the experiences of others who have created their own VRML environments and models, see Part V, "Learning from the Pros: Sample VRML Environments."
- i If you want to see what programs are available to help you create VRML environments, see Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities." These chapters discuss how those programs can help you build and furnish your VRML world.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



16-Setting Up the Camera

When a VRML browser first loads a virtual world, it has to make a decision. Where does the user start out?

One option is for the browser to position the user at the origin (0,0,0). However, since there are often objects located there, it's not a prime location. The user would also be looking at everything from "ground level," which is probably not what you want. Alternatively, a browser could just arbitrarily choose a location based on the dimensions of the world; that works a little better, but again isn't going to be right in a lot of cases.

Ideally, the author of the world should be able to specify where the user starts out. Fortunately, VRML includes nodes for exactly this purpose; they're called "cameras."

In this chapter you will learn:

- What a camera is
- The difference between Perspective and Orthographic cameras
- How to specify a field of view
- How to decide where to place a camera

You don't have to be a good photographer to use a virtual camera, but it helps.

Cameras

The rendering engine inside every browser has its own "virtual camera," which the user is free to maneuver around the world as they see fit. A VRML Camera node doesn't restrict the user's freedom of movement in any way; it's simply a hint as to where the user should start out when they first enter a world.

A VRML file can have multiple cameras. Although the version 1.0 VRML specification doesn't spell out how this situation should be handled, the approach most browsers are adopting (and which will be recommended in version 1.1) is to use the first camera mentioned in the VRML file. In addition, some browsers allow a number of cameras to be listed inside a Switch node; this allows the browser to give the

user a selection of viewpoints. This feature is supported by (at least) the WebFX and WebSpace browsers.

A camera node can be assigned a name using DEF, which can then be referenced as the "entry point" of the virtual world. A URL that references the VRML file can have #camera_name at the end of it, meaning that the user should start out at the location and orientation of the camera with that name. Cameras can be placed in the object hierarchy just like any object or light, and are, therefore, affected by transformation nodes.

Camera Properties

A camera node has three basic fields. The first two are obvious: the position and orientation of the camera. The position is just a set of X, Y, and Z values. The orientation is in axis-angle form, as used throughout VRML.

The other field common to all camera nodes is the focalDistance. This is the distance from the camera, along the viewing axis, to the "point of focus." This is used by the "examiner" viewing mode that's found in many browsers, in order to give a point around which the camera should rotate.

Starting with VRML 1.1, camera nodes will also contain two new fields: nearDistance and farDistance. Objects will be "clipped" (as described in Chapter 7) to these two distances, so that anything nearer than nearDistance or farther than farDistance will not be seen.

Perspective Cameras

The most common type of camera is created by a PerspectiveCamera node. This node simulates the way we view the real world, using perspective (as described back in Chapter 11) to make nearby objects appear to be larger than farther-away objects.

A PerspectiveCamera has a viewing angle, through which it sees the world.

The viewing angle is stored in the heightAngle field; it's the total vertical angle, in radians, of the viewing frustum (see Chapter 7 for details). There is no way (in VRML 1.0) of specifying the horizontal viewing angle; generally speaking, the aspect ratio of the display window is used to compute the horizontal angle from the vertical one. For example, if the window has a 4 to 3 aspect ratio (width to height) then the horizontal viewing angle is 4/3 of the heightAngle value.

The viewing angle is related to the zoom factor. In fact, the zoom factor is the tangent of one-half the field of view; if the heightAngle is 0.5236 radians (30 degrees) then the zoom factor is $\tan(0.5236/2)$, or 0.2679.

Orthographic Cameras

The other type of camera node in VRML is the `OrthographicCamera`. It's very similar to the `PerspectiveCamera`, except that there is no perspective effect; objects appear to be the same size regardless of their distance from the viewer. This is completely unlike the real world, so you may be wondering what it's useful for. Well, truth be told, there aren't very many uses for it at all.

There's the odd case where it's useful. Let's say you built a topless restaurant (one with no roof—what did you think I meant?). If you simply position the camera above it, you won't be able to see everything; since you'll be looking down on most of the walls at an angle, some things will be obscured. If you use an `OrthographicCamera`, you'll be able to see everything clearly; all the walls will be seen from above, with no perspective.

The only difference in the `OrthographicCamera` specification is that instead of a `heightAngle` field, there's a simple `height` field. It measures the total height of the viewing volume; as with the `PerspectiveCamera`, the aspect ratio of the window is typically used to compute the width of the viewing volume.

Positioning Cameras

There are no real hard-and-fast rules for how to position cameras in your virtual environment. If you are building a world that corresponds somewhat to the real world, then human dimensions should be your guide; position the camera near one corner of a room, pointed towards the center of the room and perhaps tilted down slightly. The Y coordinate of the camera should be about the same as it is for a human being in the real world, about 1.6 meters.

Another approach is to look at photographs of real locations, and see where the photographer chose to place his or her camera. You can do the same thing while watching a movie, or a television program

Framing is an important issue. Compare Figure 16.1 with Figure 16.2:

[Fig. 16.1](#)

An example of bad framing.

[Fig. 16.2](#)

An example of good framing.

Summary

In general, don't bother with `OrthographicCamera`; `PerspectiveCamera` is the node you want to use. Leave the `focalDistance` and `heightAngle` fields at their default values. If you're using transformation nodes to position the camera, then you don't need to use the `position` or `orientation` fields. In other words, you should create a camera using the following node:

```
PerspectiveCamera { }
```

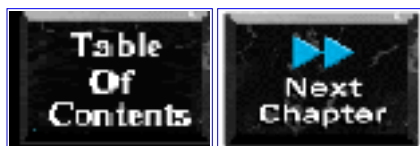
Pretty simple, really. Bear in mind that a lot of VRML browsers completely ignore the camera nodes. This is expected to change eventually.

From Here...

Now you know everything there is to know about using VRML. Almost. There's one more important area, which we'll look at in [Chapter 17: optimization](#).



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



17-Improving the Performance of Your VRML Environment

Well, by now you've probably surfed the Web and visited a few different virtual worlds. As you may have noticed, it's a pretty slow process; you click on a link in an HTML document, and it can take a very long time for the world to be transferred into your computer. Not only that, but moving around in the world is pretty clunky too; it's not smooth and continuous, but disjointed and jerky.

Of course, it's easy to just blame the hardware; once you have fiber optic cables into your home, and a multi-processor RISC workstation with special graphics acceleration hardware, everything should be fine. However, that's really a cop-out; the fact is that you can create interesting worlds that download and render at reasonable speeds on existing hardware. In this chapter, you'll learn how.

In particular, you'll learn

- Why careful design is important
- How to reduce the size of your VRML files, to minimize download time
- How to design your worlds for performance as well as appearance

Along the way, you'll also

- Learn how the Switch node works
- Learn how the LOD node works

So, let's begin.

Does Size Really Matter?

When it comes to VRML files, the answer is yes! If it takes users too long to download a world, they'll simply give up and abort the transfer. You'll have spent many long hours lovingly hand-crafting your world, and no one will ever see it.

Worse yet, they might download the whole thing and then find it's so slow to move around in that they

feel it was a waste of time. Since many users pay for their connect time, they'll send you lots of nasty e-mail about how pathetically slow your world is. That's the thanks you'll get for all your hard work!

So yes, size matters. So does rendering speed. If anyone tells you otherwise, they probably have an Onyx and a T1 line; they're in the minority, so you can ignore them. The people you (probably) care most about are the ones sitting at home with their Pentiums and their 14.4 or 28.8 kbaud modems.

Let's do a little math. Say your world is 200,000 bytes large; at 14.4 kbaud (the most common modem speed), it takes them almost two and a half minutes to download. If your world isn't worth the wait, they'll be pretty ticked off at you.

Making Smaller VRML Files

The following is a "Top Ten" list of things you can do to reduce the size of your VRML files. They're in no particular order. You should try to use as many of these techniques as possible, since the effect is cumulative; if you have one technique that reduces the file size by 25% and another that reduces it 10%, using both will take 32.5% off the size of your file and cut down the user's transfer time by a third!

Technique #1 GZIP

The GZIP compression algorithm takes an input file (in this case, a world built using VRML) and compresses it into a smaller form. It doesn't know anything at all about VRML, and can generally compress pretty much anything. VRML files compress particularly well using GZIP, since VRML is entirely ascii. There's also a lot of repetition, which GZIP can take advantage of to reduce file size; for example, there are long lists of numbers (vertex coordinates, normal coordinates, texture map coordinates, all the Index fields of the IndexedLineSet and IndexedFaceSet nodes, the color values in the Material node, and so on).

Not all browsers have built-in GZIP decompression yet, but they will soon. By using the GZIP utility on your VRML file, you can easily reduce file size by a factor of anywhere from 25% to 50%, a very worthwhile saving. The GZIP utility can be found on the CD-ROM that accompanies this book.

Technique #2 Inheritance

As mentioned in Chapter 11 and Chapter 14, VRML nodes "inherit" certain properties from their parent nodes higher up in the tree. That means that instead of storing that information once per node, you can often store it once and then inherit it down the tree.

For example, a Material table might be stored once for an entire hierarchical human figure. The colors of the arms, legs, clothes, skin, hair, and so forth only have to be stored (and therefore transmitted) once, and all the child nodes inherit that material table.

- Here's a list of the properties that get inherited:
- Materials
- Material bindings
- Normals

- Normal bindings
- Textures
- Texture coordinates
- Texture transforms
- ShapeHints settings
- FontStyle settings
- Coordinates

That last item is kind of interesting. When you create a set of coordinates (using the `Coordinate3` node, as described back in Chapter 11), those values get inherited by children of the object until over-ridden with another `Coordinate3` node. That means you could define coordinates once, and re-use them in a number of objects; for example, you could store the coordinates for your left arm, then create two `IndexedFaceSet` objects which both use those coordinates. The right arm would be scaled with a negative X value, in order to mirror-image the coordinates and generate your left arm. Listing 17.1 shows our old friend, the pyramid, instanced twice using this technique.

Listing 17.1

#VRML V1.0 ascii

Separator {

ShapeHints { vertexOrdering UNKNOWN ORDERING }Material {

diffuseColor [0 0 1, 0 1 0, 0 1 1, 1 0 0, 1 0 1]

}

MaterialBinding { value PER_FACE }

Coordinate3 { point [-1 0 -1, 1 0 -1, 1 0 1, -1 0 1, 0 1 0] }

Separator {

Transform { translation -2 0 0 }

IndexedFaceSet {

coordIndex [

0, 4, 1, -1,

1, 4, 2, -1,

2, 4, 3, -1,

3, 4, 0, -1,

0, 1, 2, 3, -1

]

```

}
}
Separator {
Transform { scaleFactor -1 1 1 translation 2 0 0 }
IndexedFaceSet {
coordIndex [
0, 4, 1, -1,
1, 4, 2, -1,
2, 4, 3, -1,
3, 4, 0, -1,
0, 1, 2, 3, -1
]
}
}
}
}

```

Inheritance allows you to avoid having to send essentially the same data multiple times.

Technique #3 Use USE

Back in Chapter 11, we learned about the DEF prefix. It allows you to define a name for a node, so that it can be referenced later.

One way of referencing it is with the USE keyword. If you DEF something, you can USE it later as often as you like. For example, you could create the complete geometric description of a spider's leg (materials, coordinates, textures, etc.) and assign it a name using DEF. You can then "instance" that leg eight times, each at a different location:

```

DEF Spider_leg Separator {
<full description of a spider's leg>
}
Separator {
Transform { translation 1 0 0 } # first right
USE Spider_leg

```



```

}
Separator {
Transform { translation 0 0 1 } # second left
USE Spider_leg
}
Separator {
Transform { translation 1 0 1 } # second right
USE Spider_leg
}
Separator {
Transform { translation 0 0 2 } # third left
USE Spider_leg
}
Separator {
Transform { translation 1 0 2 } # third right
USE Spider_leg
}
Separator {
Transform { translation 0 0 3 } # fourth left
USE Spider_leg
}
Separator {
Transform { translation 1 0 3 } # fourth right
USE Spider_leg
}

```

Notice that we only USE the leg seven times, not eight; that's because the initial definition of the leg creates the first instance. There's an enthusiastic debate going on within the VRML mailing list as to whether that really makes sense or not. A lot of people feel that there should be a way of creating a definition for something without creating an instance of it. That would certainly make sense in the case of our spider example.

You can simulate the effect of a non-instancing node by using a Switch node with a whichChild value of -1:

```
Switch {  
  whichChild -1  
  DEF Spider_leg Separator {  
    <full description of a spider's leg>  
  }  
}
```

The Switch node is exactly like a Separator, with one important difference: instead of rendering all of its children (shapes and Separators), the browser only renders one of them. Which of its children should be rendered is determined by the whichChild field:

```
Switch {  
  whichChild value  
  children...  
}
```

The value is a number. If it's zero, the first child is rendered; if it's one, the second child is rendered, and so on. If it's -1 (the default value) then none of its children are rendered.

VRML 1.0 allowed the value -3, which meant "render all the children." This has been dropped as of version 1.1, and should not be used.

Since a whichChild value of -1 tells the Switch node not to render any of its children, the leg is simply defined and not instanced. You would then USE the Spider_leg eight times.

It's very likely that future versions of VRML will treat the children of a Switch node as if they were inside Separators; for maximum compatibility, put a Separator around each child of the Switch.

Clever use of USE can reduce file size enormously; for example, if you're creating a forest of randomly placed (but identical) trees, you just define the tree once and then USE it repeatedly. You can scale the trees individually, so they aren't all the same size, and rotate them around their Y axis so they're oriented differently. You can even tip some of them a bit, if you like. If your definition of the tree doesn't have a Material node, you can even have the trees be different colors (as they would be during autumn, for example) by putting a Material node in front of each USE.

Listing 17.2 shows a small grove of trees, created using this technique.

Listing 17.2

#VRML V1.0 ascii

Generated by FOREST

Separator {

Switch {

whichChild -1

DEF tree WWWInline { name "tree1.wrl" }

}

DEF spring_material Material { diffuseColor [0.0 0.666667 0.0, _0.466667 0.466667 0.0] }

DEF autumn_material Material { diffuseColor [0.25 0.25 0.0,

_0.466667 0.466667 0.0] }

Separator {

Transform {

translation 0 0 20

rotation 0 1 0 1.078997

}

USE spring_material

USE tree

}

Separator {

Transform {

translation 10 0 10

rotation 0 1 0 3.363931

}

USE spring_material

USE tree

}

Separator {

Transform {

translation 20 0 5

rotation 0 1 0 3.265753

}

USE autumn_material

USE tree

}

Separator {

Transform {

translation 20 0 20

rotation 0 1 0 5.465932

}

USE spring_material

USE tree

}

}

Each tree USEs either "spring_material" or "autumn_material," and USEs the tree that's DEF'd near the top of the file. The first tree is inside a Switch node; this prevents it from being instanced.

Notice the comment "Generated by FOREST." FOREST is a simple C program I wrote to generate pseudo-random forests; it's described in detail in [Chapter 32](#).

Incidentally, the tree that gets WWWInline'd is included on the CD-ROM. It was produced by a commercial package called TreeTool; it's described in more detail in [Chapter 26](#).

Technique #4 WWWInline

The WWWInline node (described back in [Chapter 15](#)) is very useful for reducing file size. WWWInline lets you define a bounding box for the file it references; if the user never actually sees the bounding box (because of culling, or for any other reason) then the inline'd file might never download! This requires some careful thought and planning, but the savings in transfer time make it really worthwhile.

Technique #5 LOD and WWWInline

The LOD (Level of Detail) node (described in detail later in this chapter) can be used to make the WWWInline node even more powerful. By putting a WWWInline inside an LOD node, you can avoid transferring the inline'd file until the user gets close enough to see it. For example, you could create an entire city with each block of buildings in a separate file. If the user is more than (say) five blocks away from a particular block, that block doesn't need to be loaded. As they move towards a block, it will

eventually get loaded, but the user shouldn't notice this since they have the intervening blocks to look at along the way.

You can even have multiple versions of an object at different levels of detail, and only transfer them as needed. For example, if you're building a throne room for a castle, you could create ten different versions of the throne at various levels of detail. If the user is just running through the throne room (perhaps because something is chasing them...) they'll only see the simplest version of the throne. As they move closer, more and more detailed versions are loaded and displayed. If they never get very close to the throne, the more-detailed versions (the ones that are bigger and more expensive to transfer) never have to be downloaded at all!

Technique #6 Eliminate Duplicate Vertices

One of the most common causes of file bloat is vertex duplication. This is usually caused by translating objects from some other format which isn't clever enough to use common vertex lists. For example, consider a cube; there are only eight vertices (four on the top and four on the bottom). In some file formats (the RAW triangle format, for example) each vertex is stored once for each polygon it's in! The cube would have at least 24 vertices (6 faces times 4 vertices per face) and might have as many as 36 (6 faces times 2 triangles per face times 3 vertices per triangle). Since vertex lists make up a large part of your average VRML file, this is a real problem.

By merging identical vertices, you can reduce file size dramatically. In some cases, vertices may not be at exactly the same location (due to roundoff error), so you may need to specify a "closeness" or "tightness" or "threshold" value to let whatever merge tool you're using identify the common vertices. You can also take that opportunity to eliminate extra digits after the decimal point for each vertex; for example, if you have a vertex like

-7.2000043 2.0000005 12.0000004

You could convert it to

-7.2 2 12

and save a lot of file space and transfer time.

Unfortunately, there are currently no VRML tools out there that will merge duplicate vertices automatically. However, such utilities exist for other formats so it should be just a matter of time before there are VRML equivalents.

Technique #7 Don't Store Normals

VRML browsers are quite capable of generating normals automatically for most objects; unless you have a good reason to keep the normals around, you should try to get rid of them. The only reason you might need to store them is if they were generated by a modeling program from a higher-level representation based on something other than polygons; in such a case, the normals it generates are more accurate than the averaged ones produced by the VRML browser.

Since normals are often stored for each vertex, you can save a huge amount of space by dumping the normals. If the averaged normals are wrong, you may need to adjust the `creaseAngle` field in the

ShapeHints node; see [Chapter 14](#) for details.

If you must store normals, examine the file closely; you can often eliminate a number of digits at the far right of the decimal point (as discussed above under vertex elimination) without having any noticeable effect on the quality of the result.

Technique #8 Avoid Textures

Textures tend to be big, and big files mean long downloading times. On the other hand, you can save a lot of extra vertices and polygons by replacing a detailed version of a surface with a single texture-mapped polygon.

If you can get by without textures, or with very few of them, you'll earn the eternal gratitude of all those millions of users with 14.4 kbaud modems.

Technique #9 Use JPEG

If you absolutely must use textures, you should use JPEG. It offers far and away the best compression, and is the most widely supported texture format. The fact that it's a "lossy" format means it's much better-suited for things like photographs, rather than regular textures like checkerboards. For regular textures, you should consider using PNG (though not all browsers support it yet).

Technique #10 Tile Your Textures

Speaking of checkerboard textures... you can save a lot of space by using a small texture that gets repeated many times over a surface. For example, you could create a small 2 by 2 checkerboard pattern and repeat it four times over the surface of the board itself by careful use of texture coordinates, texture transforms, and the default REPEAT setting for wrapS and wrapT.

Figure 17.1 shows just such a checkerboard pattern; it was created using the Paint application in Windows.

[Fig. 17.1](#)

Four squares form the basis for our checkerboard.

This can then be used as a "tile" for a much larger texture, as shown in Listing 17.3.

Listing 17.3

#VRML V1.0 ascii

Separator {

Coordinate3 { point [0 0 0, 0 0 -1, 1 0 -1, 1 0 0] }

Texture2 { filename "cboard.bmp" }

TextureCoordinate2 { point [0 0, 0 4, 4 4, 4 0] }

```
Texture2Transform { scaleFactor 4 4 }
```

```
IndexedFaceSet {
```

```
coordIndex [ 0, 1, 2, 3, -1 ]
```

```
textureCoordIndex { 0, 1, 2, 3, -1 }
```

```
}
```

```
}
```

If you don't see anything right away when viewing the above world, move around a bit. In some browsers, the default view will be edge-on.

Notice that the Texture2Transform node is used to scale the coordinates from the TextureCoordinate2 node so that they lie in the range of 0 through 4; since the texture coordinates really only go from 0 to 1, and the wrapS and wrapT flags are left at their default value of REPEAT, the texture is repeated four times in each direction. Since the original texture was 2 by 2, the result is a full 8 by 8 checkerboard, as shown in Figure 17.2.

Combinations of Techniques

You can combine many of the above techniques very effectively. For example, let's say you have a complex control panel with many buttons, levers and switches. You create several 3D models of it at different levels of detail. You also create two additional models: one is a single polygon with a texture map of the panel on it, and the other is just a single one-color polygon.

[Fig. 17.2](#)

The tile gets repeated four times in each direction to create a checkerboard.

You store each of these representations in a different file, and have an LOD node which WWWInline's each of those representations:

```
DEF Control_panel LOD {
```

```
ranges [ 0.5, 1, 3, 5, 7 ]
```

```
WWWInline { name "geopan3.wrl" }
```

```
WWWInline { name "geopan2.wrl" }
```

```
WWWInline {name "geopan1.wrl" }
```

```
WWWInline { name "textpan.wrl" }
```

```
WWWInline { name "simppan.wrl" }
```

```
}
```

The most detailed panel ("geopan3.wrl") would be used if the user is less than half a meter (a foot and a half) away, the next most detailed if they were up to a meter away, and so on; beyond 5 meters (16 feet)

the panel would be just a texture-mapped polygon, beyond 7 meters (23 feet) it would be just a plain rectangle. It's quite possible that you'll never even have to download the texture, yet if the user gets really close they can see a very detailed rendering of the control panel.

Speeding Up Your Worlds

Once the user has the world downloaded into their browser, they'll want to start zipping around and exploring it. If you've designed your world well, that should be no problem even on existing hardware and beta-release browsers. If you've designed your world badly, even the fastest graphics supercomputer will be brought to its knees.

Here's a "Top Ten" list of ways to improve the rendering speed of your world.

Technique #1 Use Fewer Polygons

This is the single most important technique you can use. Every polygon increases the amount of work the renderer has to do; it has to do more lighting calculations, more transforms, more scan-conversion... it has an impact at every stage of the graphics pipeline. Reducing your polygon count is absolutely essential.

Technique #2 Use Fewer Lights

Every light source you add increases the amount of work that must be done for each vertex in the scene. More lights means more math, and more math means slower rendering. Eliminate as many lights as you possibly can.

Technique #3 Use Directional and Ambient Lighting

For reasons that will become clear in [Appendix A](#), the use of directional lighting is much less expensive (in terms of computation) than point or spot lighting. Ambient lighting is even cheaper. If you can get away with a single directional light source and some ambient light, your world will perform noticeably better than if you have a bunch of point sources.

Technique #4 Use Precomputed Lighting

If you know that the browser your users will be running has support for `PER_VERTEX MaterialBinding` using pre-computed lighting (a set of `emissiveColor` values and no `diffuseColor`, `ambientColor`, or `specularColor`), then you can create worlds where most of the objects don't need to have lighting calculations done. To produce such scenes, you may have to use special programs that do radiosity calculations; none are presently available, but some are in the works.

Technique #5 Use ShapeHints

If you know that all the faces in an object have consistent vertex ordering, and/or that the shape only has an "outside" (i.e., the inside surfaces are not visible) then use the ShapeHints node to inform the browser of that information as described in [Chapter 11](#). This allows the rendering engine to avoid lighting surfaces that won't be seen, and possibly do backface culling. The result is once again a noticeable performance improvement.

Technique #6 Subdivide the World

Use the WWWAnchor node (described in [Chapter 15](#)) to create "portals" to other VRML worlds. By partitioning your world into a number of sub-worlds, you can dramatically reduce the amount of data the renderer has to process at one time.

Technique #7 Use the LOD Node

The LOD node is similar to the Switch node that we discussed earlier, but instead of having a whichChild value it has a set of "range" values. If you're closer than the first specified range, the first child is rendered; if you're farther than that, but closer than the second specified range, the second child is rendered, and so on.

By clever use of LOD, you can achieve much greater rendering performance than you'd be able to otherwise. Every object in the scene can have a beautiful, high-resolution representationóso long as it also has a number of less-detailed versions that can be used when the user is farther away.

The LOD node looks like this:

```
LOD {  
range [ values ]  
children  
}
```

There can be as many values as you like, separated by commas. Each value corresponds to one of the children; they're ordered from nearest (most detailed) to farthest (least detailed).

Browsers are not required to treat the range values as actual distances; they can be scaled up or down depending on the scene complexity and the performance of the rendering engine. They may even change while you're browsing the world! Do not use the distances to trigger changes when the user approaches; this doesn't work in a lot of browsers, and definitely won't work well in the future.

LOD is a powerful technique that should be mastered by any serious world-builder. Currently, very few browsers support it (they just use the first, most-detailed version of the object) but that will change very soon.

Technique #8 Eliminate Duplicate Vertices

This was mentioned earlier, as a way of reducing file size. It also dramatically improves performance, since fewer vertices means less time spent transforming them.

Technique #9 Use Textures

There are often cases where using a texture map can eliminate a lot of extra polygons; that's usually a worthwhile trade-off, since polygons are expensive. Instead of creating a chessboard out of 64 polygons, use a single polygon with a checkerboard texture map.

Technique #10 Avoid Textures

Seems kind of contradictory, doesn't it? Texture-mapping is computationally expensive; using too many textures will chew up both memory and CPU time. Avoid the temptation to apply texture maps to everything; in the real world there are plenty of smooth surfaces that don't have any kind of texture at all. Use texture mapping cautiously; making intelligent decisions as to when to texture-map and when not to is a skill you'll acquire with practice. In the meantime, let the real world be your guide.

Advice to the User

Unless you have a graphics accelerator, you should try to keep your window size as small as possible. Every single pixel that the computer has to draw takes time; going from a 500 by 300 window to a 250 by 150 one can speed things up substantially. If the screen image is too small, consider switching to a lower-resolution graphics mode; if you're at 1024 by 768, go down to 800 by 600 or even 640 by 480. The image will be larger, but won't require any additional pixels; of course, you'll be sacrificing a certain amount of detail.

If you have 3D hardware acceleration, then image resolution is much less of an issue; in that case, scene complexity (and in particular, polygon count) are usually the most important factors affecting performance.

From Here...

Well, we've looked at 20 different ways of improving the performance of your worlds. You may not be able to use all of them, but you should seriously think about using as many as possible in order to create worlds that are not only beautiful, but functional as well.

At this point, you should think about making some of your worlds available on the Web, as Stephen describes in [Chapter 18](#). You may also want to look at worlds built by others, as described in [Chapters 20](#), [21](#), and [22](#). Finally, you can learn a lot by looking at some of the interesting worlds out on the Web; Stephen surveys them in [Chapter 23](#).

**Table
Of
Contents**

**Next
Chapter**

Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



18-Opening Your House on the World Wide Web

So now you have finished figuring out vertices, IndexedFaceSets, transforms, setting links, and configuring LODs. You have programmed an elegant, efficient VRML environment and you are ready to show it off. So now what do you do?

The short answer is, of course, that you need to put your VRML files on a Web server. This can be accomplished in two ways. The first is to set up your own Web server. This approach gives you complete control over the handling of your VRML environment. It also gives you a whole set of headaches and heartaches that you may rather avoid. The other approach is to find an Internet Service Provider (ISP) that will let you put files on to the Web. You have less control and may be subject to a number of restrictions, but you might sleep a little better.

This chapter discusses how you can put your VRML environment on the Internet. Some of the topics covered include:

- What's involved with setting up my own Web server?
- What do I need to tell my ISP?
- How do I get my VRML files from my computer to the server?

What's Involved in Setting Up Your Own Web Server

Whenever anyone thinks about having a Webpage or a VRML environment on the Web, her or his first thought is, "Why not set up my own Web server?" It's a good question. Certainly it is a tempting thought. You would be able to post as much material as you want to the Web. You could even invite your friends to put their HTML and VRML material on your server as well. It seems to be an ideal solution. And in the future, everyone's computer will act as a personal server on the Information Superhighway.

S-MOS SPC1500 Graphics Geometry Processor

The SPC1500 developed by S-MOS Systems was the first commercially available single chip 3D geometry accelerator for PCs.

In terms of money, the cost to set up a Web server can be astronomical. First, there is the question of equipment. While you might think that you could use your home computer to act as the server, you have to ask yourself, "Do I want to let people whom I don't know have access to the machine where I keep confidential records and personal material?" I imagine that your answer would be "no." So you will have to buy a separate machine that has the capacity to run the server, the software itself, and a modem. This will run at least \$4,000-\$5,000.

But the costs of the equipment and the software are only the beginning. You must then set up a dedicated phone line so that clients will be able to access your server 24 hours per day. Moreover, if you have a VRML environment that requires the transfer of large amounts of data (say over 100K), you will need to use a digital line. Depending where you live, an ISDN line can cost as much as \$500 just to set up and another \$500 per month (or \$6,000 per year) to maintain. (In larger centers, however, you can get one for as little as \$25 per month.) A T1 line can cost at least \$5,000 to set up and as much as \$3,800 per month to maintain. The start-up cost for a T3 line is \$15,000 dollars and over \$25,000 per month to maintain. And you will have to buy the equipment to connect your computer to these lines, and that is not cheap. An ISDN modem starts at around \$500. Routers can run in the thousands of dollars.

But even if your pockets are deep, there is no guarantee that you will be able to find a place to hook up your server to the Internet. You could try hooking it up to the LAN at your place of work or your school. But most Management Information Systems (MIS) directors, those people who operate the LANs, are not going to be very keen on letting you have uncontrolled access to their systems. Even if your MIS director is sympathetic to your cause, it is unlikely that she or he will give your server a connection without placing huge restrictions on it.

And even if you overcome all these hurdles, you will also have to maintain your server. And that takes time. A lot of time. You will have to watch for security breaches, fix protocol problems, deal with software crashes, and so on. In other words, you will have a new full-time job: taking care of your server. In the case of most Web servers, this job is taken care of by someone who gets paid. But most of us won't be able to maintain a server and make enough money to live on.

If you are a business or organization that wants to have a presence on the Internet, then these costs may be justified. But if you are an individual wanting to get your fifteen minutes of fame through the Web, you should look into alternatives to having your own server.

Putting Your VRML Environment on Someone Else's Web Server

At present, the more practical solution for getting your VRML environment on the Web is to find an ISP that allows you to set up Web documents on its server. The advantage of using this kind of arrangement is that most of the worries about the security and maintenance of the server are left to someone else. Moreover, the costs of maintaining the dedicated phone lines, equipment, and software are spread over a

larger number of users, and so the direct costs to you are smaller. The disadvantage is that your freedom to do what you want will probably be more restricted, and you will be at the mercy of your ISP. Nevertheless, for most individuals, it is much more practical than setting up your own server.

Looking for an ISP to be My Server What Should I Ask?

When looking for an ISP, you should consider a number of factors. First, you need to find out whether the ISP offers PPP or SLIP accounts. If they do not, then look somewhere else. You should also ask if they have SLIP or PPP software that has been preconfigured for their system. This will save you a lot of hassles when you are setting up. (For more about SLIP and PPP, see ["SLIP and PPP," Ch. 1.](#)) Windows 95 users should ask if an ISP provides a setup Wizard or its equivalent to connect with its service.

You also need to know the ISP's ratio of users to phone lines. There is nothing more annoying than to dial-in to your provider only to get a busy signal. The lower the ratio of users to lines, the better are your chances of getting through to your account. You will also want to find out what kinds of lines the ISP uses to connect to the Internet. At the very minimum, it should be using an ISDN line and this would be acceptable only if the ISP has a very small user base. A T1 or T3 line would be better.

You should also find out what types of modems they have available for their users. If you have a 28.8 Kbps modem but your ISP only has 14.4 Kbps modems to make the connection, then you will not be able to use your modem's full capacity. Also, make sure that the compression standard for your ISP is the same as your modem's. Otherwise, you will not be able to use the compression feature to make your downloads and uploads faster.

Once you are satisfied with the connections that the ISP will provide, you need to ask if they will set up an account that has read and write permission in the Web server's document and cgi-bin subdirectories. Some may give you this kind of access. However, it is more likely that the ISP will offer you a subdirectory and give you permission to manipulate files only within that subdirectory. This is fine. In fact, it shows that the system operator for that ISP is concerned about security, and that someone else cannot accidentally (or deliberately) fiddle with your files.

You should also check out your ISP's policy on downloading material on to the Web server. Because of concerns about "illegal" and pornographic material being posted on the Web, some ISPs are adopting a policy whereby files must be submitted to the MIS directory, and he or she will put the material on to the server for you. Although this seems to be an acceptable procedure, you will find that it becomes a huge annoyance whenever you wish to make changes. For example, I was involved with a journal on the Web based on the West Coast, but I designed its Webpages on the East Coast. Because the journal's server did not allow people to make remote TELNET connections, I wound up having to send my files to the server's administrator so that he could put the files on to the server for me. Whenever I wanted to make a change in the file, I had to send the file to him and wait for it to be put on the server before I could check to see if everything was working. It was an enormous headache to do things this way. Therefore, make sure that you will have direct access to your subdirectories on the Web server.

Looking for an ISP to be My Server How Much?

Once you've narrowed down the field of choices, you need to check the bottom line: how much will it cost? This is not always an easy question to answer. While ISPs may be willing to give Internet access to end users at cheap rates, their rates for letting you post an HTML document or a VRML environment may be much higher. You should check to see if there are additional costs for setting up a Web site over and above the ISP's regular access fee.

Another consideration about price involves how you are charged to maintain a Web page. Some ISPs charge a flat fee. Others charge a flat fee per Web page. This means that if you have a VRML environment connected to an HTML page with your resume, and another page with your picture and biography, you will be charged for having three Web pages. Others will charge you according to the number of client requests (hits) that are made to your Web site. These types of charges can also vary from provider to provider. Some will count only the hits on individual pages made by different users in a day; thus, the same client going back and forth between your VRML environment and your biography several times will be counted as only two hits. However, some ISPs will charge you according to the total number of hits per page. Under this scheme, a client who checks your VRML environment, looks at your biography, and then returns to the environment will be counted as three hits. Thus, while a flat fee may seem high at first, once you have calculated the charges you might incur when charged on a per-hit basis it may be cheaper in the long run to pay the flat fee. Always remember: caveat emptor.

I Have My Web Site Account, Now What Do I Do?

Now that you have your account, you need the tools to access it. The two pieces of software you need to accomplish this task are an FTP program (so that you can transfer your files from your computer to your account) and a TELNET program (so that you can work on them directly on the system). There are a number of FTP and TELNET programs that work over SLIP/PPP connections. The most popular FTP program for Windows-based machines is WS_FTP, a shareware program developed by John Junod (available at ftp://ftp.esra.net/pub/win3/ws_ftp.zip). Macintosh systems can use Fetch (<http://www.macatawa.org/~mthomas/ftp.html>) and OS/2 owners can use the New File Transfer Protocol Client (<ftp://hobbes.nmsu.edu/os2/unix/nftp051.zip>). For TELNET access, EWAN (a freeware program developed by Peter Zander) is available for Windows (<http://www.lysator.liu.se/~zander/ewan.html>). NCSA Telnet is available in a Macintosh version (<http://www.ncsa.uiuc.edu/SDG/software/brochure/overview/mactelnet.overview.html>). OS/2 systems can use Ckermit for OS/2 (<http://www.columbia.edu/kermit/ck0191.html>). Check to see if your ISP has copies of the software available in its connection software kit. Windows NT and Windows 95 users will find FTP and TELNET programs are already included with the operating system.

The very first thing you should do when you get your account and set up the software on your computer is to sign on and change your password. Most Web servers use the UNIX operating system, and so we will follow the procedure for that system. When you have connected to your SLIP/PPP account, run your TELNET program. The example we will use here is EWAN, but the procedure is the same for all TELNET software.

Begin by logging on to your account following the instructions given by your ISP. You should get a UNIX prompt indicating that you can begin typing UNIX commands. Type passwd and follow the

instruction to change your password. This process usually involves typing in your old password, and then typing your new password, and then typing your new password again to confirm that it is correct. Most systems will not allow you to see the password as you type, and so this confirmation process is necessary (see Fig. 18.1). Choose a password that combines letters and numbers to make it more difficult for someone to break into your account, but consider that you must be able to remember it without having to resort to a piece of paper. You also need to remember that UNIX is a case-sensitive operating system. Therefore, "theBigShot1" is different from "thebigshot1."

Fig. 18.1

Changing a password on the UNIX operating system.

Once you have changed your password, you can begin to set up your account to house your VRML environment. You can start by setting up a directory scheme to organize your information. You might place your texture maps, objects, and environments in separate directories. To make a directory, simply type `mkdir directory_name`, where `directory_name` is the name of the directory you want to create. Like typing your password, you need to remember to watch whether a letter in the name of a directory or file is upper- or lowercase.

Once you have created your directories, you will need to move through them and see what they contain. To move to the next branch down the directory tree, type `cd directory_name`. To move up one level in the tree, `cd ..` is what you need to enter. To return to your home directory, type `cd`. To move through multiple directories, you must separate each name with a forward slash (/). For example, `cd /Textures/Gifs/` will send you past the Textures directory into the Gifs directory.

You may also want to see what is in a directory. You can get a list by typing `ls`. This will give a list of the files and directories that are present. To get more detailed information (file status, date that it was created, etcetera), type `ls -l`. To check for any hidden files, you can use `ls -a`. If you want to remove a file, type `rm filename`, where `filename` is the name of the file you wish to delete.

The UNIX Operating system allows you to hide files by placing a period at the beginning of the filename: for example, `.cshrc` and `.login`. If you find you need more disk space, do not delete these files! They are there to make it easier for you to use UNIX.

The UNIX system allows you to access online manuals that can help you with any questions you have about using any of its commands. To access the manuals, type `man command`, where `command` is the name of the command you want to learn about. For example, to learn about the `passwd` command, you would type `man passwd`.

Getting Your Files from Your Computer to the Server

Once you have set up the directory structure for your VRML environment, you can begin transferring your files from your computer to the Web server. This is accomplished by using FTP. For the purpose of this book, and because I am a Windows user, I will use the WS_FTP software for my example. However, Fetch and New File Transfer Protocol Client follow similar principles. Figure 18.2 shows the main screen for the WS_FTP software.

There are other ways of transferring files from your computer to a server (and vice versa). The most popular method for computers using a terminal emulator is the Kermit protocol. However, I am focusing on using FTP through a SLIP/PPP connection because it is easier and because anyone accessing the Web is more likely to choose this type of connection over using Kermit.

Fig. 18.2

The WS_FTP main screen.

You will notice that the WS_FTP screen is divided into two identical parts. The left side is labeled "Local System." This is your computer. The right side is the "Remote System" which is the server. You will notice that the top window on both sides lists directories and/or drives, and the bottom window lists the files that are in that directory.

Once you have logged on to your SLIP/PPP account, you can start the WS_FTP software by double-clicking on its icon. It will begin by asking you for your user ID, your password, and the host name for your account. For example, my user ID is engl5105, my password is my shell account's password, and my host name is nexus.yorku.ca (see Fig. 18.3). If I were logging on to an anonymous FTP site, my user ID would be anonymous and my password would be engl5105@nexus.yorku.ca, which is the same as my e-mail address.

Once the connection has been made, you can begin transferring files from your computer to the server. You begin by moving to the directory that holds your files on your computer. With WS-FTP, you can navigate through the directory tree using the directory window. Simply point to the directory you wish to enter and double-click on it. If you wish to move up the tree, click on the two dots at the top of the list. Let's say that you want to transfer the VRML file thing.wrl from the directory C:\WEBFX\VRML. From the DOS root directory you would double-click first in WEBFX in the Local System's directory window, and then you would double-click on VRML. You follow the same procedure to navigate the directory tree in the Remote System's directory window.

Fig. 18.3

Connecting to a user account using FTP.

Once you are in the right directory, you can transfer files from your computer to the server. Decide which file you wish to send and choose the correct file type. For graphic files like GIF and JPEG, you must use the binary designation. For HTML, VRML, and Perl scripts, you need to use the ASCII designation. Since thing.wrl is a VRML file, you would choose ASCII. Once you have selected the correct file format, double-click on the filename in the file list. This will start the transfer process (see Fig. 18.4).

Fig. 18.4

Transferring a WRL file from your computer to the server.

When the transfer has been completed, you will see that thing.wrl appears in the file list of the Remote System (see Fig. 18.5).

Fig. 18.5

WS_FTP after the file transfer has been completed.

Repeat this procedure for each file that you want to transfer to the server.

Once the files have been transferred, you may want to rename some or all of the files to make them conform to the UNIX operating system's conventions. For example, the .htm extension used in DOS and Windows for HTML files should be changed to .html in UNIX. Files using the .jpg extension can be changed to .jpeg. Moreover, if there is a mistake in the filename, you will have to make the adjustment in UNIX. To change the name of a file in UNIX, type `mv old_filename new_filename`, where `old_filename` is the file's current name and `new_filename` is the name to which the former is to be changed.

Setting Permissions

After completing all the transfers and adjustments, the final task that you must do is define who can access your files. Do you want to limit access to users who use the same server, or do you want anyone connected to the Internet to have access? You should discuss this issue with your server's system administrator. She or he will help set up your account to conform to what you want. Typically, you would use a command like `chmod ogtr yourfile`, where `yourfile` is the name of your HTML file.

From Here...

Once you have found a Web server and have uploaded your VRML files to it, you are ready to invite the world to your "house." To learn more about the Internet and how it works, see the following:

- i [Chapter 1, "The Internet: Its History and How It Works,"](#) describes how the Internet got started, how it works, and how you can get on it. It also describes the history and workings of the World Wide Web.
- i The "Annotated Bibliography" lists books and on-line references on various subjects covered in this book, including the various languages of the Web.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



19-Proposed Additions and Changes to VRML 1.0

Throughout this book, references have been made to the ever-changing state of VRML. At first, it may appear to be a bit of a moving target; however, with the release of the 1.1 version of the specification, a lot of things are expected to be cleared up.

In this chapter, we're going to be looking at what some of the changes will be in 1.1; by the time you read this, the 1.1 version of the spec should be complete. When in doubt, refer to the spec; everything you read in this chapter is based on information that was current as of the end of October, 1995.

It's entirely possible that additional features proposed for future versions of the spec may make it in as early as 1.1; check Chapter 27 for a list of the new nodes in the works.

In this chapter, we'll examine

- Areas that have been clarified in the VRML specification
- Changes to VRML syntax
- Changes to Materials and Bindings
- Changes to Transform nodes
- New nodes and fields
- Changes to grouping nodes

Clarifications

There are a number of areas in which the original 1.0 version of the VRML specification was unclear or ambiguous. One of the goals of the revised version is to clarify those items.

Textures

The original specification did not indicate which image file formats should be supported. Since there are literally dozens of formats for storing a bitmap in a file (PCX, TIFF, BMP, etc.) it's important to specify which ones must be supported by all browsers; that way, world-builders can be sure their worlds can be seen by one and all.

The two standard file formats that have been proposed are JPEG (file extension ".jpg") and PNG. The JPEG standard was developed by the Joint Photographic Experts Group (hence the name JPEG), and it uses a "lossy" compression technique to achieve dramatic decreases in file size. PNG is the Portable Network Graphics format, designed as a replacement for GIF.

Why does GIF need a replacement? Because the algorithms it uses have been patented (yes, that's rightópatented!) and, therefore, its use is restricted. PNG suffers from no such legal constraints. PNG has an "alpha channel," which makes it useful for storing transparent texture maps.

LOD

The LOD (Level Of Detail) node is also being clarified. The term "center point of the LOD" is misleading; what's really meant is the point (0 0 0) in object coordinates. This point is transformed into world coordinates, and the distance between it and the camera position is used as the basis for range comparisons.

VRML browsers are free to adjust the distances specified in the range field as needed, in order to achieve improved rendering performance; they are guidelines only, not rules. It's therefore a very bad idea to try to "fake" behavior using them (things like doors that open as you approach them); that almost certainly won't work at all in many browsers. The range values will also typically be scaled by the current transform.

Browsers are free to read WWWInlines at any time, including WWWInlines inside of LOD nodes. See [Chapter 17](#) for more information about LOD and WWWInlines.

Scene Graphs

The scene graph is just a convenient way of storing the world in a VRML file; it is not usually maintained in-memory by the browser. This is important to remember when discussing object behavior.

Cube

Some people were confused by the use of "width," "height," and "depth" in the Cube node. The width is the size of the cube in the X direction, height is the size of the cube in the Y direction, and depth is the size of the cube in the Z direction.

WWWInline

The `bboxCenter` field is used with the `bboxSize` to define a bounding box around the referenced object, in that object's coordinate system. It is a hint, nothing more; it saves the browser having to load the referenced VRML file in order to determine whether it can be seen or not. The bounding box is, of course, affected by the current transform.

WWWAnchor

If object "A" contains a `WWWAnchor`, and is attached to another object "B" which also contains a `WWWAnchor`, and the user selects object A, it's object A's anchor that gets used. In other words, the anchor that's deepest in the hierarchy "wins." Another way to look at this is that if the object you select doesn't have an anchor, its parent is checked; if it doesn't have one either, the grandparent is checked, and so on. This is consistent with how "sensors" will work once behavior is added; see [Chapter 28](#) for more information about behavior and sensors.

MatrixTransform

People were trying to do some very strange things with the `MatrixTransform` node, and not all browsers were supporting them. The only valid matrices are ones that are the result of some combination of scaling, rotation, and translation. The final column of the matrix should always be 0 0 0 1, never anything else. (It's probably a good idea to avoid the `MatrixTransform` node altogether, and just use `Transform`; it's easier to understand and easier to modify.)

Syntax

There have been a number of changes and/or clarifications regarding the syntax of VRML.

MFFloat and MFString

Two node types were accidentally left out of the list in the original VRML spec; `MFFloat` is a multi-valued list of floating point values, and `MFString` is a multi-valued list of strings.

The # Character

Using a `#` character as part of a node name is a bad idea; many parsers will truncate the line at that point (since everything after a `#` is a comment). Strings are immune to this truncation, so a `#` inside of a quoted string is okay.

SFColor

The spec originally said the floating point values were in "standard scientific notation," which is a very odd thing to say. Colors are just floating point numbers, and are read the same way as every other floating point number in VRML; you should be able to parse it using the `scanf( )` function in C.

Field Names and Node Names

The spec didn't say so explicitly, but all field names are expected to begin with a lowercase letter, and all node names with an uppercase letter; `thisField` is a field name, but `ThisNode` is a node name. This is so that parsers can (if they choose) detect the end of the list of fields and the start of a node's children automatically.

If you create a new field or node (using the extensibility features of VRML) then you should add something like a suffix to the name based on your domain name, with the periods replaced by underscores. Something like `somehost`, `some site`, or `some domain` guarantees uniqueness.

For example, you could define a new class of separator called a `Region`:

```
Region sunee uwaterloo ca {  
  
isA "Separator"  
  
...  
}
```

The Header Line

The header will be changing, of course; it will now be

```
#VRML V1.1 utf8
```

The "utf8" refers to the character set being used.

URNs and URLs

Anywhere that a URL is allowed, a URN will be allowed. This should make life easier when it comes to retrieving commonly used shapes, textures, sounds, and so on.

Materials

Here we go. Materials are among the most confusing and ambiguous areas in the VRML specification. People often wonder why a particular scene doesn't look the same in any two browsers, and the answer (more often than not) has to do with materials.

There are several things that are being worked on in relation to materials. A standard way of computing single coefficients for the ambient, diffuse, and specular reflection properties will be spelled out. The lighting equation will be specified.

The whole complicated idea of material cycling is being dropped and good riddance!

`PER_VERTEX` and `PER_VERTEX` material binding is just plain strange, and a lot of browsers haven't implemented it at all. The recommendation for version 1.1 of the spec is that these bindings only apply to the `diffuseColor` and `transparency`, and in many cases even those won't be supported.

It will be possible to flag certain materials as "unlit," improving performance and allowing the use of precomputed lighting. The trick is to have multiple emissiveColor values and nothing else at all in the Material node. It's a bad idea to have both lit and unlit materials in the same object, since a lot of browsers can't or won't support it.

Transparency is a measure of how transparent a material is; the higher the value (in other words, the closer to 1.0) the more transparent the material is. A value of 0.0 for transparency means that it's completely opaque, and that's the default. It will be explicitly stated that transparency values other than 0 or 1 may not be used by all browsers; the transparency would be rounded to 0 or 1.

Cameras

If you put more than one camera in a VRML file, the first one will be the one that gets used. The others can be ignored, or (if they've been given names) they can represent alternate entry points into the world.

If you use DEF to give a camera a name, then #name can be tagged onto the end of an URL to take you to that entry point in the virtual world.

The focalDistance field of a camera can be used as a navigational hint as to how fast to move; it can also be used as a center of rotation for an "examine" mode in a browser.

Two new fields have been added: nearDistance and farDistance. They give the distance to the near and far clipping planes for the camera; I described the clipping process back in Chapter 11.

Scope of DEF'd Names

It is strongly suggested (and may eventually be required) that DEF'd names be unique within a VRML file.

Scope of Extensions

When you create a new node type or field, it remains defined for (at least) the entire file; for example,

```
RegionEXT {  
isA Separator  
fields SFString name  
name "someplace"  
...  
}
```

```
RegionEXT {  
name "someplace else"
```

...

}

There is no need to have an "isA" or "fields" field in subsequent RegionEXTs.

New Default Values

Starting with VRML 1.1, the ShapeHints node will have a default value of COUNTERCLOCKWISE for vertexOrdering, and a default value of SOLID for shapeType. This will make rendering much more efficient, since backface removal is on and backface lighting is off.

Hints

There are two new ways of specifying hints. One is with a NavigationHints node, which gives the translationSpeed and navigationMode (which can be WALK, FLY, EXAMINE, or DEFAULT).

The other type of hint is a named Info node. The use of these named nodes is being discouraged; all of them will have equivalent nodes in future versions of VRML. The names that are being used so far are as follows:

BackgroundColor

A string containing three floating-point numbers separated by spaces, giving the red, green, and blue components of the background color. This should eventually be replaced by a proper node, or perhaps made part of the proposed Environment node.

CollisionDetection

A string containing either "TRUE" or "FALSE" to indicate whether collision detection should be enabled on a particular object. Should eventually be replaced by a proper node with a field of type SFBool.

Title

A string that is displayed in the VRML browser's title bar. Will be replaced by the WorldInfo node.

SceneInfo

A string giving information about who created the VRML world, when they did it, copyright information, and so on. Will be replaced by the WorldInfo node.

Viewer

A string whose value is "walk," "fly," or "examiner." Replaced by the NavigationHints node.

ViewerSpeed

A string containing a single floating-point number that gives the viewer's walking (or flying) speed in meters per second. Replaced by the NavigationHints node.

Grouping Nodes

Big changes here. Several nodes had "leaky state," meaning that changes inside them (such as new materials, textures, etc.) would "leak out" and affect nodes specified later in the file.

- Two nodes that did this strange thing (Group and TransformSeparator) are being dropped entirely and should no longer be used.
- Switch nodes and LOD nodes used to be "leaky" in version 1.0 of the specification, but are leaky no longer. Each child of a Switch or LOD now behaves as if it were surrounded by a Separator node.
- WWWInlines have never been "leaky." Thank goodness.
- Switch nodes used to be able to select all their children at once. This is very hard to implement properly, and of limited use; it's being dropped.
- The syntax for Separators is being changed slightly, to make it clear that a Separator is just another way of saying "object." The details of this are not yet clear, but they're being worked on.
- The issue of light stopping at Separators remains controversial. It's been suggested that the specification should at least point out that many browsers will not implement this particular "feature."

That's It!

It may seem like there have been a lot of changes, especially considering it's a small increment in the version number! However, take heart; the changes in subsequent versions will be fewer in number, and will mostly be additions rather than alterations. The changes in going from 1.0 to 1.1 are mostly to solve some early teething problems which would otherwise have impeded the progress of VRML.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



20-What Is VRML About?

by Jeff Sonstein

After having nothing but Bernie and my voice for the last nineteen chapters, we thought that you should hear from other people about their experiences in creating VRML environments. The first of these is Jeff Sonstein, the keeper of the vrmLab site at New College of California. Jeff is an active member of the VRML mailing list and has a great deal of experience in creating VRML environments.

In this chapter, Jeff discusses the following:

- His approach to planning and creating the vrmLab
- Problems that you may encounter while creating your VRML environment
- How to scale your objects

VRML Is Not About Hardware or Software; VRML Is About Imagination

When I talk with people about creating VRML, often the first questions they ask are about what hardware they should get. If they don't ask about hardware, then they want to know which software package they should buy to begin learning about VRML. People joining mailing lists about VRML almost immediately ask questions about graphics cards and the merits of this \$5,000 modeling package over that one. I get some very blank looks at my response when asked how to learn about VRML. I tell people that VRML is not about hardware or software; VRML is about imagination. I tell people to get a metric tape-measure, sketchpad, and a good ASCII text editor, and to look around, think, make messy drawings, and measure things.

Head Mounted Displays and VRML

There are both good and bad results from Hollywood's discovery of "VR on the Net." The good news is that many more people have been exposed to the idea of three-dimensional graphical representations; information and ideas presented using form and space. The bad news is that Hollywood has dealt with

the problems of creating such representations as problems of hardware. Just use faster processors and more memory and wider bandwidth, and "VR on the Net" magically becomes a reality. The good news is that more imaginations are thinking about VR, and the bad news is that those imaginations have been focused on things rather than on ideas.

"Hheads-up Displays," "Total Immersion Suits," and so on; these things are flashy, but have little or nothing to do with VRML. Indeed, the fact that they are not needed is part of what makes VRML so powerful and the interest level so explosive. If VRML required flashy and expensive hardware, then nobody but hackers would care, and nobody but hackers at research center and film studios would have the hardware to actually do anything. A big part of the power of VRML is that anybody can compose and view VRML on what I think of as "Jane Everybody's Machine." If you write things which require a supercomputer on your desktop and unlimited bandwidth, then no one will ever look at it but you. Thankfully, there is no need to write such code any more (although some people still do).

I try to keep the mythical machine used by "Jane Everybody" in mind when I compose, and I recommend you do the same. In my imagination, this machine is a simple 486 with 8 megabytes of memory, sitting at the end of a telephone line with a 14.4 Kbps modem, and running Microsoft Windows 3.1. This is a powerful enough machine to be able to display and allow navigation through VRML scenes, and it is also now a cheap enough machine that millions of people have access to one somewhere in their lives. Very good VRML can be written with this target audience in mind, and no suits or gloves or helmets are needed for them to see what your imagination creates. I think "Rule One" for the VRML artist should be: "Make it so anybody can see it" (right after "Keep It Simple, Stupid").

So what's the problem with suits and goggles and helmets and such? Nothing really. You can probably find a decent "Hheads-up Display" for less than \$1,500. So why not design VRML from the start to use this technology? Well, these sorts of things can cut people off from their "surround." If I must put on goggles or a helmet to see something, then I must accept being isolated from what is going on around me in the "real world." People do not want to be cut off from what is going on around them. People want to be able to answer the phone and notice their boss peering over the cubical wall, and see the kids getting the scissors out of the drawer; people want to feel connected to their environment, not isolated further. Don't worry about fancy hardware when you are thinking about composing VRML; focus on imagining a coherent environment for people to visit and explore, and one which helps them feel more connected with the world somehow. VRML which cuts people off from their environment or doesn't seem to do anything but sit there looking "pretty" is, to my mind, a failure.

The VRML Composer's Eyes

Good VRML composition requires not just using your imagination, but bringing multiple types of imagination to bear. I think of these different perspectives as the videographer's eye, the builder's eye, and the ad executive's eye.

The composer must think about things like a videographer, for good VRML is not a static snapshot of a scene or a "structured architectural walkthrough." The videographer's eye is what helps a VRML composer to think about how things are initially presented to the viewer, and to think about the likely places people will move to when looking around in the scene. Make no mistake about it, people will move about and look at things from perspectives other than just the "opening shot" you set them up to see

at first. You should make a habit out of looking at scenes and objects you create from below, above, and behind. You should set the scene or object "spinning," and think about how it looks from all angles. The videographer's eye is about things like perspective and motion.

I once worked on a commercial VRML site with a person who had lots of modeling experience. The problem was that he was not used to thinking about any movement happening through a scene outside the constraints of a pre-structured or forced-path "walkthrough." This was an outdoor scene: many blocks forming a "floor" for a section of a city, several buildings, and a small park. There was only one building in the entire scene which was not solid and which could be flown into. Of course, the first thing I did when I went through the scene was to fly into that one building and look up at the inside of the domed ceiling. There were no inside surfaces (faces) to the dome, and the backs of faces are invisible in VRML. When I looked up to see the inside of the dome, there was of course nothing there: the dome had disappeared. When composing VRML scenes, thinking with the videographer's eye includes thinking about what perspectives and vistas will be naturally attractive and what you can do to make those perspectives "work right."

The composer must think about things like a builder, for good VRML is not random collections of unrelated objects. The builder's eye is what helps a VRML composer think about whether the objects and arrangements of objects in a scene "look right" together, form a coherent whole, and express some theme. It is my experience that the "best" VRML scenes tend to happen when this is the first perspective taken. If I have a theme and one or more metaphors in my head, then the visuals and the focal points come much more easily to me. Developing one idea at a time is difficult at best without a framework, and the builder's eye focuses on first the framework and then the details.

The composer must also think about things like an advertising executive, for good VRML grabs the attention of the viewer right away and then keeps it focused on what is important to the composer about the scene. What do you do when you click on a link in your HTML-browser which just leads to a huge graphic, and you suddenly realize you're going to have to wait for three hours while it downloads? You click on the STOP button and go somewhere else, of course. There are lessons from our experiences with HTML which we can apply to VRML. The ad executive's eye is about reminding myself that I am trying to "sell" an idea or a set of ideas to a client. I must first get his or her attention. After I have the attention, I must present him or her with images and metaphors which communicate clearly a specific range of ideas. I must make him or her want to explore my scene and want to get involved in the ideas expressed there. I must continue to hold attention where I want it to be for the "life" of the session. The ad executive's eye helps me do this.

Theme and Metaphor The VRML Composer's Primary Tasks

The vrmLab at New College of California is aimed at being a virtual laboratory for exploring some of the possibilities of VRML. The basic theme I chose for my first scene was building a "Temple of Knowledge (see Fig. 20.1)," the secondary theme which immediately suggested itself was the vrmLab's logical location as an informational nexus on "The Web," and (because I work in a Library) the first metaphor which suggested itself was that of a Reference Librarian's office "in" the Internet. I think "Rule Two" for the VRML composer (right after "KISS" and "Remember Jane Everybody's machine") should be: "Build

the forest first, then worry about the pine needles." In other words, first build the central theme, and then add the details and the props to support that theme (for more about planning your VRML environment, see ["Laying the Foundation," Ch. 10](#)).

Fig. 20.1

The Temple of Knowledge: embedded in the Web.

vrmlab's Temple of Knowledge

The Temple of Knowledge was the basic theme, so I built it first. The Temple itself is a simple structure; it doesn't "do" much of anything, but it is a central visual element for the scene (see Fig. 20.2). The Temple ties things together into a visual whole, provides a "frame" to contain and to delimit the "office," and serves to draw the eye to where I want the viewer's attention.

Fig. 20.2

Shot of Temple building alone. The Temple building provides a frame of reference.

Being a basically lazy person, I first went "cruising" through sites on the Internet looking for a Temple someone had already built. I found several, and some of them were very nicely done. But all of them were just too "big." What I mean by "big" in the context of a "virtual space" is not the same as what I mean in the context of a "real space." The "size" of a building in a real space is a concept linked with height and width and depth; the "size" of a building in a virtual space has to do with the number of characters in the file describing that building: how fast can you get it from a server to a browser? how long does it take before the viewer actually sees something?

I envisioned the VRML Reference Librarian's office as looking in some ways just like a regular office, with a desk and books and a lamp and some shelves and so on. This being a "virtual" space, I also envisioned the office as having elements which would not likely be found in a "real" Library, like drawers and doors which lead to other places and books which do things and columns which speak and so on.

Props

Props are very nice things, and I believe they should do things. The Reference Librarian's Office at the vrmLab has a number of ordinary-looking things in it which offer cues to people that they are indeed "in" an office (when they are actually staring at a painting on a TV screen). There is a reference desk which looks like a standard Library study-table because I used the "tape-measure and sketchpad" approach to build it. On the desk are things that look like big books. Viewers can click on a book and go to a World Wide Web search service (currently Yahoo, Lycos, or WebCrawler). They can click on the television sitting on the desk and go to an HTML page giving information about New College (in essence, a commercial for the college). If a viewer wants to send me e-mail he or she can click on the giant pencil. Viewers can get information about VRML by (surprise) clicking on the question-mark. The theme is reinforced both by the forms (that of common office items) and the functions (books leading to reference sources, a TV leading to a commercial, and so on) of the props in the vrmLab scene. First I

conceptualized and built graphical representations of my central themes, then I built the supporting objects and structured their groupings.

I think attaching meanings to props in VRML scenes is an important way to engage the viewers and sustain interest. People like to explore, and they like to find things which do something or inform them. One of the things which makes VRML interesting is the composer's ability to create objects which are familiar, but do unfamiliar and fantastic things. The file cabinet in the vrmLab has two drawers, one of which takes you to another virtual space when you click on it and one of which just gives a prompt about a planned experiment. What is useful about the file cabinet as a prop in this scene is just that it "fits" somehow, and that people automatically think to point to it to see if it does anything. To reinforce the message that the viewer should pay attention to the file cabinet, it is the first thing that shows up inside the Temple office space.

I know a lot of people who write VRML scenes like to cram a lot of "geometry" into each file their server has to send out. Sysadms hate to have their machines serving up lots of smaller files. I know because I am the sysadm for New College of California. All that aside, I sometimes use lots of smallish files to reinforce or sustain an idea or a theme or to focus attention where I want it to be.

Building an Informational Nexus

The secondary theme of the vrmLab scene is that it is an informational nexus on the Web, and the metaphors mostly focus on the Reference Librarian's office. The vrmLab scene is designed to bring your attention to the file cabinet, the books, and so on. The load order of the scene reinforces the points where I want your attention to go. First you see the frame (the Temple itself), and then you move your eyes from left to right across the "interior" of the office: file cabinet, then reference desk. The next thing I want you to pay attention to is the spiders. I use them as links to "generic" HTML files like browser company homepages and so on. The eye is prepared for this new focus by being moved towards the "front" of the office as the giant pencil and the question mark load, and then the spiders begin popping into view from the foreground towards the Temple. First the viewer's eye is drawn to the Temple structure in the middle, then from left to right as the office loads, then from the rear towards the front, and then suddenly to the immediate foreground.

Here is what the VRML file which describes the vrmLab looks like. Notice how the central theme is built first and then reinforced through office props, how the construction the spiders (my metaphor for the Web) gathering at the vrmLab brings your eye back to the office, and how anchors and their prompts are used to attach meanings to objects:

Listing 20.1

#VRML V1.0 ascii

vrmLab home.wrl

Separator {

PointLight { # upper center, back towards camera

on TRUE

intensity 0.1

color 1.0 1.0 1.0

location 0 100.0 40.0

}

PointLight { # upper right, back towards camera

on TRUE

intensity 0.1

color 1.0 1.0 1.0

location 5.0 100.0 40.0

}

PointLight { # lower center, back by camera [fill]

on TRUE

intensity 0.1

color 1.0 1.0 1.0

location 0.0 0.0 40.0

}

PerspectiveCamera { position 1.0 1.75 41 }

DEF BackgroundColor Info { string "0.847059 0.847059 0.74902" }

Separator {

WWWAnchor {

name "http://www.newcollege.edu/index.html"

description "New College of California top-level html Document is here..."

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Buildings/temple.wrl" }

}

}

Separator {

Transform { translation 6.25 0 20.0 }

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Icons/3Dcursor.wrl" }

}

Separator {

Transform { translation -5 0 4.5 }

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Buildings/stairs.wrl" }

}

Separator {

Transform { translation -3 0.45 -4 rotation 0.0 1.0 0.0 0.79 }

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Shelving/filecab.wrl" }

}

Separator {

Transform { translation 0.25 0.0 -2.5 }

WWWInline {

name "http://www.newcollege.edu/vrmLab/Warehouse/TablesAndChests/_c_table.wrl"

}

}

Separator {

Transform { translation -0.75 0.9 -3 scaleFactor 2 2 2 }

WWWAnchor {

name "http://beta.yahoo.com"

description "Click on this book to search Yahoo..."

DEF refBook Separator {

WWWInline {

name "http://www.newcollege.edu/vrmLab/Warehouse/BooksAndPens/_aBook.wrl"

}

}

}

}

Separator {

Transform { translation -0.525 0.9 -3 scaleFactor 2 2 2 }

WWWAnchor {

name "http://www.lycos.com"

description "Click on this book to search Lycos..."

USE refBook

}

}

Separator {

Transform { translation -0.35 0.9 -3 scaleFactor 2 2 2 }

_WWWAnchor {

name "http://www.webcrawler.com/"

description "Click on this book to search Webcrawler..."

USE refBook

}

}

Separator {

Transform { translation 0.25 1.25 -3.25 rotation 0.0 -1.0 0.0 0.5 }

WWWAnchor {

name "http://www.newcollege.edu/vrmLab/index.html"

description "vrmLab top-level html Document is here"

WWWInline {

name "http://www.newcollege.edu/vrmLab/Warehouse/ComputerEquipment/_monitor.wrl"

}

}

}

Separator {

Transform { translation 4.5 0 -6.5 }

WWWAnchor{

description "The University College of Cape Breton V.R. Lab"

name "http://vrpc2.uccb.ns.ca"

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Shelving/shelves.wrl" }
}

}

Separator {

Transform { rotation 0.0 -1.0 1.0 0.5 scaleFactor 0.25 0.25 0.25 }

WWWAnchor {

name "mailto:jeffs@ncgate.newcollege.edu"

description "email a note to the author..."

WWWInline {

name "http://www.newcollege.edu/vrmLab/Warehouse/BooksAndPens/pen.wrl"

}

}

}

Separator {

Transform { translation 3 0.5 4 scaleFactor 0.1 0.1 0.1 }

WWWAnchor {

name "http://www.virtpark.com/theme/vrml/"

description "vrml-charged vrml documentation may be found here..."

WWWInline { name "http://www.newcollege.edu/vrmLab/Warehouse/_Icons/helpicon.wrl" }
}

}

html references

Separator {

Transform { translation 1.5 0 30 rotation 0 1 0 0.75 }

WWWAnchor {

name "http://www.intervista.com/"

description "InterVista... the WorldView wizards"

DEF html1 Separator {

WWWInline {

name "http://www.newcollege.edu/vrmLab/Warehouse/Creatures/_htmlSpider.wrl"

}

}

}

}

Separator {

Transform { translation -1 0 30 rotation 0 1 0 0.5 }

WWWAnchor {

name "http://www.sgi.com/"

description "SGL... the WebSpace warlocks"

USE html1

}

}

followed by a number of additional spiders

starting in the immediate foreground

and leading back to the Temple

and up the stairs

}

Sizing Everything Up

Notice that the reference books are oversized, the pencil is gigantic, and the spiders are way too big. As I said before, one of the key elements which makes VRML so attractive to people is the composer's ability to express fantasy elements. The reference books are oversized because people couldn't differentiate between each of them in this context. Viewers kept getting confused and treated the three books as one object. I scaled them up in size and separated them more than they would have been on a "real" reference desk.

The pencil is gigantic because people couldn't tell what it was when they first entered the scene. Only when a viewer flew into the office and up to the desk (where the normal-sized pencil started out) could

they tell what it was and that perhaps it was worth investigating. I wanted people to notice the pencil and to have an easy time figuring out how to e-mail me a comment, so I finally scaled it way up in size and stuck it in the middle of the room.

The spiders would be positively frightening if I met one of them in "real" reality; each is around a meter in length. The function of the spiders is that of an icon: they stand for a link to an HTML file somewhere. They need to be big, so they are. The spiders are the only thing in the scene which looks like they might be "alive," so a feeling of motion towards the Temple is engendered both by placing them in a "vee" formation leading to the Temple and up the stairs, and by having them load from the foreground towards the building.

Fix It or Throw It Away

If you feel that something isn't working (that is, if it isn't instantly obvious to someone what it is or does), my advice is fix it or throw it away. Sometimes there is a great idea for a wonderful metaphor, and it just doesn't work right. When that happens, I have to recognize when it is time to stop trying to fix it and time to throw the idea away (at least for a while). An example of this is my attempts at placing the vrmLab within a terrain to represent the landscape of the World Wide Web. Several different approaches were failures for one reason or another, so I put the idea aside for possible future development.

I wanted to put some sort of a "floor" or "ground" underneath the vrmLab scene. Viewers like having a ground from which to understand figures, and from there is a real sense of "place" in a real "Web" that I wanted to capture visually. My first attempt was a terrain, and it worked but was unusable in this context. I figured out how to use a modeling package to generate terrain in a more or less controlled fashion, convert it to VRML, and simplify its geometry enough to make it a reasonable file size and a reasonably small number of vertices. My terrain consisted of a low range of hills in the background, and a medium-sized mountain with a mesa or butte on top. I placed the vrmLab scene on that mesa. I thought it looked great and it didn't slow navigation down too much on a PC, but I got lots of reports that it brought fancy graphics workstations "to their knees" when people tried to move around in the scene. It worked right on some machines but not on all, so after a while I took it out.

Next, I figured I'd try putting an actual giant spiderweb underneath the scene as the "ground." Dave Kurtz (davek@thoughtport.com) created and donated the spiders, so I asked him to build a web. I tried out several webs, both ones Dave created and ones I created, and none of them "worked right." When they had enough complexity so they looked right, they were such big files full of so many vertices that they slowed the loading process down terribly. When they were simplified enough so they were reasonably sized files containing reasonable numbers of vertices, they looked terrible. My next experiment with this will use the IndexedLineSet node, but that's a future story.

You can learn more about using IndexedLineSet node in ["VRML Standards and Practices," Ch. 11](#), and in ["Putting Up Walls," Ch. 12](#).

The Videographer's Eye Camera, Lights, and Composition

The opening shot, the first view of the scene that a person sees, may be the most important and most powerful moment in someone's experience of your scene. There should be a way for the viewer to "grasp" something of what you are trying to convey in one glance. The opening shot should play on the things the mind does to get distance information to achieve a "three-dimensional" effect. The opening shot should lead your eye into focusing where the composer wishes.

Perspective, presence, nearness, touchability—all these terms have been used to talk about the elements which make a two-dimensional picture seem to have three dimensions.

When seen from above, the scene seems "flat" and lifeless. From this perspective there is no near and no far. Nothing seems "touchable," the scene feels "wrong" somehow to a viewer. When seen from the side, the scene seems a "jumble" and incoherent. The groupings are not clear and the theme doesn't stand out on its own, objects seem out of reach somehow behind the columns; again the scene feels "wrong" somehow. These views will all work, but only if they are introduced to the scene from a perspective which reinforces the images, themes, and groupings you intended.

One powerful organizing technique in laying out the opening shot for your scene is what Ansel Adams called the "Near-Far" approach. In the normal opening shot of the vrmLab, there are objects in the immediate foreground which lead off into the middle distance to the Temple. Because the spiders are all the same "size" and shape but get smaller as the formation recedes from the foreground to the steps of the Temple in the middle distance, they help lend a sense of perspective, scale, and "realism" to the scene.

Details, Details

Some people go nuts putting textures onto everything to try to make their scenes photorealistic; some people drive themselves crazy getting the layout drive and geometry in a scene to exactly match the real place they are modeling down to a million decimal places. This is a basic error in thinking, both because people like the fact that VRML is not "real real" and because it makes for very large and very slow scenes (which turns people off). A few hints can go a long way. I needed a simple bed for a scene; nothing fancy, not a central prop or an important role, just a simple bed. I made the bed out of "primitives"; basic geometric shapes understood by VRML, expressible in a few characters, and thus, fast to download, and capable of being rendered very quickly by VRML browsers (which are optimized for the rendering of primitives). With a simple and small woodgrain graphic texture "painted" on to its surfaces, it seems realistic enough. I could have precisely modeled a real bed with all of its curves and complex geometry. I could have placed high-resolution textures onto its surfaces, and I could have made it look "really real." The problem is that then the bed would be so big and slow to download and render, that it would dominate the scene. My simple bed was more than sufficient that I made with just a few characters.

Controlling the Gaze

When people look at something they do not stare fixedly at one point; they scan the field of vision or object. You can take advantage of where and how people tend to scan things to ensure that they see what you want them to see in more or less the way you want. One composition technique photographers use is called the "Rule of Thirds." Look at this view (see Fig. 20.3) of the vrmLab, with lines added top to bottom cutting it in thirds vertically and lines added left to right cutting into thirds horizontally:

[Fig. 20.3](#)

Opening shot of vrmLab with lines added, visualizing the Rule of Thirds.

Part of why the stairs and the admittedly small spiders climbing them stand out in the overall impression a viewer gets from this scene is because of their placement. In the vrmLab's opening shot, the stairs and the spiders climbing them form a more or less vertical "line" about a third of the way in from the edge. Notice where the line the Temple roof forms draws your eye, and which spiders in the foreground stand out the most in your memory when you look away from the scene. You can reinforce the division of the scene into sectors by these "lines" and through the placement of objects along them.

Lighting

Did you ever get sick, and spend time sitting in front of the "tube" watching soap operas? You know, the TV programs with telephone conversations which span the decades? Take a look at the lighting they use sometime. The basic "no frills" lighting pattern for video is: one light set high and to one side of the camera (the "key"), one set low and to the same side or centered (the "fill"), and one set high and centered. This gives you enough shadowing so that objects have depth, yet not so much that it looks weird. Much the same lighting can be used effectively with VRML.

All browsers do not deal with lighting effects in the same way at this point in the development of VRML. You need to resign yourself to the fact that what looks properly lit in one browser will look dim in another and too bright in a third. What can I say? Pick your audience, think about what it is likely to be using, and then fake it. Use lighting to reinforce your audience looking where you want it to when the scene first opens. Make sure your lighting helps people see the details later on, too. Remember to put the places you want to tempt them to explore in the scope of some light. It does no good to have a beautiful object nobody can see.

Exteriors and Interiors

It also does no good to have an exterior opening shot of a beautiful "outdoor" scene containing buildings and rooms which tempt one to enter them, and then leave the viewer struggling to get inside. If you must combine interior and exterior elements in your scene, be careful that viewers can navigate easily from one to the other. If the opening shot puts the viewer "outside," most browsers will respond to navigational commands in-scale with being outside. When you are in a large space, it makes sense for a small mouse-movement to result in a relatively large (say, 5 meters) camera movement. The problem arises when trying to enter that beautiful office space: a small mouse-movement to get viewers "square

on" to the door or window they wish to enter through will still result in that large camera movement, and so they will have a hard time getting inside. I know, that they can just "fly" through the walls (not all VRML browsers have "collision detection" yet) if they wish. However, watch "just plain folks" using VRML browsers under this circumstance, and you will notice that they automatically try to enter through doors and windows. It can be a frustrating experience for viewers to be able to see the place they wish to get "into" but be unable to navigate through the doorway.

Because of the effects one can achieve, It is worth restating some things about load order and load balancing. The concern with load balancing is how many files the server is having to "deal up" to client browsers. If you put everything into one file or just a couple of files, then the server will have to "work" less and the clients and the server will spend less time negotiating connections and deciding if files need to be downloaded. The tradeoff is that the viewers will have to wait until everything has been downloaded before they will see anything. If you break things up into a swarm of small files, then the server will spend more time negotiating and coordinating with the client browsers and deciding what to do. The tradeoff for this extreme is that the viewers will get to see parts of the scene quickly, and will get a vision of the totality of the scene long before everything is downloaded. The composer needs to make judgments about how much to put in any given file and about what order to download the files. Your judgment should be based on the effects you are trying to achieve: test it out by going to a 486 with 8MB of RAM, and downloading the scene while taking notes on what you like and do not like about the download process.

The Builder's Eye Putting Real Things into Unreal Spaces

As we noted earlier, VRML composers can go crazy trying to make things too real. By the same token, ignoring scaling issues can make a potentially powerful scene or a beautiful object merely comical. One "trick of the trade" I use to check scaling and translation issues is to insert a standard-sized primitive object into the scene, usually at the origin. I needed a booth to use as a newsstand in a scene I was building, so I got a modeling genius I know to build me one. I brought it up in my VRML browser, looked at it from all angles, and then looked at the VRML code in a text editor to see how things were grouped. The newsstand was beautiful: relatively small file size and vertex count, nice coloring and lighting (even without textures, it still looked like a wooden booth), and very intelligent groupings of the parts of the booth (so different anchors, materials, and textures could be applied to different components of the booth). Then I checked on the scale and placement of the booth by adding a sphere (see Fig. 20.4) at the beginning of the file, something like this:

#VRML V1.0 ascii

Separator {

Sphere { } # accept the default of a 1-meter radius and location _at the origin (0/0/0)

DEF aNewsStand Separator {

... code removed for brevity

}

}

Fig. 20.4

Using a 1-meter sphere to check an object's scale and placement.

Notice how it looks like a teeny-weenie booth, perched on a giant golf ball: the effect is unintentionally comic. Remember that scenes are composed of multiple objects, and that the relative size and placement of each object within the scene as a whole is an important issue.

Sometimes it is appropriate to scale objects up or down in size for a given scene; sometimes the surreal is appropriate (remember, this is virtual reality not real reality). In the vrmLab scene, I wanted to use books as a metaphor for and a gateway to reference services on the World Wide Web. I had three search services I wanted anchors to in this scene. I put three books together on the reference desk, just like "real" books would look on a "real" reference desk: normal size books, one right up against the next. But it didn't seem to work in the way I wanted.

There were three problems with this arrangement: it was too hard to tell from the opening shot what they were (scaling), it was too hard to tell from closer up that they were three objects rather than one (spacing), and it was too hard to pick one particular book to click on (navigation). Again, the message is that a composer must not just make things but must also look at the scene and try to use the scene like "Jane Everybody." I solved these presentation problems by scaling the books up in size (they are way bigger than "real life" books, but you can discern what they are from the opening shot), and by moving them apart from each other on the desk (they would fall over if they were "real life" books).

These problems become even more critical to watch for when you are "importing" objects created in a modeler and converted to VRML. Remember the cityscape we talked about before? Each of the buildings and trees and props were originally modeled in a CAD package, and the resulting files were converted from the CAD package's format into VRML. No attention was paid to scaling issues. The individual objects were of vastly differing scales, and none of them were in-scale to "real life." As a result, when I first assembled the objects into a scene, we had some 300-meter tall one-story buildings and some 10-meter tall five-story buildings. It looked very weird. When I tried bringing a preexisting model of another structure (a model done to proper scale) into the scene, it looked even weirder. I ended up spending a lot of time just getting things into a realistic scale (so prebuilt to-scale objects could be added from an object library) and into a common scale (so things would "fit" together in the scene properly). Composers should note that converters often have sufficient controls to allow one to choose the final size of an object after conversion. Try to make the resulting files "sensible" in their sizes, and try to make them adapt a common scale.

Just as scaling can be a problem when assembling objects from modelers and from your object library, relative location can also be a problem. One thing some otherwise excellent conversion tools do is always make the middle of an object or cluster of objects be at the origin (0, 0, 0, on the X, Y, Z). As you may imagine, a newsstand which is half-way buried in the floor looks pretty funny. Another "trick of the trade" to tell where an object will end up (relative to the rest of a scene) is to place a "floor" underneath by putting a wide, deep, and shallow cube into the scene with its top edge on the 0/0 X/Z plane. The VRML code might look like this:

#VRML V1.0 ascii

Separator {

DEF theFloor Separator {

Transform { translation 0 0 -0.005 }

Cube { width 50 depth 50 height 0.01 }

}

DEF aNewsStand Separator {

... code removed for brevity

}

}

Like scaling and relative location, grouping issues are important. Sometimes it makes sense to put like objects or related things together in one group, and sometimes it doesn't. I wanted the steps beside the Temple building to load up as one "thing," but I did not want them to load at the same moment as the building. I wanted the eye to be drawn to the steps by the load order so I could take people from left to right as the loading progressed, so the steps come after the Temple building rather than being in the same file. I wanted to avoid the spiders all loading up at the same time (although some browsers defeat this arrangement), so I made each of them a separate object, loading one at a time for the dramatic effect of moving the eye gradually from the foreground back to the Temple.

It can be hard for a composer to tell just where something is actually going to end up in a scene. Where will this new object go? It is often frustrating to wait and wait repeatedly while a new object with a medium to large file size is loaded and moved and reloaded and moved and so on. Another "trick of the trade" is to use a simple object (like a sphere) in place of the object you want to add to the scene, and gradually float the simple object into place in the scene. To load, move, reload, and reposition a simple object over and over can be quicker and less frustrating: I call this "stupid floating-sphere tricks."

Note that one of the drawers of the file cabinet was left open. This helps the "mind's eye" to see it immediately as a file cabinet, and not just a cube. Note that the monitor on the reference desk has been swiveled, and is no longer in a straight line with anything else. Again, this minor rotation helps establish what the object really is to the viewer. Remember how we talked about using a metric tape measure, a sketchpad, and a simple ASCII text editor to design things? There are no better tools for planning than tape measure, sketchpad, pencil, and imagination. Make lots of messy drawings first, and then worry about making the objects.

This is just what I did with both the file cabinet and the monitor: I measured and sketched out what a real file cabinet is like, and I did the same with a real television. I then used those sketches to build the objects. Each of the drawers is a separate grouping within the file cabinet object-file, and so each can be manipulated without affecting the other or the overall object. Each major part of the television is a separate grouping within the object-file; the knobs and the screen and the body can each have meaning attached or be otherwise manipulated without affecting anything else. In this way, I can place a picture

on the screen of the television in one context and make a knob on the front do something when clicked on in another context.

Other examples from the vrmLab object warehouse include constructing a straight chair and a simple bed (see Fig. 20.5). To make these, I started out by looking at real objects and sketching what they looked like and classifying their different parts. Then, I sat on each of them to see how high the frames should be and how high the mattress or cushion should project above the frame, and so on.

[Fig. 20.5](#)

Close-up of the straight chair.

The Ad Exec's Eye Grabbing and Holding Someone's Attention

The last kind of "eye" we will talk about is that of the "Ad Exec." We've examined some of these concepts earlier on in other contexts. You need to pay attention to load order: which objects show up on the viewer's screen in what order? You need to find the balance of making it "real enough" without making it so "real" it takes forever to download and come into view: how "real" is "real enough" for this context? You need to figure out what you want them to pay attention to, and when: how much should the scene be broken up into multiple object-files? In what order should things load? Where do I want their attention to go at each step of the loading process?

Last, but not least, you should remember to plan for things not working right for all people in all browsers. A good example of this is textures. Sometimes composers of VRML will assume that everyone can "see" textures in their browser, and so they will create VRML with no colors on anything and everything dependent upon textures displaying in order for the objects to "look right." Quite a large number of people have browsers which do not do textures correctly, or at all. Some of the browsers which do not do textures will not crash when loading a scene with texturesóthey will just ignore the textures. When this happens, the viewer gets a scene with nice geometry which looks very weird because everything is that strange default "VR-white" color. It takes very little effort or bandwidth to put coloring onto objects as well as textures, and may make things much more viewable.

Summary

We've talked about a lot of things here, so let's summarize.

1. VRML is not about hardware or software; VRML is about imagination.
2. What helps make a scene "work right" for the viewer is a unifying theme and thoughtful use of metaphor.
3. The composer should focus first on what is important in the context of this particular scene (the forest), and save the details for later (the pine needles).
4. Camera, lighting, and scene composition ideas from video and still photography can be used to

powerful effect in VRML scenes.

5. Load order can be used to reinforce groupings and metaphors, to move the viewer's attention where the composer wishes it to be focused, and to keep people interested in the scene. Load balancing can be used to make servers work less. The two demands need to be balanced with each other as well, so that the efficient use of a server doesn't destroy the visual effect of your scene.

6. Scale things "right" for the context they appear within, don't just copy "real reality."

7. Make the scene look lived-in; people aren't neat and tidy all the time, and some disarray lends a subtle note of realism to a scene.

8. Get and retain their attention where you want it to focus, and make it visually and metaphorically fun.

Above all else, make it fun and use your imagination.

You can visit the vrmLab site at <http://www.newcollege.edu/vrmLab/home.wrl>.

From Here...

Once you have found the browser that best suits your tastes and computer platform, you are ready to explore and create your own VRML environments and objects. To learn about how to start building your own VRML environments, see the following:

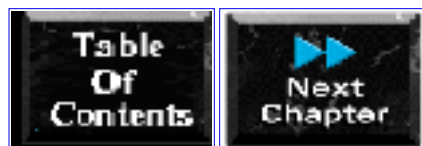
ï If you are ready to start programming in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) These chapters provide a step- by-step description of how to create VRML environments as well as the standards and practices for VRML programming.

ï If you want to look at some VRML environments and models, see [Chapter 23, "A Guide to Other Sample Environments on the Web."](#) It lists some of the VRML sites that are currently available.

ï If you want to see what programs are available to help you create VRML environments and objects, see Part VI, "Making It Easier: 3D Modelers, VRML Worldbuilders, and Utilities." These chapters provide an overview of the software that can help make it easier to program a VRML world.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



21-Using Perl and the LOD Node in VRML Environments

by Daniel F. Ancona, IV (dfa4y@jefferson.village.virginia.edu)

The following accounts look at two interesting VRML applications. Daniel F. Ancona's article describes how he combined Perl and VRML to create the Rossetti Room. Bill Rouady's article looks at the problems encountered and solutions found while creating a texture-mapped globe and how he used the LOD node to allow users to "zoom in" for closer views.

The Attainment of the Wombat Interactive Modeling of Dante Gabriel Rossetti's Studio with VRML, HTML Forms, and Perl

Project Background

One morning in May of 1995, I sat in front of the terminal, feeling equal amounts of excitement and dread. I'd been given some free reign in deciding the path of the project I'd been assigned to adding a three-dimensional element to the Institute for Advanced Technology in the Humanities' D.G. Rossetti archive and I had a neat idea, but one that was going to require a fair amount of heavy wizardry. I was starting to learn Perl, and had picked up a fair amount of UNIX, and more recently, HTML through my various jobs, but I wasn't sure I could pull this off. Jerome McGann, the professor of English who created the project, was there, and when I told him my idea he got really excited, and started pulling books out and flipping pages. He eventually came to the image in Figure 21.1.

He also showed me a plan view of the house Rossetti lived in (see Fig. 21.2). So I knew what sort of content I would try to create (that room), and I had an idea for an interface to it.

The idea I had came from what I knew about Radiance, a public-domain rendering package I used while working at the National Renewable Energy Lab in Golden, Colorado. I was working on energy-efficient

lighting design for a site in Washington, D.C. Once I had finished the official report, I had the opportunity to try to come up with some "before and after" renderings to show what the rooms would look like before and after our recommended changes were implemented. The program was hard to use (controlled almost entirely from the UNIX command line), but it was able to do something that very few programs can: it completely changed the way I thought about the problem of 3-D graphics. Since it was the first time I had really used some of the more advanced features of UNIX, it changed the way I thought about computing in general. Coincidentally, while I was waiting for the renderings, I used the web for the first time. Many times I found myself stumbling out of work and seeing the last purple traces of the sunset vanishing over the front range of the Rockies, as I had been web surfing and lost track of the time.

Fig. 21.1

Image of Rosetti's Studio.

Radiance is similar to VRML in that it uses a plain old text-based scene-description language to describe the objects and materials that one wants to render. However, rather than generating a world that one can move through interactively, Radiance creates high quality ray tracings, and this is what I initially wanted to do with the Rossetti project. The idea that was germinating in my head was to create a web interface to a Radiance model of the room, and allow a user anywhere on the web to punch in a view of the room they'd like and have it generated on the fly for them. I pictured an imagemap plan view of the room—the user could click once for where they wanted to "stand," and click again for the direction they wanted to aim their virtual camera. Radiance allows zoom and wide-angle renderings, so these could have been specified also. Then the user would hit a button on the form and a few minutes later would be able to see their picture.

Fig. 21.2

Plan view of 16 Cheyne Walk.

There were several problems with this. The first one was the problem of server load: making one of our servers create all those renderings would have slowed down the machine a lot when the load was light, and if the load ever got heavy, it probably would've crashed the machine. Or, it simply would've taken hours for anyone to get their rendering back. Another problem was the interactivity part of it—a 2-D interface to a 3-D space seemed like a bit of a kludge. A clever kludge, perhaps, but still a kludge, and I was looking for more.

At this point, I had an experience similar to the VRML equinox that Mark Pesce describes in VRML: Browsing and Building Cyberspace. A bunch of problems and solutions all came together at once. With some help from the local UVA SGI gurus, the WebSpace beta was installed on the Indy workstation that I'd been using for another project for the Institute. I had just learned the rudiments of a very powerful modeling program for the Macintosh platform called Form-Z (another program that changed the way I think about 3-D), and when I tried converting a DXF file from Form-Z with the converter that came with WebSpace, it looked terrific. I knew I could use Form-Z to make the room (there would be difficulties with this later, but they were relatively minor). The real nexus came when, after I'd flown around in a VRML model of Stonehenge in WebSpace for a little bit, I chose "view source" off the WebSpace menu and could marginally read it. VRML would be a great medium for the Rossetti room. And I was inspired by the hidden and beautiful 3-D world that Tom Meyer and David Blair had created for WaxWeb, which

I'd seen in 2-D earlier that year.

The creation of this model was a part-time effort over most of the summer of 1995. Given what I know now, I could re-create it a lot faster than that. That time includes starting from scratch on the modeling program, the VRML spec, and Perl. The simple Perl script at the end of this chapter was the first Perl script I wrote.

Creation of the Room

The first step was creating the 3-D model from the information I had, mostly Figures 21.1 and 21.2. I'm going to describe how I created it in Form-Z, but most of these techniques are available in any CAD package in one form or another. An experienced user of MicroStation or 3D Studio should be able to adapt these techniques easily enough. Also, I'm going to go through the process more or less the way it actually happened, including the mistakes. In working on a few projects that push the technological envelope, I've noticed that they often follow an iterative path to development—you try one direction, work on it for a while, realize you screwed up near the beginning by doing something that you didn't know would be a problem at the time, so you go back and start over. It's a little different model for learning that we get used to in school, and it can be frustrating until you see that it's simply the way these things work. Then it can be a lot of fun.

I started by laying down the floorplan. Although I had the plan of Rossetti's whole house, I wasn't sure how big a model I wanted to make, so I started with one room. I figured out the right dimensions and made a 2-D plane that size. Next were the rough walls, which I extruded up from the widths I could see on the plan view.

The plan view showed some windows, even though the pencil drawing didn't, so I took a guess at their sizes and what they might look like and used a boolean subtract to remove the windows from the wall. I did the same thing for the fireplace, and made a few rectangular polygons for the mantle and base. I made rather an ornate mantle for above the fireplace, with beveled edges and extra detail.

At this point, the room was more or less recognizable. When I moved to the position I thought the drawing was made from, the proportions looked more or less correct, so I started on the arches visible on the right side of the drawing after making the two pillars by the door. I made this shape by drawing a few circles along the bottom edge of a rectangle, playing around with them until they more or less fit the dimensions of the floor plan and the pillars and looked like the drawing, and then extruded this shape to the right size. I then put another polygon behind the arches to fill in the area above the door.

The next thing to make was the paintings. Since we had the real-life sizes of these paintings, I simply created rectangular, two-dimensional polygons the size of each and moved them to various locations around the room. (For more about creating polygons, see Chapter 12.) The picture showed them scattered more or less willy-nilly around the room, so I didn't follow any particular pattern in placing them. The room still looked a little thin, but I was reluctant to get into modeling furniture since I didn't know how hard or easy it would be. Instead, I tried adding molding. I started by extruding the cross section shown on the left in Figure 21.3. This looked good when I rendered it in Form-Z, so I left it.

[Fig. 21.3](#)

Molding cross sections (big, small, and small corrected for VRML).

The reasons for the other two cross sections that I eventually used will be explained in the next section.

As my skill with Form-Z increased, I made a very nice couch that looked like the one in Figure 21.1, but I despaired of being able to include because of its high polygon count. However, this problem would be solved by a VRML and script trick.

Translation to VRML

The model looked pretty good when I rendered stills of it in Form-Z, so I started converting it. Each object was on its own layer, and Form-Z allowed DXF out by layer, so that's what I did. After bringing all the DXF files over to the Indy, I converted the files to Inventor format with `dxftoiv`, and then converted the Inventor files to VRML with `ivtovrml`, the converter that is included with the SGI distribution of WebSpace. ([For more about VRML utilities, see Chapter 26.](#)) This process worked perfectly except for one wall—the wall with all the windows in it. It converted, but the program complained about an error that I didn't quite understand (I later posted to the `www-vrml` mailing list and found that it had something to do with the way Form-Z makes its DXF files.) I tried several different methods of making that wall, and it continued to complain until I made it as a series of individual rectangular polygons. I noticed that these render a little funny in WebSpace—the lighting from the point light that I later added in the corner seems to be applied to each polygon individually, so the boundaries don't exactly match. But it's close, and I have a feeling this is the sort of error that users don't even see until it is pointed out to them.

The big problem was that it ran very slowly. I estimated the frame rate at around 2 frames per second (FPS)—little better than a 3-D slide show. I brought this up on `www-vrml`, and at one point Bernie Roehl posted a list of ways to improve the frame rate. "Reduce the number of polygons" was listed about four times, so that's what I started doing. The general heuristic I used from then on was this: evaluate every object in the scene in terms of the information it adds, and remove any objects that add small amounts of information. I started with the outside walls, and then simplified the windows into single polygons. I simplified the mantle for the fireplace, and removed some of the outside of the walls around the fireplace itself. I simplified the profile of the molding to the one in the middle in Figure 21.3, but when I put that into the model I noticed some strange triangular patterns that appeared across it's length when I converted it into VRML. I'm not sure why this happens, but it does whenever two planar polygons are placed too close to each other (the same thing happened with the paintings, which I simply moved farther away from their adjacent walls). This led to the shape on the right in Figure 21.3, which uses a minimum number of polygons but still looks good in the model.

Troubleshooting

Troubleshooting

I later discovered that these triangular patterns happen because of finite depth-resolution in the Z-buffer. However, my approach to correcting the problem was the correct one: move the objects further apart.

Now it was time to add the textures: those included the paintings as well as textures for the rest of the

room. I concatenated all the files together and brought them up in my favorite UNIX text editor, Lucid emacs. Most versions of emacs include balanced parentheses checking, and this was invaluable in debugging the VRML files after I'd edited them.

To insert the WWWAnchors and texture nodes, I fired up emacs and began editing the VRML manually. (For more about using WWWAnchor node, see Chapter 15.) An example of the result is shown here:

Listing 21.1

#VRML V1.0 ascii

WWWAnchor {

name "http://jefferson.village.virginia.edu/rossetti/pictures/_S244Gdoc.html"

description "S244G"

Separator {

Info {

string "Dame_Sketch"

}

Texture2Transform {

scaleFactor 1 2.35

}

Texture2 {

filename "http://urban.arch.virginia.edu/~dfa4y/dgr2/S244G.gif"

}

Material {

ambientColor 0.125 0.125 0.125

diffuseColor 0.5 0.5 0.5

}

Separator {

[...etc...]

}

}

}

The painting textures were from the archive, of course. In xv, the X-windows graphics program, I made all the paintings much smaller than the ones available in the archive so as to minimize download time. The marble texture for the fireplace came from one of the big texture warehouses on the web, TextureLand. I had a hard time finding a good floor texture, so I fired up the small video conferencing camera on the Indy and took a snapshot of the carpet in the room where that machine is and then mucked around with it in xv.

The WWWAnchor above is a Separator around the whole object, and the Texture and Texture2Transform add the painting. The necessity of the Texture2Transform node was confusing at first since I thought that the texture should fit exactly in the size of the object created. I'm still not exactly sure why the transformation was needed, but it was, so I manually adjusted it until each painting looked just right in the frame. (I couldn't figure out an easy way to do this interactively, so I would tweak it in the file, save the file, then reload the .wrl.)

I added these nodes to each painting, so that each one would have the appropriate texture and be a link to its respective page in the archive. I also made the roof a link back to the VRML creation page, and the door on the left when you enter a link to the top of the archive page. The process I used to find each of these nodes in the .wrl file was a little laborious (I went through each file and stuck a Material node with a bright emissiveColor underneath each separator, although I could've just gone back into Form-Z and saved each object I wanted to link as a separate layer), but it allowed me to see how the DXF converter made the files. Also at this point, I removed a matrix transformation that the conversion process had added to each file. The matrix transformation was performed once, in the opening file with all the room information in it. I tried removing the matrix transform node altogether, but it caused moving forward in "walk" mode on the browsers I tested it on (WebSpace SGI, mostly, but VR Scout for Windows 3.1 more recently) to work as expected and thus had to be left in.

Troubleshooting

Troubleshooting

Note that because the matrix is scaling the world, the speed at which you move will be different.

I also used a test node to set up the lights. Form-Z saved the model in inches, which I found very difficult to think in when moving around. To keep my bearings, I made a small test Cube{} node, underneath a Separator with a bright emissive Material node and a Transform node, like this:

Separator { #This block makes a little, bright red cube

Material{

emissiveColor 1 0 0

}

Transform {

translation 220 140 80

}

```
Cube{}  
}
```

Using the red cube to guide my placement, I laid down one `PointLight` source near the middle of the room, and a `DirectionalLight` facing the wall with all the paintings on it.

I used Level Of Detail only once on the very detailed couch. Form-Z has an excellent rounding and filleting feature (far better and more predictable than any other CAD system I've used), and I went a little crazy with it. When I tried to add it to the .wrl, it slowed the whole thing down. To try to combat this, I manually added the LOD node to replace the couch with a regular rectangular prism. It works, but even with the LOD in place, including the couch still slowed the browser down significantly.

At this point, I had a bunch of VRML files that all looked good, and just needed to be put together. It was at this point that I thought of using a script to allow the user to add whatever paintings (and later, objects) he or she desired. This had unexpected benefits that I didn't even realize until after I'd started working on it.

The Form and the Perl Script

Since it's hard to consider a script without thinking about the HTML form that calls it, here is an example of two of the relevant parts of the form. The first is the opening of the form, where it specifies the script to be called. Also shown is the opening line of the table to be created:

```
<form method="GET"  
action="http://urban.arch.virginia.edu/~dfa4y/dgr/dgrvrm1.cgi">  
<table border=1 cellpadding=2>
```

I decided to go with a table even though I'm not sure of the status of tables in the HTML standard (I know my friend's AOL browser doesn't like them at all). This project is generally experimental, so using experimental HTML tags didn't bother me too much.

The second part of the form is an example of one row of the table. The most important part (as far as the script is concerned, anyway!) is the checkbox form tag in the first `<td>` cell of the table:

```
<tr>  
<td align=center><input type="checkbox" name="S244G">  
<td><a href="S244G.jpg"></a>  
<td align=center><a href="/rossetti/pictures/S244gdoc.html">S244G</a><td>The Blessed  
Damoze1,sketch for background</a>  
<td>15.5x36.5in
```

That table row was repeated more or less for all the rows of the table. The strange number (S244G in this case) refers to the Surtees number. Surtees is one of the major scholarly reference works for Rossetti and

is a convenient way of keeping track of Rossetti's works.

Now that the form and VRML had been created, all that was left was the script. It's very simple Perl, and I've added lots of comments to show what the script is doing at each point. Most of the idioms are either from "Programming Perl" (the Holy Writ of Perl programming) or were passed on to me by Thorny Staples, a project manager and programmer at IATH.

Listing 21.2

```
#!/usr/local/bin/Perl
```

```
# dgrvrml.cgi
```

```
# A Perl script to create a page of HTML to access several VRML files, one of which is created on the  
fly by concatenation of several other VRML files. Created by Dan Ancona (dfa4y@virginia.edu) for the  
Institute for Advanced Technology in the Humanities, University of Virginia, Charlottesville, Virginia.  
Last modified on 6 November 1995.
```

```
#Open STDOUT to a label called screen
```

```
open (screen, ">&STDOUT");
```

```
# Deliver the correct MIME type so the browser knows what's coming. This could be changed to send a  
stream of VRML directly, but I prefer to have a page that lists the objects that were included, and so that  
the user knows exactly when the VRML is coming (i.e., when they hit the link that says "view the whole  
room").
```

```
print (screen "Content-type: text/html\n\n");
```

```
sub ReadQuery {
```

```
## Reads the query string from the HTML form, and puts all of the objects to be added to the room in an  
array.
```

```
#This statement gets the query string from the environment variable that the web browser sets when the  
"Make VRML" button is pushed. This string can be seen in the Location window of the screen that  
comes up after the button is pushed.
```

```
$querystring=$ENV{"QUERY_STRING"};
```

```
# Now split apart the variables and put them in an associative array.
```

```
%queryarray=split(/[=&]/,$querystring);
```

```
# Go through each item in %queryarray (remember that the pairs of elements in an associative array are  
referred to as keys and values. Searching through an array like this is idiomatic and elegant). I'm not  
exactly sure how the body of the loop worksóI copied it :). All I know is that it translates the way that  
variables are stored in the query string back into normal text. Note that the values of the array are all  
"on," and are discarded. The way the form is set up, any variable that appears in the query string is  
automatically going into the .wrl.
```

```
foreach (keys %queryarray) {
```

```

$P{$_} =~ tr/+//;

$P{$_} =~ s/%([a-zA-F0-9][a-zA-F0-9])/pack("C", hex($1))/eg;

}

# Now that the keys have been translated, put them in a simple array.

@objects=keys %queryarray;

# Now join all of the objects in the array into a string that looks like "file1.wrl file2", to be used in the
concatenation.

$addstring=join ('.wrl ', @objects);

}

sub WriteVRML {

## This one line procedure is where the actual work of the program takes place: Call the unix "cat"
program to concatenate all the .wrl files together. These files are somewhat misnamedóneither room.wrl
nor end.wrl are truly VRML. room.wrl is missing a closing paranthesis, and end.wrl consists solely of
that missing closing parenthesis. <shrug> Just following one of the Perl maxims: do the first thing you
think of. The backticks tell Perl to perform what they enclose in the OS.

'cat room.wrl $addstring.wrl end.wrl > dgr.wrl';

}

sub WritePage {

## Now that the VRML file has been created, write out a page announcing what files have been included
in the whole room .wrl, and gives the user a chance to link to any of the files at will.

print (screen "<html><head><title>DGR VRML Created</title></ _head>");

print (screen "<body>");

# Uncomment the line below to check to make sure the right items are being passed into your array.

# print (screen "@objects<p>");

print (screen "<font size=+1>VRML Created!</font><p>");

print (screen "<a href=\"dgr.wrl\">View the whole room (dgr.wrl)</_a><p>");

print (screen "Or, view each picture or object by itself in _3D:<p>");

print (screen "");

# The same idiomatic structure used before. $_ refers to each element of @objects as it is looped
foreach (@objects) {

```

```

print (screen "<li><a href=\"$.wrl\">$.wrl</a>");
}

print (screen "</ul>");
}

###MAIN

## Nothing left but to call the procedures in the order they've been listed.

&ReadQuery;

&WriteVRML;

&WritePage;

```

The real work of the script then is really the backticked system call to "cat"óthe UNIX command to concatenate files. The rest is just a wrapper that gets the input from the form, and makes output to let the user get at the VRML that it puts together.

The end result of the whole process is shown in Figure 21.4. This shot was taken with all of the objects on the form turned on.

I started the script as a simple example of integrating Perl and VRML and as a pointer to what might be possible, but it ended up being far more practical and useful than I initially had thought. The script now acts as a sort of extended LOD feature. If a user wants lots of detail (like when I took the screen shot shown in Figure 21.4), he or she can add the extrasóthe couch, the molding, and the shelf. If speed is needed, for example on a portable PC for a demo, the extra details can be removed altogether. This probably won't be necessary for very long as browsers get more efficient and more powerful hardware-based polygon-pushers become available, but it is a practical use for existing technology.

[Fig. 21.4](#)

Screenshot of the finished model.

Conclusions and Possibilities

In conclusion, mark the page with Figure 21.4 on it with your finger, and flip back a few to Figure 21.1. Look at Figure 21.1 for a while, then flip back to Figure 21.4. Repeat this a couple of times and see what you think. Better yet, fire up a web browser yourself, choose your combination of speed and detail, and walk around for a while (the URL for the Rossetti Room is <http://jefferson.village.virginia.edu/-dfa4y/dgr/>). No, it's not immersive, but with sufficient suspension of disbelief, it almost feels like you really are driving a golf cart around Rossetti's studio.

The goals for the script evolved as it was created, but it does accomplish a few things. Primarily, it demonstrates the possibility for linking VRML with Perl. The next step in this direction is to automatically create the polygons for the paintings directly from data already in the archive. The archive is all written in SGML (Standardized General Markup Language, of which HTML is a subset), and

contains all the data used to create the paintings, although they would have to be placed either randomly about the room, or interactively, after they'd been created. Also, the room is an example of one type of spatial metaphor that can be given to the data of the archive. The archive is big, but the rest of the house could easily be digitized (from Figure 21.2 and other available pictures, for example) as a method of organizing the data and giving users some kind of positional feedback while they are perusing the works of the archive. The dual browser format commonly used now, where the VRML and the HTML browsers appear in separate windows, is ideal for this, since users can view both the content of the archive and their position in 2-D and 3-D simultaneously.

All this is possible with the technology available today. As the specification progresses, some of the more complicated aspects of the model will be possible. Beyond more detailed lighting models, collision detection, and higher frame rates, far in the future lies the "omega point" of the development of this model: the realistic portrayal and behavior of the wombat that Rossetti kept as a pet. Only at that point, the Attainment of the Wombat, will the model truly be complete.

Ever Changing Details Paper Software's LOD Earth

by Bill Rouady

One cannot deny that VRML offers a large array of dramatic opportunities for the 3-D modeler. The ability to offer the user a real-time 3-D experience—as opposed to canned, pre-rendered animations—improves your chances of creating an eye-opening experience. The interactive nature of VRML provides you with the opportunity to give users a unique experience based upon these "travelers" own interests and curiosity.

One of the most interesting, and easily exploited, aspects of the VRML specification is the Level Of Detail (LOD) node. LOD allows a VRML world to change with changes in the user's view. With the camera representing the user's eyes in a VRML world, its location can trigger active events, new sights, or change the entire scene in which the user travels. This experience adds a new level (no pun intended) to the VRML world. And we will see more and more VRML environments growing beyond being static models of some virtual objects in space, to becoming active worlds that change before your very eyes.

This article looks at the development process that went into the creations of Paper Software, Inc.'s LOD Earth VRML model (see Fig. 21.5). This world allows you to move from an orbital view of the Earth to an aerial view of a city block in San Francisco. It makes extensive use of LOD and texture mapping. And the experiences that we went through should help you understand how these nodes can be used to help your VRML environments become more interesting.

NOTE

For more about using the LOD node, see "Improving the Performance of Your VRML Environment," (Ch. 17). To learn more about using texture maps, see "Adding Realism to Your VRML Environment," (Ch. 14).

The LOD Node

Using LODs can be both simple and confusing. The very nature of the LOD node helps to explain this apparent paradox. A basic LOD node is coded as follows:

```
LOD {  
range [r]  
center x y z  
Separator #1 {...}  
Separator #2 {...}  
}
```

The parts of this little puzzle are easy to distinguish. When the camera is at a position greater than r units from the point in the 3-D space defined by $x\ y\ z$, then whatever code that exists in Separator #2 is active (that is, it becomes visible), and the code in Separator #1 becomes inactive (it is invisible). When the camera passes through the imaginary barrier that exists r units away from $x\ y\ z$ and is at a position less than r units from the defined point, then Separator #2 becomes inactive and Separator #1 becomes active.

You can imagine that a LOD is an invisible sphere with a radius of r and a center point at $x\ y\ z$ in local coordinates. It is important to recognize that the trigger barrier is in the shape of a sphere. You must always take into account that you are working in 3-D space and have little, if any, control over the path that the user will take. Therefore, you have to respect the possibility that the user may enter a LOD from an angle that may be different than the approach that you had intended. If a LOD becomes confusing when entered from the "wrong" direction, then it is not working as effectively as it should.

How the point exactly r units from the center point is handled by various browsers differs. Some browsers make a basic cut from the active to the inactive Separators while others will do some type of transition (e.g., a dissolve) between the two Separators. Either way, the movement of the camera position changes the VRML world in which the user is traveling.

[Fig. 21.5](#)

Screenshot of the opening scene of Paper Software's LOD Earth.

One area where the LOD node may be confusing is in its syntax. The basic layout of a LOD is that the first Separator represents the closest range, the next Separator is for the next range, et cetera. This makes basic sense in organizing the LOD structure. However in the vast majority of its application, a LOD is designed to change as the user approaches the imaginary center point. Most often it is the furthest LOD that the user will actually experience first, with the nearer LODs only appearing if the user continues to approach the center point. Thus, in most cases, the LOD Separators are organized with the first active Separator being the last in the LOD list and the last active Separator being the first in the LOD list: that is, in reverse order of the actual user's experience! While this factor doesn't make the use of LODs difficult, it does require that special attention be paid to how it is coded and for the programmer to be especially careful in choosing the order of the Separators.

Moreover, a LOD is not limited to two Separators and one imaginary boundary. In fact, there is no defined limit to the number of ranges that can appear in a single LOD. A LOD could be constructed with any number of comma delineated range triggers [r1, r2, ...rn] and a corresponding number (n + 1) of Separators. In this way, a series of triggers can alter the VRML world as the user gets closer to the center point of the LOD. These triggers can be seen as a series of concentric spheres emanating from a shared center point. As the camera moves through each sphere's trigger barrier, the next LOD Separator is activated and the previous one becomes hidden. It is possible to do a primitive form of model animation. Using this technique, we can set a series of triggers close together with different transition states of the model exposed by each successive trigger. For example, to make a door swing open, a series of four LODs could be used. The first would be the door fully closed and the last being the door fully open. As the user approaches, the LOD triggers two intermediate stages where the door is partially open. As a result, when the user approaches the door (where the center point is), the door appears to swing open!

TIP

The use of LOD to simulate behavior is very browser dependent, and therefore should generally be avoided.

One other important aspect about the use of LODs is their ability to be nested. Once again, there is no defined limit to the number of LODs that can be nested. In fact, I have successfully navigated a VRML world that consisted of more than eighteen nested LODs! While performance may take a bit of a hit in such a situation, this feature is the foundation of Paper Software's LOD Earth VRML Model.

Planning the LOD Earth

Now that the basics of the LOD have been reviewed, we will review the creation of a project that makes extensive use of the LOD and its capabilities and outlines many of the pitfalls and opportunities that arise from a deeply LODed VRML environment. The purpose of the LOD Earth was to create a journey that allows users to begin from a point far out in space, gazing upon the Little Blue Earth as it hangs suspended in the universe, and to travel closer and closer until they are gazing upon a city block in the City of San Francisco.

The complexities of designing such a journey can be quite astonishing, with some only realized after actually beginning to piece together the materials for the project. To begin, we were faced with the idea of trying to maximize the texture quality of the Earth throughout the journey. Some browsers put restrictions on the size of mapped textures (128€128 or 256€256 pixels) which creates problems when the user zooms in on the object. For example, when using a 128€128 texture map of the earth, the entire United States will fall into an area about 7 pixels high and wideónot exactly a high resolution image.

In order to be able to use a series of smaller-resolution images instead of extremely large, single images, it was necessary to use the IndexedFaceSet node to create the model as opposed to using a sphere primitive. There are pros and cons to doing this. The primitive sphere is much more efficient an object. With a simple declaration of its radius and position, the browser creates the sphere internally. Therefore, the browser can also render this sphere more efficiently than non-primitive objects.

In contrast, the texture coordinates of the primitive are hard-coded. The programmer has no control over them. Moreover, you cannot use multiple texture maps or "slice-up" the primitive sphere into parts. After all, that is the point of the primitive. Thus to accomplish our task, we had to use the IndexedFaceSet node to create our sphere. It was these indices that would be manipulated in order to subdivide our model into smaller and smaller segments so that higher resolution texture maps could be applied.

Planning and Making the IndexedFaceSet Earth

Creating a sphere using the IndexedFaceSet node can be a challenge. As a matter of fact, anybody who would be willing to calculate and build such a sphere from scratch should seriously consider some form of therapy. The most efficient way to accomplish this task is to cheat. Fortunately, the VRML programmer has a number of very good cheats available.

In our case, the original sphere was created using Autodesk's 3D Studio. Using this program, we were able to build a sphere to our exact specifications, (number of total faces, vertices, et cetera), and then export the model for conversion from its native .3ds format to the VRML .wrl format. AutoDesk provides a free plug-in VRML converter for its program (it's not perfect, but it works) as do many other 3D graphics software manufacturers. In addition, there are a number of shareware/freeware utilities that also perform the conversion process. As this field develops, we are sure to see great leaps in authoring tools and conversion programs, but for the moment, a workman-like programming effort is still necessary for some aspects of the creation.

After creating the sphere, we sliced it up into four segments while still in 3D Studio in order to take advantage of the conversion to come. Careful planning is the most effective tool at this point. We needed to segment the model so that it would not move in 3D space. Otherwise our LOD Earth would not only increase in resolution as you moved forward, but it would dance around the screen as well! By slicing the earth into four and then 16 segments, we developed the first three LODs of the process. Since each segment would be mapped with a 128€128 image, we would be able to change the effective resolution of the entire Earth surface from 128€128 to 256€256 and then 512€512 a sixteen fold increase in quality!

Now was the moment of truth. The various Earth models were exported and the .wrl files created, They were pretty ugly. Looking at the .wrl file was vaguely comparable to sorting through the calculator tape of an accountant on April 16. Each vertex was identified, then each face was defined by a combination of vertices, and finally mapping was applied to each face.

This was a moment of decision for us. There were two options that had to be considered. One was easy, and the other was not. As a VRML programmer, it is best to have as much control over the display of the .wrl file as possible. Different browsers may do things differently, and so the greater the amount of control that we could maintain, the higher was the likelihood that the model will display effectively on the wide variety of browsers. This helped resolve the dilemma of whether or not U/V coordinates need be applied to the IndexedFaceSet just created.

The U/V coordinates are a very effective tool. A face that is created from vertices with no U/V coordinates supplied is at the mercy of the browser's mapping algorithm. In many, if not most, cases, the browser will perform an effective job at mapping the faces correctly. However, we are dealing with an object where texture mapping is crucial. The maps need to lay correctly on the face or seams will be seen between the different objects and our high-resolution Earth will start to more effectively resemble a

segmented orange which happens to have skin that looks like the surface of the Earth. The problem with applying these coordinates at the time the model was being created was straightforward: there was no effective conversion tool that could perform this feat. It would have to be done by hand.

It was apparent that we would have to somehow map each coordinate in order to create a consistently viewable model. Fortunately, laziness is another mother of invention, and cheats were available to help alleviate some of the difficulties. 3D Studio had an ASCII output format that included the U/V mapping coordinates for each vertex of the model. Additionally, we had the good fortune that the order of the vertices in this ASCII file matched the order of the vertices in the VRML conversion plug-in's output. Because of this, we were able to output both files and, using a text editor, cut and paste the U/V coordinates into the VRML file so that we could guarantee our texture maps would always map the same way.

Making an IndexedFaceSet Sphere

That task accomplished, we moved to the nitty-gritty task of breaking down the faces level by level and applying increasing levels of texture to smooth our journey—as well as changing the types of texture to add further information to the effect. But before jumping headfirst into that aspect of the process, it is important to take a few moments to examine the IndexedFaceSet node and how the U/V coordinates are applied to each face. This will be very important as much of the continued calculations have to be done manually since no tools or easy methods exist for this particular task.

The IndexedFaceSet (as shown in Fig. 21.6) is defined by two three-sided polygons mirrored vertically and horizontally. The vertices are marked as 0,1,2,3, and each vertex is defined by x y z coordinates (remember, we're dealing with 3D space). The vertices are indexed based on their order in the file and not on their order in the model. The index numbers are then used to define the faces as shown in the following code sample:

```
Coordinate3 {
```

```
point [ x0 y0 z0,
```

```
x1 y1 z1,
```

```
x2 y2 z2,
```

```
x3 y3 z3 ]
```

```
}
```

```
TextureCoordinate2 {
```

```
point [ 0.0 0.0,
```

```
0.0 1.0,
```

```
1.0 0.0,
```

```
1.0 1.0, ]
```

```
IndexedFaceSet {
```

```
coordIndex [ 0, 3, 2, -1,  
0, 1, 3, -1, ]  
textureCoordIndex [ 0, 3, 2, -1,  
0, 1, 3, -1, ]  
}
```

As noted, the index numbers used in the IndexedFaceSet section are based upon the order in which the points are listed in the Coordinate3 section. The -1 is used to indicate the end of the face definition.

Applying the U/V coordinates works in a similar fashion. Each vertex has a matching U/V coordinate definition. The order and number of U/V coordinates is equivalent to the vertex coordinates. In other words, the first vertex coordinate is matched to the first U/V coordinate and so on. The U/V coordinate numbers are based upon a planar definition of the textual map with point 0,0 at the upper left of the image (see Fig. 21.7). The textual map has no third dimension so a z-type coordinate is not necessary (nor would it make much sense). After the U/V coordinates are assigned to each index, they are then assigned to the faces. This process is under the IndexedFaceSet section and completely matches the coordIndex definitions.

[Fig. 21.6](#)

Example of an IndexedFaceSet polygon.

[Fig. 21.7](#)

Applying U/V coordinates.

The importance of understanding these details becomes apparent as we begin cutting and creating new faces in order to apply more texture maps to a defined space. Since no current authoring tool could perform these feats, it was a manual process. With this groundwork laid, we began the building of the .wrl file.

Constructing the LODs

The first step in the final .wrl construction was the integration of the three Earth detail levels. This was a fairly straightforward process as each of the three models replaces the initial model in its entirety. Also, they can all share the same center point. And since the LOD, item is a sphere, the most logical center point that they share is the center of the sphere.

There is one point to which we must pay careful attention. Because we will be nesting LODs further and the replacement models will not consist of the entire Earth, the segment (in the 16-part model) should be moved to sit at the top of the 16-part Separator. The reason for this is that we will be removing just this segment in the next LOD and we do not want the rest of the Earth to disappear along with it. Therefore, we move it to the top so that we can nest all the LODs in one place and will not have to keep searching all over the .wrl file as scenes change. This improves not only readability of the .wrl file, but it also reduces confusion while writing and debugging the code.

The First LOD

The major LOD is now created as follows (the camera starts 500 units from the center point):

```
LOD {  
range [ 250, 400]  
center 0 0 0  
DEF Earth16 Separator {  
SanFran Segment {}  
Remaining Segments {}  
}  
DEF Earth4 Separator {  
}  
DEF Earth1 Separator {  
}  
} #Close LOD
```

This code sets the first LOD at 400 units from center and shifts the model from the single-object Earth with its 128€128 texture map to the 4-Part Earth with its virtual texture map of 256€256 (or four 128€128 texture maps). The second part of the LOD is activated when the camera passes within 250 units of the center and shifts the model from the 4-Part to the 16-Part Earth. It also changes the virtual size of the entire Earth's texture map from 256€256 to 512€512. These shifts might sound semi-dramatic, but in fact, they are not. It is a positive aspect of doing smaller transitions. Rather than dramatic changes in the quality of the Earth, the shift is visible but more subtle and enhances the viewers' experience of travel. After all, as you travel towards something, it becomes clearer and clearer, it does not pixelate beyond recognition and then come into focus again!

More LOD Levels

We now begin moving into the specific detail of creating LODs with higher-resolution texture maps to bring us from outer Earth down to the city streets of San Francisco. We begin by creating our first nested LOD. Nesting the LODs is necessary to avoid having two IndexedFaceSet that share the exact same 3-D space. If this happens your image may flicker, the wrong texture may show, or you will get an error message as you move through the different levels. This LOD will occur within the Earth16 Separator and functions with the San Francisco segment. We would like to leave the remaining 15 Earth segments intact. You probably recognize at this point that the remainder of the Earth will pixelate beyond recognition as we get closer to our goal. That is correct. However, the remainder of the Earth is not the focus of our journey, so it need not be an ultra-high texture, and we will still be able to give the viewer the dramatic possibility of turning the camera to see the horizon or entering our San Francisco from an angle without the rest of the earth disappearing and making San Francisco look like a city on a flat rock

drifting through space.

We set the LOD replacement to the previous segment as a four-part segment. This new segment was created by breaking down the coordinates of the replaced segment (see Fig. 21.8). In essence, the new model is four separate models of flat faces, each with its own 128x128 texture map. The new vertices are calculated by taking the median of the two known vertices to define the coordinates of the new vertex. Careful attention must be paid to the signs (positive/negative) of the vertices, especially if one is positive and the other negative. Also, all three vertices must be calculated this way: don't forget the z-axis! The central vertex of the section is calculated by using the newly calculated side vertices. This actually permits a self-check as the top/bottom and left/right combinations should produce the same result.

This might sound a bit complicated. Remember that often you would be working with vertex coordinates that resemble 54.325164, 73.654654, -12.355454, and the calculations can be quite tedious. Also remember that the high decimal accuracy is important because, as we move deeper into this maze of faces, small flaws will become more obvious and seams will start to appear, even though the seam is a result of a bad calculation two or three LODs back!

We now code the four new faces as outlined in the VRML spec and above. Each of the four faces will be a unique and separate model. And yet the combination of all four will fully replace the segment that is being removed. The U/V coordinates simplify at this point since we are only dealing with four vertices on each model. Basically the example of the U/V Coordinates outlined previously in Figure 21.7 will be the same for each of the faces as we move along.

[Fig. 21.8](#)

Breaking down the coordinates.

We are now prepared to LOD these new models into the scene. Again, one of these new models will be the piece that is replaced in the next iteration of the process, and that piece should be at the top of the LOD so that it can be nested again (and again, and again). We also want to change the center point from which the LOD will mark its range. Why? If we use the center of the Earth and users come from the opposite side of the planet, the LODs will be loading in despite the fact that they are completely invisible. Making the LODs appear only when visible makes the code more efficient. Moreover, we might want to have more hot spots on the Earth to LOD into than just the Bay area, so we want the LODs that come in to be applicable to the journey that the user is making.

For purposes of our example, we will call this new center point xSF, ySF, zSF. The relevant portion of the code would be as follows:

```
DEF Earth16 Separator {
```

```
LOD {
```

```
range [150]
```

```
center xSF ySF zSF
```

```
DEF NewPieces Separator {
```

```
NextSanFranPiece {}
```

Other Pieces {}

}

Separator {

SanFran Segment{}

}

} #LOD Close

}

The continuation of the process is fairly predictable. Each successive LOD takes one of the current portions, and replaces it with four new faces, each individually mapped with a new 128€128 texture which, in turn, has one of its faces defined by a LOD node, and so on. In this way, you have an exponential shrinking of the scene down to a very small level of detail in comparison to the original starting point out in space. In the original San Francisco Earth model, the detail traveled down through 15 levels of LODs. In other words, by the time the user reaches the final LOD texture, they are looking on a texture map that, if mapped along full surface of the Earth, would be more than 4.1 million pixels high and wide. That would be one amazingly horrific download.

Texture Maps

There are a variety of issues and questions that arise during the process of creating a LOD model such as the one described in this chapter. The actual creation of the .wrl file involves many decisions in which there is no real right or wrong answer, yet these decisions can heavily affect the process and ease of the creation.

One problem that became apparent very early in the process was the issue of texture maps. The question was how big to make them. As was mentioned earlier, some browsers put restrictions on the size of texture maps while others do not. In fact, some don't show textures at all. In this case, the best move seemed to be a lowest-common-denominator approach; that is, provide textures that any texture-mapping browser would be able to use. The smaller texture maps also mean that there would be less effort on the part of the browser, and therefore better speed in the rendering process. In this particular project, using high resolution maps didn't seem to provide enough of a benefit to offset the extra overhead that would be required because the LODing of the physical faces would still need to be carried to the same level either way. San Francisco takes up a given amount of the Earth's surface and higher-resolution mapping doesn't change that fact.

The nature of the texture maps was another issue. As the process was carried out, different types of maps were used at different levels of the model. This provided benefits and complications throughout. One of the key benefits came through the fact that the journey itself could provide information along the entire route. The original map was aimed at providing a realistic view of the Earth from space. This map was created from a composite of Earth photos from space. The choice arose as to whether the journey should maintain a realism throughout or use some other types of maps to enhance the journey. The eventual result was a type of compromise decision. While we wanted to maintain a realistic journey, we also realized that a purely real journey could be boring. The continuous refining of the image would not be a

very stimulating trip. So, the second set of textured LODs were based upon a relief map image. This provided a generally realistic shift from a flat Earth to a bumpy Earth, maintained the color integrity, and also provided an opportunity for the viewer to see the contours of the region to which they were traveling. Infra-red surface images were discussed but were not chosen because of the radical change in the color palette that would occur. However, they could be added later.

The different sets of images came from a variety of sources. Some were based on different types of satellite imagery, US Geological Service images, Shuttle images, and high altitude photographs. Each presented us with its own color scheme and perspective. Correcting the color scheme to more closely resemble its predecessor was not a very difficult chore. Any image manipulation program would do the trick, but the perspective problem was a bit more troublesome. The different sources sometimes had radically different perspectives on the scene; high altitude photos could come from almost any angle although most were very close to straight up. Satellite images usually came from the south; they also could be slightly from the east or west. Shuttle images were not only taken from different angles, but often had shifts in the camera position as well. Manipulation of these images to match was fairly time-consuming and required a more robust image manipulation program than simple color shifting. The main point to the matching was not to link the new set of textures together, but rather to provide as seamless a match as possible between the newly LODed faces and the older faces. This effort was semi-successful. Perfect matches were difficult, and some cheating was required to achieve the matches. In the end, the close-up of San Francisco was an infra-red image, and the colors made a jarring shift (see Fig. 21.9). However, it was decided that the effect was OK as it tended to announce the arrival at the end of the journey.

To Inline or Not to Inline

During creation of the .wrl file, the issue of "To Inline, or not to Inline" reared its head. (For more about using the WWWInline node, see Chapter 15.) Obviously one of the huge values of the WWWInline code is to "objectify" the .wrl and call upon these other pieces as necessary. Not only do Inlines reduce the size of the final .wrl file, they also enhance modification, making it easier to implement changes that affect only pieces of the puzzle. Our original intention was to use a great deal of Inlines in the model, but it didn't turn out that way. We kept watching the size of the main .wrl and found that the additional face creation did not add a lot of coding to the file. In fact, the final .wrl file was still under 200K and had not been GZIPped.

We had an added complication. We were not making wholesale replacements of the model but just pieces of it which made the Inlines less consistent in their use and construction. Although we did not end up using the Inline approach with this original project, the better approach would be to use them. The extra housekeeping necessary to maintain a group of files is more than offset by the advantages they offer in the object-oriented approach and, as we see ahead, in debugging problems in the code. For these purposes, a WWWInline node can be invaluable.

Fig. 21.9

Screenshot of the San Francisco close-up.

Debugging and Other Issues

When we discuss the creation of the project, a lot of details are left out. These are usually the dumb mistakes and problems that occur during the process. While dumb mistakes and typos don't make for a very interesting topic, they're a vital part of VRML codingóvital in that the existence of a typo or dumb mistake can play havoc with a file to such a point that it seems nearly impossible to track down the cause of the error.

One of the first things to do to minimize such an occurrence is to carefully choose the tools that will be used in the process. To begin, you will need a good text editor. There is WordPad for Windows 95 users. The first choice for most people using Windows 3.1 is Notepad, unless the .wrl file grows too large and Write must be used. DOS users often times will use the Edit command, and certainly, all these choices are acceptable because the only output is a simple ASCII text file. Using a full blown word processor seems like overkill, and can often bog down the system since it is nice to keep the browser running so that changes can be checked and tested conveniently. We used all of these tools at some point or another, but the personal favorite came from an unlikely source, the Visual C++ text editor. This editor proved to be very valuable because of its laying down of code with good tabbing to make it easy to read and the built-in bracket-checking tool. This last tool was a godsend. When dealing with large .wrl files, particularly when they include many nested LODs, the bracket situation can easily fall out of control. A missing bracket can crash the file, and extra brackets can make for some very strange model behavior.

The best approach to take when doing a complicated .wrl is the piece-by-piece method. Sometimes it's not the most efficient means, but if a debugging nightmare does occur, earlier efficiency doesn't count for much. Basically it is a "one set at a time" creation process. After each new set of models is inputted and its LODs created, the model is tested one level at a time. The assumption is that by testing the file in this way, you can easily pinpoint a region where the problem has occurred. It is most likely in the new code, but this is not always the case. Sometimes the new code simply triggers an error that has been undetected elsewhere. However, the new code is usually the culprit. This approach minimizes the amount of searching you will need to do. After inputting 20K of new code and then observing an apparent error, your day is most likely going to be a little worse off than if only 4K of new code had just been entered.

Debugging is a continuous process. It would be unwise to build the entire .wrl before beginning the error-checking process. There are also some excellent tools available to handle this task more efficiently. WebSpace includes a program called VRMLint that does a pretty good job of parsing a file and pinpointing errors. However, I have had instances where the actual error was hundreds of lines away from where VRMLint had indicated it should be. The browsers themselves can be good debuggers since some indicate errors when parsing the file. You can also look for problems in the image of the model. This can sometimes give you a clear picture of where the error is occurring. As was mentioned earlier, the bracket-checker in the Visual C++ text editor proved to be very valuable. Other development environments will have similar tools. This is a personal process, dependent upon how the coder likes to determine errors, but the tools are out there to make the job a bit easier.

When creating a .wrl, remember that you have little control over the user. As sure as the sun sets, users will fly over to the one area of the model that just looks like hell. Remember that this is a 3-D world that includes a front, back, sides, top, and bottom. It is where a square becomes a cube and that its faces goes from two (front and back) to six. The third dimension on your computer screen is something of an

illusion. Three dimensions do not really exist on a flat computer screen. However, the models exist within the virtual 3-D space of the browser. The browser takes this 3-D model and interprets its image for the 2-D screen. The best way to generate consistency is to test a .wrl on more than one browser (and even platform if possible) to see what areas of code translate the best and what areas don't. This is the kind of experience and knowledge that others are very happy to pay for.

An awareness of browser limitations can be an important consideration when creating a VRML file. Some browsers have already instituted their own VRML extensions similar to Netscape's extensions to HTML. If your .wrl file depends on browser-specific nodes that are not part of the VRML spec, then you should make this clear to the user. You should also realize that some people will not be very happy about this.

You also need to be aware that some of the browsers currently available cannot show texture maps. Obviously, without texture maps, our earlier San Francisco fly-in would be pretty dull. But this is not always the case. Don't ignore the materials quality of your models simply because they are mapped. Set a material color that will still have some effectiveness in the scene if a browser cannot show textures or, due to some problem, a Texture2 node fails to load.

Thinking About Speed

Most VRML files are designed to be run via the Internet. However, we do our testing on a local machine. What this means is that often the creator does not discover until the end of the process that although they have a beautiful file that is only 75K (a small/medium download for the modem user) the associated texture files and such total a couple of megs! Large files are a pragmatic reality, but minimizing file size is always a high priority. One personal frustration is seeing a link to a .wrl file that is listed at 3MB or 4MB. Regardless of my interest in the file, that's a heavy load when on a modem, and I will usually pass it by. Therefore as a point of consideration to users, indicating file size is a big must.

Speed and efficiency of code is another big factor. VRML files are a tough processing job. On-the-fly 3D rendering can grind a high-speed workstation down to a crawl. It is very important to consider this when creating the file. If you can make it out of primitives, do so. The browsers are optimized to handle primitives, even deformed. So to make the browser's job easier and your files view better, remember those primitives.

This danger really comes into play when using a modeler and converting the files. None of the conversion tools I've seen translates the file into VRML primitives. This has to be done manually after the conversion is complete, otherwise every model, including the obvious primitive possibilities, will consist of IndexedFaceSet nodes. Sometimes you want this as in the LOD Earth, but more often you won't. Use the conversion to identify the extents of the model (by checking the coordinate ranges of its vertices) so that it is easier to use the Transform node and place the primitive afterwards.

Use of primitives leads to a short rehash of mapping textures to objects. Primitives are pre-mapped by the browser code. If the mapping is extremely sensitive or vital, the non-primitive approach may be necessary. The question often arises about the non-primitives that do not have mapping coordinates applied. How exactly does the browser do the mapping? My experience indicates that most browsers use the bounding box of an irregular model to determine mapping coordinates. Sometimes this works fine, other times it's not so great. Here is a time when testing on different browsers can make a big difference.

In general, if you can map an irregular object manually, it is wise to do so. This will certainly cause the .wrl file to grow, but it also increases its compatibility across the array of browsers and gives the creator a good level of control over the display of the model.

Most of the troubles and problems encountered during the creation of .wrl files up to this point fall into one of the above categories. Keeping in mind the limitations of the VRML spec and the browsers that interpret it makes all the difference in the world when trying to develop .wrl files that many people can enjoy. Speed and size considerations and careful debugging often go unnoticed by users, but they certainly know when these issues haven't been considered. None of this is to say that all .wrl files should be tight little objects that can download in seconds. An extremely complex and detailed .wrl can be a beautiful model to view and travel through. Paying careful attention to these various issues simply makes your beautiful models more fun!

Some Thoughts on VRML's Potential

I have probably unfairly concentrated on the limitations and work-arounds necessary when dealing with VRML. To conclude this piece, I would like to take a few moments to focus a bit on the gee-whiz aspects of VRML and take a peek at the possibilities that lie ahead. It is a young and potentially rich field. Further refinements and improvements of browsers and the VRML specification, as well as the various browser enhancements that are already appearing, promise to make the VRML of the future an even richer development environment than it is today.

We are already beginning to see the realization of a number of special VRML enhancements. The ability to play sounds in a LOD or to have a continuous background music score greatly enhances the experience that a user has. To watch an object change by using a LOD and then hear an accompanying clip creates a multimedia experience that helps to bring 3D to an even more real level.

The WWWAnchor is a VRML feature that opens amazing doors for a .wrl file. Rather than being some neat object in 3D space that a user can fly around or spin, this node permits 3D interface design (not just 3D shaded buttons). Use of such a design was part of the consideration in the creation of a VRML model known as Weather World (see Fig. 21.10). Originally created to demo Paper Software's WebFX browser, this model shows the viewer a 3D model of the Earth with the various weather satellites in orbit as well as the view of the Earth from each satellite. By clicking on a satellite, you receive the latest infrared image from that satellite showing the cloud cover in that region of the world. Not a particularly difficult model to create but a useful one. Further enhancements will completely eliminate the satellites and replace them with a continuously updated cloud map on the Earth itself so that the user need only spin the globe to view the area they are interested in. Users will be able to zoom in and change from a cloud display to a ground-based radar display showing precipitation, and then zoom into a local area to receive the latest weather forecast or current report.

Transparency is another aspect of the VRML spec that can be utilized for dramatic and interesting images. In addition, some browsers support the transparent GIF file. This gives even greater control over the transparency of the materials. Using transparency, apparently complex objects can be created with very simple geometry—even primitives. In the course of designing a logo model, one of the elements was a film reel. Rather than model the film reel, a cylinder was used (it couldn't be a primitive because of the different mappings that were needed). A transparent GIF was applied to the cylinder which gave the

illusion of a full 3D model. It also allowed for another trick utilizing a browser-specific enhancement.

Fig. 21.10

Screenshot of Weather World.

The next trick was to make the film reel spin in a continuous loop. Using the animation capabilities of one of the browsers, an animated GIF strip was created. It succeeded in creating a film reel model that turns continuously as the user travels the .wrl. In addition because of the transparency and the use of two-sided polygons, you can actually look through the openings in the reel and see the other half of the reel on the other side!

The use of animation and video will push the capabilities of VRML forward. While there are no specs yet defined for this purpose (any existence of these items are browser-specific) the opportunities it presents are almost unfathomable. When one looks at the way sound and the moving image have revolutionized the way people view the desktop PC, it is obvious that such an enhancement in the VRML world could present an equivalent revolution.

New enhancements to the HTML browsers also enhance the capabilities of VRML. The ability to embed an active .wrl file into an HTML page, or have it appear in a separate active frame, means that the .wrl file model can be integrated into the HTML code to either enhance the page, provide examples or models that would be less effective when viewed as static, flat images, or to allow links that change the HTML page while keeping the embedded .wrl intact.

One amazing capability slowly coming to fruition is the integration of the JAVA language with many VRML browsers. JAVA is a language developed by Sun Microsystems to enhance the power of the Web by using small applets that allow users to interact with a Web page. These applets are downloaded from the server to the user's local machine and run locally. This creates a "one code fits all" approach to interactive programming on the Web. Integration with VRML browsers would permit active models that respond to events (not just locations as LODs do), AI routines for models to follow, and constantly changing or updating .wrl files that don't require new downloads because all the work is being done on the local user's machine! The possibilities are truly mind-numbing for it offers something that many users still find unbelievableófully interactive 3-D worlds.

VRML has a lot ahead of it; but the important thing to remember is that many of the possibilities are not in the future, but are here now. The true limits of the current VRML spec have not been pushed. Many of the current tools can do incredible things in VRML; they only require a creator who's willing to push that extra mile or try that experiment to bring something to fruition. We are still in the early days of this new language, and down the road we will look back upon many of the "amazing" things being done today and see them as archaic or trivial. But this doesn't make them any less worthy. The true value of VRML today is that it presents an open foundation to those who wish to try to build something upon it.

My suggestion? Give it a try.

To see the LOD Earth and other neat sites, visit Paper Software's Cool Worlds Webpage at <http://www.paperinc.com/wrls.html>.

**Table
Of
Contents**

**Next
Chapter**

Notice: This material is excerpted from [Special Edition Using VRML](#), ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



22-Some Hand-Crafted Worlds

Worlds created using authoring packages or translators are all fine and good, but quite often they're bulky and inefficient. The smallest, fastest, and in many cases most interesting worlds are those that have been built by hand.

In this chapter, we're going to examine three such worlds. The first two are ones I created specifically for this chapter, to illustrate simple world-building techniques. The third is created by a high school student named Richard Sibley, who taught himself VRML and created a simple but well-crafted world.

In this chapter, we will

- Look at three sample worlds
- Learn to use libraries of existing objects
- See an example of LOD in action
- Work with materials and textures

Above all, we will see that you don't need powerful tools to create effective worlds.

Example 1: The Farm

Our first example is a farm. There's a simple algorithm that we'll be using to create all of our worlds, and the farm gives us a chance to put that algorithm to work. It works as follows:

1. Decide what objects you want to include.
2. Create a VRML file to describe the world.
3. For each object, figure out how to obtain it and then add it to the world.
4. Edit the file to reposition and reorient the objects as needed.

In our case, we'll keep the world simple. Our set of objects will include a grove of trees, some grass, a barn, and a truck.

The Trees

To create the grove of trees, we'll use two different programs: an object generator and a world generator. The object generator is TreeTool, a commercial package from InWorld VR. It enables you to interactively create trees; it's described in more detail in Chapter 26. The trees are converted to VRML using the WCVT2POV utility, also described in Chapter 26.

The combination of TreeTool and WCVT2POV produces a single VRML tree. What we really want is a grove of trees, of varying sizes and random placements. We could use Fountain or some other authoring package, but in this case there's an easier way: a program called FOREST, which is described in Chapter 26 and examined in great detail (including the source code!) in Chapter 32. The full source listing for FOREST is in Appendix E.

The output of FOREST is redirected into a file called forest.wrl:

```
forest -r 10 -c 10 -h 5 -v 5 >forest.wrl
```

We then create our overall world file, and use WWWInline to load the forest:

```
<LX>}
```

Figure 22.1 shows the resulting random grove of trees.

These trees are of different sizes and orientations, but if you look closely you'll see they're all the same basic shape. If you wanted to, you could generate another random tree using TreeTool and run FOREST again; the resulting grove could be WWWInline'd as well, so the two would be merged. There would have to be some trial-and-error, since it's quite possible that after merging there would be more than one tree in a given spot (which would look quite strange). Since you probably don't have TreeTool, I've included some sample trees on the CD-ROM that comes with this book.

Fig. 22.1

Some trees floating in space.

The Grass

The grass will be a single rectangular polygon, with a texture map applied to it. There are a number of texture sites out on the net, so finding a grass texture isn't hard. With the grass added, our world file now looks like this:

```
<LX>}
```

The grove of trees has been shifted slightly, to avoid having trees right on the edge of the grassy plain. If we wanted to, we could've redimensioned the grass instead.

The texture is loaded from the file "grass01.gif," and a texture transform is used to scale the grass 25 times in each direction. Because of the default REPEAT setting for wrapS and wrapT, it gets tiled. This is very much like the checkerboard example that we looked at in Chapter 17.

The Material node specifies the color green; if texturing is turned off (or not implemented in a particular

browser) then the user will at least see a green rectangle for the ground.

The coordinates of the rectangle are specified using a `Coordinate3` node, and the `IndexedFaceSet` is created. Note the one-to-one correspondence between entries in the `textureCoordIndex` field and the `coordIndex` field. With the grass added, our world now looks like Figure 22.2.

[Fig. 22.2](#)

Some brilliant trees, outstanding in their field.

The use of texturing slows down the performance of the world, so you may decide it's not worth it. You can experiment by simply commenting out the `Texture2` node in the world file.

The Barn

Our next step is to add the barn. We could certainly create one in a modeling package, but in this case we don't have to. There's a Web site called the MeshMart (<http://cedar.cic.net/~rtilmann/mm/index.htm>) that has a great collection of public-domain objects. Among them is a barn, in 3D Studio format.

We use `WCVT2POV` to convert the 3DS file into VRML; refer to Chapter 26 for details. We can insert the barn into our world pretty easily:

```
<LX>}
```

Notice that the barn was scaled to 1/50th its default size; whoever built it, built it big!

There's another problem: the world as shown in the next code listing will have the barn positioned at the origin. Not only that, but the coordinate system the barn was created in is different from the one used by VRML; as shown in Figure 22.3, the barn winds up being tipped on its side!

[Fig. 22.3](#)

Man, what a storm!

The solution is to add a rotation that turns the barn 90 degrees (1.5708 radians) around its local X axis (1 0 0). We also threw in a translation to put it in the right place:

```
<LX>}
```

Now it looks a lot better, as shown in Figure 22.4.

[Fig. 22.4](#)

Green Acres is the place to be!

Our world is almost complete, but let's add one more item to fill it out a bit.

The Truck

Once again, we turn to the Mesh Mart to find an item we need. They have a selection of trucks; none of them are quite what we're looking for (that's one disadvantage of relying on libraries) but there's one that's not bad: a "semi."

Once again, WCVT2POV lets us convert the file from 3D Studio format to VRML. Our final version of the farm, in VRML, is shown in the following listing.

```
<LX>}
```

Notice the block of comments at the start; this is to let people know who built the world and where the objects came from. You could also put this in an Info node, of course; the advantage of doing that is that it would (theoretically) get preserved by programs that read and write VRML files, whereas regular comments (created using the # character) would typically get discarded.

The final world is shown in Figure 22.5.

Note that we could keep adding items to this world indefinitely. People could also WWWInline this world in another one, just as we WWWInline'd the items we used. Since it loads its geometry and textures from other files, the world itself is quite small—less than 1K (1024 bytes).

How It Was Done

The process of interactively placing objects, rotating them, and so on is pretty tricky. One approach that works very well is to open up two windows on the screen, as shown in Figure 22.6. One window is your VRML browser (showing the world) and the other is an editor (containing your VRML code for the world). You can make a change, save the file, then click the "reload" button in the browser to see the effect of the change.

[Fig. 22.5](#)

The whole works.

Another useful technique is to comment out parts of the world you're not working on; this will make it reload and render faster. When you've made the necessary changes, just un-comment everything and away you go.

[Fig. 22.6](#)

The two-window approach to world-building.

Example 2 An Alien World

Our second world is inspired by the covers of pulp science fiction novels from the 1940's and 1950's. It'll feature an alien landscape, a rocket ship, an astronaut, and a big ugly monster.

The Landscape

Terrain is tricky to do by hand, but easy to create algorithmically. For our alien landscape, we'll be using a program called FRGEN; it's described in detail in Chapter 26. It's a fractal landscape generator that can produce a wide variety of different types of terrain; in this case, we'll be using it as follows:

frgen -or plain

This will read the file "plain.fr", which contains a description of the starting shape of the fractal. It will generate RAW format output (the "-or" option), which we convert to VRML using WCVT2POV.

Our world now looks like this:

<LX>}

Although the landscape is rectangular, it's generated so that it lies at a 45 degree angle to the coordinate system. We solve that by adding a rotation of 45 degrees (0.7854 radians) around the Y axis (0 1 0). We also scale the terrain to a more reasonable size while we're at it. The result looks like Figure 22.7.

Other landscapes could certainly have been used; however, once one is generated it will be important to stick with it (as we'll see later).

Fig. 22.7

The planet Xenon, before humans arrived.

The Rocket

Again, we could create the rocket ourselves using a modeling package or a CAD program. Fortunately, the Mesh Mart has a ship of just the right vintage for us. Again, it's converted using WCVT2POV, which is also used to change its color from red (which would blend in too much with the red alien landscape) to a shiny silver.

However, the ship needs to be re-oriented (90 degrees around the X axis) and scaled to the right size. Most importantly, it also has to be positioned on the landscape. This took a lot of fiddling, since the landscape is uneven and it's hard to know the height (Y coordinate) of any given point.

If the Y value is too small, the rocket will appear to be buried in the sand; if it's too large, it will appear to be hovering over the ground. While both of those are reasonable, they're not what we want in this case; we would like our rocket to be resting upright on the surface of the planet.

If VRML had support for heightfields (which future versions will, in the form of an ElevationGrid node) then things would be a lot easier; the rocket would just be positioned at a Y coordinate that could be obtained from the X and Z coordinates. You would still need to choose a relatively level area, to ensure that the height of all four fins would still be at "ground level"; this is natural enough, since you'd have to do that if you were landing a real rocket on a real planet!

The following listing is the world with the rocket.

<LX>}

The planet with the newly arrived spaceship is shown in Figure 22.8.

[Fig. 22.8](#)

The humans arrive!

So far, the world seems safe enough. But appearances can be deceiving....

The Monster

Every good pulp science fiction novel featured a big ugly monster on the cover. To be true to the genre, we have to do the same.

Creating organic shapes (like weird space aliens) is very difficult. Fortunately, there are tools that make it easier; the one we're going to use is called LPARSER, and it creates organic shapes using things called L-systems. All of this is described in greater detail in Chapter 26, but for our world we'll just use an existing LPARSER file called "dinofly.ls." LPARSER writes a DXF file, which we convert to VRML using our old friend WCVT2POV. The resulting world, complete with a menacing creature, is shown in Figure 22.9.

[Fig. 22.9](#)

Uh-ohóthe planet's inhabited!

The world file now looks like this:

<LX>}

Again, some fiddling had to be done with scale factors, rotations, and translations to get the monster where we want it (including at the right height).

The Astronaut

An intrepid traveler from Earth is our next addition to this world. Once again, we turn to Mesh Mart. No astronauts as such, but a number of humanoid figures we can adapt. In this case, we use one called "man1.3ds." It's a 3D Studio file, which we can convert to VRML using WCVT2POV. In this case, however, we'll also use WCVT2POV to edit some of the colors. The object is constructed as a number of separate parts, each of which can have its color changed independently. The figure starts off as all-red (which would again blend into the alien landscape), so we create a spacesuit for him by turning most of his body into a light blue. The exceptions are the head and neck, which we'll be putting a helmet over; they are made flesh-tone. The resulting figure, scaled and translated, is shown in Figure 22.10.

[Fig. 22.10](#)

Our hapless hero.

Of course, without a space helmet his head would explode! (Well, not really, but he'd still be happier with one.)

The helmet is just a Sphere node, positioned around his head. It's contained within a larger Separator;

that way, the helmet moves with the astronaut if we decide to reposition him (or give him behavior, once it's implemented in VRML). The following code listing contains the astronaut with his helmet.

<LX>}

Notice the transparency value of 0.4 for the helmet; this makes it semi-transparent, as shown in Figure 22.11.

Fig. 22.11

Nice helmet, but look behind you!

Browsers that don't support transparency will "round down" the 0.4 to 0.0, so the helmet will be an opaque sphere. This is slightly wasteful, since the browser will have to spend some CPU cycles rendering the nice flesh-tone head and neck that you won't even be able to see; fortunately, most browsers do support transparency.

Using LOD

The LOD node was described back in Chapter 17; here we see it being used. This is another good reason for putting the world together by hand—a lot of authoring tools don't support LOD yet. Of course, a lot of browsers don't support it either, but that's another story.

In our case, we're going to give the astronaut three different levels of detail. The one we have already (the "man1.wrl" file) will be the most detailed, and it will only be visible out to a distance of approximately 30 meters (this is shown in Fig. 22.12). Beyond that, it will be replaced by a light blue cylinder for the body, and a white sphere for the helmet (as shown in Fig. 22.13). Beyond 40 meters, the entire astronaut is replaced by a light blue box of the appropriate dimensions (shown in Fig. 22.14).

Fig. 22.12

Our most-detailed version of the astronaut.

Fig. 22.13

From further away, he looks simpler.

Fig. 22.14

And from far away, he's just a box.

Note that Figures 22.12, 22.13, and 22.14 are "cheats"; I've adjusted the ranges to be smaller, so that you can get a better look at the different versions. The cylinder-and-sphere version wouldn't be used until you got farther away, for example.

The following code listing is our final version of the world, complete with an LOD node for the astronaut (check out the range field) and some comments at the top.

<LX>}

The entire world is only 75 lines long and less than 2K in size. No programming or modeling programs

were required. The two-window approach was again used to assemble and fine-tune the world, and it worked well.

Example 3 A Home Space

Our third and final example was contributed by Richard Sibley, a high school student who taught himself VRML using little more than the specification itself. The world is impressive, especially considering it was put together entirely by hand.

At the time Sibley began working on this world, there were no VRML authoring tools available; he built the entire room using a text editor and a VRML browser. As you can imagine, that's a time-consuming process; however, the results speak for themselves. The room is not only visually interesting, it's also very efficient; without texture maps, the entire room is less than 12K in size. After gzip compression, it shrinks down to 2K (which means it would transfer in less than a second on a 28.8K modem). That compares very favorably with most of the worlds available on the Web, which tend to be hundreds or thousands of kilobytes in size.

Here is Sibley's own description of his work:

"I decided to make a virtual room similar to mine because I could make actual measurements of the objects in my room and therefore none of the objects would be out of proportion. I thought it would be interesting to see how similar I could make it in comparison to my real room. Also, I could add things (such as the guitar) that I don't really have and I could make changes to my virtual room in order to improve upon my real room.

"My first step in creating the room was to graph all the objects in it. I used two graphsóone with the x and y axis, and one with the x and z axis. I began entering the data on the computer by creating the walls first. Then I used cubes to represent all of the objects in the room and little by little I added more and more detail to them. I then added texture mapping to a few of the objects. I think that this step made the biggest difference in creating a realistic environment. Finally, I added light sources.

"I think that the final version of my room looks quite good. It doesn't look like my real room that muchóbut that doesn't really matter. My favorite part of the room is the guitar. I love the green texture map and I really like the amount of detail I was able to use. My real guitar doesn't even look as good as the virtual one!"

Figure 22.15 shows the basic entry view of the world. Each of the pieces of furniture was based on an actual item in the real-world room; the objects were measured, and those measurements served as the basis for constructing the virtual equivalents.

[Fig. 22.15](#)

The entire room.

The room is complete, but only has three walls; the "fourth wall" was left off to allow us to see inside from far enough away to view the entire room at once. Figure 22.16 shows another view of the room.

[Fig. 22.16](#)

A different angle; note that one wall has been removed (at right).

The guitar in the near right corner of the room was one of the first items Sibley built, and it's almost as complex as the rest of the room put together; a closer view of it is seen in Figure 22.17. It's even more impressive when you realize he actually built it by hand, vertex by vertex.

[Fig. 22.17](#)

The guitar.

The room's textures were added at a fairly late stage in the development of the world, but they definitely enhance the realism of the scene. However, they also point out a problem with VRML software. Originally, Sibley started out using WorldView; the early version of that browser didn't support texture maps. His next browser was Fountain, and it was while using that package that he added the textures to the room.

It turns out that there are ambiguities in the VRML specification related to textures; as a result, texture-mapped IndexedFaceSets don't look the same in all browsers. In particular, Fountain and WebFX display the scene differently. Figure 22.18 shows the same file displayed in Fountain; compare it to Figure 22.15.

[Fig. 22.18](#)

The same room, seen in Fountain; note the differences in textures.

Overall, Sibley's work is a remarkable achievement.

From Here...

In this chapter, we've examined three worlds. All of them were built primarily or entirely by hand, without using any authoring tools. Make no mistake: authoring tools have their place. However, the amount of control and compactness you can achieve by hand-editing VRML has yet to be matched by any of the available tools. This will change eventually, but in the meantime, you should seriously consider doing it by hand. If nothing else, you'll learn a lot about how VRML works.

From here you have a couple of choices. In [Chapter 23](#), Stephen will take you on a tour of other VRML worlds that are available on the Web; in Chapters [24](#) and [25](#), he'll introduce you to some 3D modelers and worldbuilding tools. I'll be back in [Chapter 26](#), to show you some of the tools that were used in this chapter (and others besides!).



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



23-A Guide to Other Sample Environments on the Web

What's the good of having a car if you don't have a place to go? And what's the good of having a VRML browser if you don't know where to look for interesting VRML environments and models?

I thought that you might want some places to start your exploration of the three-dimensional Cyberspace, and so I've come up with the following list of our favorites. This is just a short guide to help you start on your journey. But at the end of this chapter, I'll give you some help for finding more.

The following list provides the name of the site, its URL, and a short description of what you will find. The location of each site was accurate at the time that I wrote it, but material on the Web has a nasty habit of migrating to other places or disappearing altogether without warning. I have looked for a variety of material: entertaining, provocative, and (gasp!) educational. I hope that you will find the kinds of VRML applications that will appeal to your aesthetic sense and inspire you to create your own space in the virtual universe.

Arts and Entertainment

Chimerium: An Interactive VRML Installation.

<http://www.construct.net/projects/chimerium.welcome.html>

Chimerium is a virtual world populated by different forms of mythical creatures called Chimera. These Chimera are exhibiting animated motion around a constantly changing environment. You can create your own Chimera from a library of virtual objects and watch them interact with creatures created by other users.

Interactive Origami in VRML. <http://www.neuro.sfc.keio.ac.jp/~aly/polygon/vrml/ika>

This is one of my favorite sites for cultural reasons. It shows you step-by-step how to create origami: the Japanese art of paperfolding. The models show you how to create different origami squid. You can manipulate each model to see how the folds are made and then click on the model to move on to the next

step. You will need to access this site using a Web browser before going on to the VRML models. A very Zen kind of experience.

The VR Lounge. <http://www.iuma.com/IUMA-2.0/vrml>

The Internet Underground Music Archive's Virtual Reality Lounge is a site where you can find out about the underground music scene. You can listen to sound-bites of many of the bands in the archive.

Virtual Vegas. <http://www.virtualvegas.com>

Virtual Vegas is a virtual gambling environment. It is a full-service, on-line entertainment center where you will find a movie theater, music hall, and places where you can "meet" other users. It even lets you "gamble" using virtual cash. You can play VRML slot machines using a combination of a VRML browser and a Web browser. Best of all, you won't lose your shirt (yet).

WAXWeb. <http://bug.village.virginia.edu/>

WAXWeb is based on the film Wax, or The Discovery of Television among the Bees directed by David Blair. Blair and a team of computer people have converted the film into an interactive movie, website, and MOO. It follows the original film's narrative thread, but adds a number of interesting (and often disturbing) elements through its use of sound, video, and 3D models. Bizarre and intriguing.

Science

Black Holes, Galaxies, and Teukolsky Waves from the Relativity Group.
<http://jean-luc.ncsa.uiuc.edu/>

This site has a number of VRML models of astronomical and physics-related material. If you have ever wanted to look around a black hole, this is the place to be.

VRML in Chemistry. <http://ws05.pc.chemie.th-darmstadt.de/vrml/>

This VRML site displays models of the enzyme Cytochrome P450. It includes links to help you navigate different parts of the model. A good application of VRML to scientific modeling.

VRML 3D Models. <http://www.chem.vt.edu/yip/VR/3DAO.html>

3D models of atomic orbitals. It includes models of water in different states (steam, ice, and liquid).

VRML Sites with Behaviors

CAVE Applications. <http://jaka.eecs.uic.edu/dave/vrml/CAVE>

If you don't have access to a CAVE Automatic Virtual Environment and you can't put down the big bucks it would take to set up your own, this site is the next best thing. It is a collection of CAVE

environments that have been re-created using VRML. Some are pretty complex, so you will need a fair amount of memory and computing power to get the full effect.

Modzelewskiego Street, Warsaw. <http://andante.iss.uw.edu.pl/cgi-bin/modzel>

This VRML environment is a re-creation of a part of Modzelewskiego Street in Warsaw, Poland. The light and sun position represents the actual sun position in Warsaw. During the night, the street is illuminated by city lights. The different viewpoints that have been programmed into the environment include Jacek Szamrej's apartment window. Jacek is the creator of this model. Definitely a scene where you will want to turn off your browser's headlights.

Silicon Graphics Depot: Kitchen Planner.
http://webpace.sgi.com/projects/kitchen/sa_demo/index.html

Starting from a Web page, you can create a custom kitchen from a library of parts. You can view models of each part individually along with their process and other information. Once you have selected the items and design you want for your kitchen, you can explore a VRML model of it. The purpose of this application is to show how companies can use VRML and database technology to market their products and services. It is a pretty cool use of VRML, too.

The Virtual Underworld. <http://www-syntim.inria.fr/syntim/underworld/underworld.html>

This VRML site is a world where you can connect your own VRML environment to the Underworld's corridors. A fun way to look at other VRML sites.

VRML Abacus. <http://speckle.ncsl.nist.gov/greg-bin/abacus>

The VRML Abacus is a "working" model of the ancient calculating machine. You can click on the beads to "move" them.

VRML Globe. <http://www.gwha.com/~jym/VRMLWorld.wrl.gz>

While the VRML globe itself is a rather plain textured sphere, its main attractions are the small spheres that dot it. Click on any of those smaller ones and you will go to another VRML site. This is a g-zipped file, so make sure that your browser can uncompress it.

Other VRML Sites

Experimenting with VRML. <http://amber.rc.arizona.edu/vrml.html>

A collection of VRML models created at the University of Arizona. It includes models of a Polynesian platform bird feeder and a Pirogue cajun canoe.

VirtualSOMA. <http://www.hyperion.com/planet9/vrsoma.htm>

A re-creation of the South of Market Street Area (SOMA): the part of San Francisco where most of the

Internet, software, and computer animation development occurs. VirtualSOMA is the first virtual city to be loaded on to the Internet. You can click on buildings to walk into them, play an animation, or link up with the Web home page of the company occupying the building.

Object Libraries: Where to Get Stuff for Your Environment

Gallery Warehouse. <http://www.newcollege.edu/vrmlab/Warehouse/>

A site containing various .wrl files maintained by the New College of California.

Mesh Mart. <http://cedar.cic.net/~rtilmann/mm/index.html>

The Mesh Mart has been developed to provide the Internet community with a source for 3D mesh object files. These files are not in VRML format, but they can be converted using various utilities such as WCVT2POV (see Ch. 26).

VRML Models. <http://www.ocnus.com/models/models.htm>

A collection of VRML models that have all been tested using vrmlint. All the files are g-zipped.

The VRML Object Supermarket.
<http://www.dcs.ed.ac.uk/generated/package-links/objects/vrml.html>

The VRML Object Supermarket is maintained by the United Kingdom Virtual Reality Special Interest Group (UK VR-SIG). Its aim is to provide a collection of small .wrl files for creating VRML applications. Part of this collection can be found on the CD-ROM accompanying this book.

Where to Go from Here

Now that you've explored some or all of the sites that we have listed, where do you go to find more? And how do you learn about any new site being posted on the Web?

One place to look is the VRML Repository, (<http://www.sdsc.edu/vrml>). The Repository is the central link to all VRML related material that is available online. It includes a page with links to a large variety of VRML environments on the Internet (<http://rosebud.sdsc.edu/SDSC/Partners/vrml/examples.html>).

If you're interested in the technical side of VRML, you should subscribe to the VRML Mailing List. The list deals with all VRML related issues, and is one of the main points where the future of the VRML specification is being discussed. To subscribe to the mailgroup, send an e-mail message to info-rama@wired.com. Do not put anything in the subject line, and type subscribe www-vrml <your e-mail address> as the only line in the body. The sequence <your e-mail address> should be replaced by your e-mail address, of course.

You can also subscribe to the vrml-behaviors Mailing List. It is an unmoderated technical discussion on

describing behaviors within VRML. Topics include how to add interaction and animation behaviors to VRML, scripting language issues, implementation details, etcetera. To subscribe to this group, send an e-mail message to listserv@sdsc.edu. Do not put anything in the subject line, and type add `vrml-behaviors` as the only line in the body.

We strongly recommend that you "lurk" on these mailing lists for a while before jumping into the discussion.

An indication of the phenomenal growth of VRML can be seen in the number of VRML-related sites on the Internet. When the VRML 1.0 Specification was first released, a search for "VRML" on the Webcrawler search engine would come up with about 40 hits. Today, that figure is around 600.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



24-An Overview of 3D Modelers

While there are many advantages in programming VRML files using an ASCII editor, there are times when being able to see what you are creating as you go along can make your life easier. This is particularly true when you are trying to create objects that are more complex than a collection of simple shapes. The solution is to use a 3D modeler.

The distinction between a 3D modeler and a worldbuilding toolkit is that the former is better at creating objects rather than designing entire environments. This distinction can often be blurry. For example, Fountain is listed here as a 3D modeler; however, it can also be used to create a VRML environment. Conversely, Virtus Walkthrough is a worldbuilding toolkit, but it can be used to create a single object. You should regard these distinctions as flexible. There are also utilities that serve as "object generators" and "world generators"; Bernie will be discussing these in Chapter 26, and showing how to program them in Chapter 32.

This chapter provides an overview of 3D modelers. It focuses on modelers that can save VRML files. However, I feel that it is also useful to make some notes about non-VRML modelers. The chapter also examines the following issues:

- What does a 3D modeler do and how does it work?
- How does computer hardware factor into a 3D modeler's performance?
- When creating a VRML object is better done with an ASCII editor rather than a 3D modeler.
- An outline of some 3D resources.

What Does a 3D Modeler Do and How Does It Work?

Chances are that if you have seen a movie in the last few years, you have probably come across the work of a 3D modeler. The dinosaurs in Jurassic Park, for example, were created using sophisticated 3D modeling software. Moreover, many of today's products are designed using 3D modelers before an actual prototype is made. And 3D modelers are being used to create material for everything from television station logos to educational multimedia.

We need to distinguish between computer animation and 3D modeling. If you think about how stop-motion animation is created (The Nightmare Before Christmas is a good example), you get an idea of the difference. Before anything could be filmed, models of the different characters had to be made. These models were then painstakingly moved frame by frame. The computer-generated dinosaurs in Jurassic Park worked in much the same way. The designers used 3D modeling programs to create 3D models of the various dinosaurs to be used in the movie. These 3D models were then "moved" frame by frame using a computer animation program. The difference between Jack the Pumpkin King and the Raptors was that the latter existed only in the computer as a series of coordinates, texture maps, and polygons.

So one way of thinking about 3D modelers is to see them, as the name implies, as programs that let you build 3D models. But a better way to understand them is to think about their relationship to a graphics browser. If we think of VRML browsers as interpreters of the VRML language, a 3D modeler would be our translator. Most 3D modelers provide a graphic interface that you can use to see what you are building as you are building it. Your ideas are presented in a visual form that you can manipulate on your computer screen. You can create shapes, alter them and/or add to them, add color and textures, and even manipulate lighting. At the same time, the 3D modeling program is storing all the information about your model in a format that it can recognize.

Let's say that you want to create a table for your environment. Using a 3D modeler, you can build a table with rectangular legs (see Fig. 24.1).

In order to store the information for this table, the modeler creates sets of polygons. The following is a partial listing of the VRML-formatted file produced by ClayWorks to create the table in Figure 24.1.

[Fig. 24.1](#)

A table created with the ClayWorks 3D modeler.

#VRML V1.0 ascii

Separator{

Coordinate3 {

point[

-5.5000 -1.8500 2.9500,

5.6500 -1.8500 2.9500,

5.6500 -1.8500 -2.9000,

-5.5000 -1.8500 -2.9000,

-5.5000 -2.4000 2.9500,

5.6500 -2.4000 2.9500,

5.6500 -2.4000 -2.9000,

```

-5.5000 -2.4000 -2.9000,
-5.1000 1.7500 -2.0500,
-4.6500 1.7500 -2.0500,
-4.6500 1.7500 -2.5000,
... }

Material { ambientColor 0.65625 0.81250 0.92188}

IndexedFaceSet{
  coordIndex[
    6,7,3,2,-1
  ]
}

Material { ambientColor 0.39063 0.43750 0.65625}

IndexedFaceSet{
  coordIndex[
    14,15,11,10,-1
  ]
}

...}

```

You can see that the ClayWorks modeler does not produce cubes or other primitives, but individual polygons. Moreover, each face of the polygon is defined separately. This method of recording data takes up a lot of disk space. The full listing of the table in Figure 24.1 takes up more than five pages! However, most 3D modelers use some variation of this method because it makes it easier to let users change the shape, color, texture, and lighting of an object.

How a 3D modeler saves all this information also depends on the software you are using. All modelers save coordinate information; but how they use this information to create shapes can differ. For example, some modeling programs like ClayWorks use polygons to store details about the shape of their models. This means that all your models are constructed using shapes with straight edges. To get a rounded shape, these polygonal modelers have to use a large number of polygons. Other types of modelers use splines to create their models. The advantage of this modeling method is that it lets you create truly rounded shapes. Some modelers use both polygonal and spline-based methods, but these modelers are always better at one method than the other. However, VRML does not use spline-based modeling and there are no plans at the moment to introduce it. Therefore, generating a polygonal 3D model is the only option for the time-being (even if the modeler uses splines internally).

3D modelers also use different file formats to save their data. A VRML file is usually saved in ASCII, but it can be compressed using the g-zip format. However, most of the other formats store their modeling data in binary form as a way to save space. There is no 3D modeler available that lets you work on and save a VRML file compressed with g-zip yet. If you have a large environment that you are working on and want to compress, the only thing you can do is to decompress it when you want to work on it and compress it after you are finished. As VRML becomes more popular, I expect that this situation will change.

Computer Hardware and 3D Modeler Performance

As with almost everything in the computer biz, in the world of 3D modeling, bigger is better: this is true for processing power, memory, video capability, etcetera. Using a Silicon Graphics workstation with 128 MB of RAM will be a much more pleasant experience when creating a complicated 3D model than using a 386 PC with 8 MB of RAM.

The reason that computer hardware plays such an important role in a 3D modeler's performance is the same reason for VRML browser performance: the computations needed to render and display a 3D model are enormous (see "What Do You Need to Use VRML?", Ch. 8). The burden on a 3D modeler is even greater because of the need for your computer to do computations on a collection of polygons rather than shape nodes and to allow for changes in the model.

Processing power is an important factor in modeling performance, but it is not as important as memory. Imagine that you are a computer running a 3D modeler. Your user creates a sphere that takes up 30 polygons. You must calculate the position, color or texture mapping, and the lighting and shading for that object. Now your user decides to alter the shape of the sphere by compressing the top and bottom. You must calculate the changes to each of the polygons and how these changes will affect the color and lighting. Moreover, you need to be making these calculations while at the same time rendering the object on the user's display. All these operations must be done in RAM. If you find that the amount of RAM is not sufficient to hold all this information at once, you must stop, write some of the information to your hard disk, perform the operations in RAM, reload the material you saved and do those operations, and then continue with the next set of commands being sent by the user. If you have to constantly write to the hard disk, the delays will grow.

Some of the burden on your processor and RAM can be offset by using a graphics accelerator board. These devices will off-load the calculations needed to display the image on your screen from your main processor and RAM, thereby freeing up resources to work directly on the model. In the case of 3D accelerator boards, you can transfer even more of the rendering burden away from the main processor. Chips like 3DLabs' GLINT and PERMEDIA processors are designed to do much of the 3D graphics calculations that your main processor would normally have to do.

Therefore, the hardware you bring to 3D modeling is as important as the software. These programs crave computer resources. Bringing more resources in terms of memory, processing power, and graphics capability to the process will greatly enhance your ability to create the models you envision.

Using a 3D Modeler versus Building a VRML Model by Hand

The first consideration when using a 3D modeler is whether or not you really need to use one. As we have already seen, these programs store their objects as a set of polygons. At first glance, this does not seem to be much of a problem. However, the size of your VRML files will become more and more of an issue as your environments and objects become more and more complex.

The key to deciding which method you should use to create a model is the level of its complexity. Creating VRML models using an ASCII editor is not a very intuitive process. Trying to visualize an object using `Coordinate3` and `IndexedFaceSet` nodes is difficult at best.

However, visualizing shape nodes is relatively easy. And these nodes are much more compact than the same shapes produced by a 3D modeler. For example, we can create a sphere using the ClayWorks modeler, but the size of the file would take up over 13 K or about 12 pages of code. Compare this to the following VRML code:

Separator {

Material {ambientColor 0 1 0}

Sphere {radius 1}

}

This file would occupy only a few bytes and achieve the same effect as the sphere created by ClayWorks. Moreover, the `Transform` and `DEF` nodes allow us to change the size and create more spheres without taking up a great deal of file space.

The guiding principle that should govern whether or not you use a 3D modeling program or create the model by hand should be this: if the object you wish to create involves using complex or irregular shapes, then you should use a 3D modeler to create them. But also remember that you should always go through the modeler's output and look for places where you can optimize the code. The modeler is simply a tool to help make your job of creating VRML objects easier. You should not rely on it to do the hard work for you.

ClayWorks

ClayWorks is a fully functional 3D modeling package developed by Tim Lewis. It uses an intuitive interface that allows the user to create complex 3D models with relative ease. It will work on a 386 computer and currently operates under DOS. However, we have used the program in Windows 3.1 using the full screen option in a PIF file and have had no problems running it. A full, commercial version is to be released by Imaginative Software by the end of 1995.

The ClayWorks interface includes four viewing screens: three editing screens and a 3D graphics window. Each of the three editing windows is labeled X-Y, Y-Z, and X-Z, corresponding to front, side, and top views. The model is rendered in wireframe in these screens. The 3D graphics window displays the model

along with any colors or textures that have been applied. A toolbar and color palette are located on the left side of the screen, and function modes for the different drawing tools appear at the top.

ClayWorks provides you with a 3D cursor that lets you keep track of the three-dimensional aspects of the model. Since the editing screens show the model on a two-dimensional plane, it helps you keep track of the missing dimension. The 3D cursor shows up as a red cross. You can move it around any of the editing windows by clicking and holding the right mouse button and moving the mouse. As you move the cursor in one window, there will be a corresponding adjustment in the others.

The most important tool to master in ClayWorks is the Select tool. To use it, click on the arrow at the top left corner of the toolbar. You use this tool to select the points or shapes you want to manipulate. Click on a node and while holding the left mouse button down, move the cursor and drag it around. You will see a bounding box forming. To define the nodes or objects that you want to change, simply surround those points and release the button. You will see the bounding box, which you can use to resize the objects within it. You will also see points, which you can use to change the shape of the object.

The ClayWorks Drawing Functions

ClayWorks gives you a number of drawing tools to create your models. To associate a color with the object you draw with these tools, click on the appropriate color on the palette. This chooses a foreground color that will be the default color until you click on another. The following is a list of the different drawing functions and how to use them:

Square: to use this tool, click and drag in any of the editing windows. The surface of the square is rendered as a double-sided polygon in the current foreground color.

Cube: click and drag in any of the editing windows. The depth of the cube is taken from the width, and all its polygons are single-sided.

Dot: creates a single vertex with no polygon or wireframe connections.

Cone: creates a cone with a base. Its polygons are single-sided, but you can make them double-sided using the "Selection/editing" functions. Click and drag the mouse to create the base of the cone. You can then increase the height of the cone by using the Select tool and extruding the bounding box. The cylinder's base can be any equilateral polygon with 3 to 30 sides.

Circle: click and drag the mouse until you get the size of the circle you want. This is not an actual circle but a flat, equilateral polygon with 3 to 30 sides. Its surface is double-sided.

Cylinder: creates a cylinder with a base and a top. Its polygons are single-sided, but you can make them double-sided using the "Selection/editing" functions. Click and drag the mouse to create its base. You can then increase its height by using the Select tool and extruding the bounding box. The cone's base can be any equilateral polygon with 3 to 30 sides.

Sphere: click and drag the mouse to create the sphere. If you want its equator to go east to west and face the front view, create it in the X-Z editing window. If you want the equator to go north to south and face the front view, create it in the Y-Z window. The selector at the top of the screen controls the number of segments and the number of sides for each.

Rotated Form: this drawing mode can be used only when some nodes have been selected. Clicking in an editing window will create a rotated form by sweeping the pattern of the selected nodes through 360 degrees around the center of the editing space. Again, the number of segments range from 3 to 30.

Grid: creates a flat grid with the number of squares specified using the two number boxes at the top of the screen. Click and drag the mouse until you have the grid size you want.

i Text: selecting this tool lets you enter text into the gray box at the top of the screen. To position the text in the model, click in an editing window. The text will appear at that point using the currently loaded font. To load a font, go to the font menu and select "load font." Fonts available in ClayWorks are Borland ".CHR" stroked font files. Borland fonts are wireframe and do not create surfaces when drawn. You can give them depth by using the Extrude tool.

Saving Models Created in ClayWorks

ClayWorks allows you to save your models in a number of different formats:

- 3d: ClayWorks' internal file format
- RWX: RenderWare scripting format
- POV: the Point-of-View raytracer scene file format
- OBJ: a simple triangle format
- PLG: the file format used by Avril and Rend386
- GEO: the Geomview format (for OOGL)
- VRML: the VRML file format (uncompressed)

Saving in formats other than Clayworks' own 3D will usually work, but in some cases details may be lost.

CAUTION

In the current versions of ClayWorks, an error in the program saves VRML formatted files using a .WLD extension instead of the correct .WRL extension. This is a mistake in the extension designation only. The file itself is in ASCII. Therefore, you only need to rename the file using .WRL.

For More Information...

For more information about ClayWorks, see <http://cent1.lancs.ac.uk/tim/clay.html>.

Fountain (Caligari Corporation)

Caligari Corporation's Fountain is an integrated VRML authoring tool that allows you to build environments and browse VRML worlds on the Internet (see the Fountain section in "[VRML Browsers](#)," [Ch. 9](#)). It operates under Windows 3.1, Windows NT, and Windows 95. It is optimized to take advantage of any 3D acceleration card that comes with a 3DR driver. Fountain technology has been licensed by

Microsoft for inclusion in its "Blackbird" graphics package.

Although Fountain can be used to build entire environments, we feel that the program's real strengths are in its modeling abilities. The program uses a single, 3D perspective view with pull-up menus. Its engine is based on the company's trueSpace 3D modeler and computer animation program. Its graphics capability includes rapid modeling in perspective space, manipulation of texture-mapped VRML objects in real time, and interactive lighting. It includes 3D shape creation tools such as 3D primitives, 2D polygons, face editing, lathing, sweeping and tips, as well as the extrusion of TrueType fonts. You can choose between infinite and local light sources, and paint not only entire objects, but also individual faces and vertices. As well as importing VRML models, existing 3D geometry can be imported via trueSpace, DXF, 3D Studio, Wavefront, LightWave and Imagine filters, as well as 2D formats like PostScript, BMP, and JPG.

Making Objects with Fountain

When you first start Fountain, you will be in its browser mode. Select the worldbuilding icon on the toolbar. Once you are in worldbuilding mode, you can start creating a new environment by selecting File Scene New Scene. You will see a larger toolbar and a grid on the viewing area (see Fig. 24.2).

NOTE

If you want to know the function of any of the icons on the toolbar, simply move the cursor over the icon. The Help Bar will list its function. You can also turn the Help Bar on or off in the Help menu.

[Fig. 24.2](#)

Fountain's Worldbuilding mode.

Troubleshooting

Troubleshooting

While we created our Fountain model using the 3DR rendering setting, it has been our experience that building objects in this mode is very slow. By clicking and holding on the icon, you can switch to wireframe mode. Using this rendering method while constructing the various objects is much faster and allows you to have a better idea of the size and position of the objects in your model. When you are ready to apply materials and textures, switch back to 3DR mode.

To begin, let's make a cube. To do this, click on the Primitives panel button. This will bring up the Primitives panel, which gives a selection of different shapes. Clicking on a Primitive's icon with the right mouse button will call up that shape's properties panel. You can then control the size and attributes of that shape. Clicking on a Primitive's icon with the left mouse button will cause that shape to appear in the building area. We can create a cube by left-clicking on the Cube tool. A cube will appear on the viewing screen (see Fig. 24.3).

[Fig. 24.3](#)

Creating a cube using Fountain's Primitives toolbar.

We can now transform the cube using the tools in the Point Edit control panel. When using these tools, you control changes on the x and y axes by clicking and holding the left mouse button. You control changes on the z-axis by clicking and holding the right mouse button. Clicking and holding both buttons will make uniform changes on all three axes. For example, you can rotate your cube by clicking on the Rotate Object tool, clicking and holding on the cube, and moving the mouse. However, let's resize the cube. Click on the Scale Object icon and move the cursor to the cube. Click and hold the left mouse button and start moving the mouse. You will see that you can change the height and thickness of the cube as long as you keep the left mouse button pressed. When you have scaled the cube to the size you want, let go of the button. The cube is now more like a wall (see Fig. 24.4).

[Fig. 24.4](#)

The cube after being scaled.

We can repeat the process in order to add another wall to our model. However, we also want to move this new wall closer to one end of our first wall. In order to move an object, you need to click on the Object Move icon. Then move the cursor to the new cube, click and hold the left mouse button, and drag the object to its new location. You will notice that by placing the new wall against the first one, it appears as if they are joined (see Fig. 24.5).

[Fig. 24.5](#)

Two walls "joined" by using Move Object.

We can create a cone and a sphere using the same process. Simply click on the appropriate shape icon and move the mouse to where you want to place the object. Then click and hold the left and right mouse buttons, drag the mouse until the object is the right size, and then let go of the button. We now have a cone and a sphere in our model (see Fig. 24.6).

[Fig. 24.6](#)

Adding a cone and a sphere.

Once we have all the objects for our model, we can begin to add colors and textures to them. To add color, click on the Paint Faces icon. The four Paint panels—Material Preview, Color, Texture Map, and Shader Attributes—will appear on the screen (see Fig. 24.7). The Material Preview panel lets you see what your selections will look like before you apply them to an object. The Color panel lets you select the color that you want to apply. The Texture Map panel lets you control how you apply image files to an object. The Shader Attributes panel lets you adjust other material properties. You can smooth surfaces, add ambient lighting to specific objects, adjust a surface's shininess and roughness, and vary an object's transparency.

[Fig. 24.7](#)

The Paint panels.

If you want to color parts of the sphere blue, click on the shade of blue you want on the Materials Color palette. You then need to select which Paint tool you wish to use. The Paint Face tool applies the material

appearing in the Material Preview panel onto individual faces of an object. The Paint Vertices tool will paint one side of an object and vary the color intensity along each face. This tool will recognize only colors; other material attributes will be ignored. The Paint Over tool will replace any material attributes that have been previously applied. And the Paint Object tool will cover the entire object with the material appearing in the Material Preview panel. Click on the Paint Face icon. You can then begin to color all or only some of the polygons that make up the sphere by clicking on each one (see Fig. 24.8).

[Fig. 24.8](#)

Coloring individual polygons.

You can also color the entire object by selecting the Paint Object tool. Choose another color from the palette and click on the Paint Over icon. Move the cursor to the cone and click on it once. It will immediately become that color (see Fig. 24.9).

[Fig. 24.9](#)

Using the Funnel tool to color the cone.

Once you are finished coloring the objects in your model, you can save it using the VRML format by selecting the File menu and choosing the "Save VRML Scene" option. You will then get the "Export VRML File" dialog box. If you don't specify the directories where inline objects and textures are to be saved, they will be placed in the same directory as the .WRL file. Moreover if you do not specify filenames for your inlined objects, Fountain will save them using the first five characters of the .WRL's name and adding three digits for each inlined file starting from 000.

TIP

If you want to reduce the size of your VRML files, turn the "Pretty Formatting" and "Preserve Hierarchy" options off and decrease the precision to one or two digits. The "Inlining" options let you choose what should be saved in the main file and what should be saved in other files. You should inline all textures unless they are very small or you are using non-JPEG images.

The default setting for Fountain uses DOS pathnames for textures and inline file information. You can use the "Unix Paths" option to use forward slashes instead. Now that you have a VRML-compliant version of your model, you can look at your model in a VRML browser like WebSpace (see Figs. 24.10 and 24.11).

[Fig. 24.10](#)

The opening view of our Fountain- generated VRML model in WebSpace.

[Fig. 24.11](#)

Another view of our model in WebSpace.

Troubleshooting

Troubleshooting

While Fountain can produce files that are compliant with the VRML 1.0 specification, it also uses nodes that are compliant with only Caligari's version of VRML. The following listing shows some of the code from our model:

```
#VRML V1.0 ascii  
# created by Caligari Corp. Fountain (tm)  
# copyright 1995 Caligari Corporation  
  
Separator {  
  
CALIGARISceneInfo {  
  
fields [ SFVec3f background, SFVec3f environ,  
SFVec3f fogColor, SFBool fog, SFLong fogNear,  
SFLong fogFar, SFString envName, SFString _backgroundName ]  
  
background 0.502 0.502 0.502  
  
environ 0.000 0.000 0.000  
  
fogColor 0.502 0.502 0.502  
  
fogNear 1  
  
fogFar 500  
  
fog FALSE  
  
}  
  
TransformSeparator {  
  
MatrixTransform {  
  
matrix  
  
1.000 0.000 0.000 0  
0.000 0.000 -1.000 0  
0.000 1.000 0.000 0  
4.827 4.089 1.121 1  
  
}  
  
DEF LocLight PointLight {  
  
color 1.000 0.647 0.376
```

```

on TRUE
location 0.000 0.000 0.000
intensity 1.300
}
}
TransformSeparator {
  MatrixTransform {
    matrix
    1.000 0.000 0.000 0
    0.000 0.000 -1.000 0
    0.000 1.000 0.000 0
    -3.631 3.046 5.116 1
  }
}
DEF LocLight_1 PointLight {
  color 0.008 0.698 1.000
  on TRUE
  location 0.000 0.000 0.000
  intensity 1.300
}
}

```

You will notice that the file begins with one of Caligari's extensions to the VRML 1.0 specification.

Although most browsers ignore these extensions, there have been some reports on the VRML mailing list about rendering problems involving these nodes. The Fountain-generated models we tested, however, seemed to pose no problems in WebSpace 1.0 for Windows NT. There were, however, some texture-mapping problems resulting from ambiguities in the spec. We suggest that if you are planning to post VRML files created by Fountain, you should use the Save option that stores standard VRML code.

For More about Fountain...

To learn more about Fountain, see Caligari's Fountain page at <http://www.caligari.com/ws/fount.html>.

Two Other VRML-Compliant 3D Modelers

In addition to ClayWorks and Fountain, two other 3D modeling packages have included VRML in their list of compatible file formats.

A VRML-compliant 3D modeler for the SGI Sun and Fujitsu platforms is Radiance Software's Ez3d. There are plans to develop versions for Windows NT/95, HP-UX, and IBM/AIX. Ez3d is touted as an easy-to-use, low-cost modeling system. It uses a spline/polygonal modeling system for sculpting and manipulating objects with real-time feedback. It includes a "Potter" tool that allows you to bend and mold objects as if you are using your hand. The system comes bundled with the Rayshade raytracing software. You can visit Radiance Software's Ez3d Webpage at <http://www.webcom.com/~radiance/> for more information.

Strata Inc.'s StudioPro Blitz is a full modeling, rendering, and animation system for Macintosh computers. It provides full support for QuickDraw 3D, QuickTime VR, and VRML. Its modeling interface is well-designed and easy to use. It includes an extension called the 3d Sculpter. This program lets you distort an object by pulling and pushing on vertices. StudioPro also has excellent lighting controls and an extensive texture map library.

Is There Anywhere I Can Find 3D Models?

Of course, another way of getting 3D models for your VRML environment is to find a place where they already have been made. And there are such places popping up on the Web. Many of these sites let you download their models for free. They often include objects saved in VRML and other 3D file formats. If you find that the model you want to use is not available in VRML, however, do not fret. You can use one of the many utilities that have been created to convert other file formats into VRML.

If you want to find VRML and other models that are available for free, check the VRML Repository's list of object libraries at <http://rosebud.sdsc.edu/SDSC/Partners/vrml/examples.html>. (We have also included a list in "A Guide to Other Sample Environments on the Web," Ch. 23.) Also be sure to check the CD-ROM that comes with this book; there should be several object collections there.

Another possibility is to go to a commercial data set repository like Viewpoint Datalabs. Viewpoint collects and sells 3D models. It currently does not have data sets in VRML format; however, it does have models available in most of the popular commercial 3D file formats like Alias, Softimage, and 3D Studio. It would be possible to convert these files into VRML format using one of the utility programs currently available; however, there may be restrictions on their use.

Viewpoint's 3D models are very good in terms of their construction. Many computer graphics and animation companies use Viewpoint products for their work. However, most of these models would not be appropriate in a VRML model meant for general consumption on the Internet. Most of the company's models are extremely complex. Some of its models use over 100,000 polygons. Moreover, these models are very expensive. Indeed, their prices average in the hundreds of dollars.

If you would like more information about Viewpoint and its products, see its Webpage at <http://www.viewpoint.com/>.

From Here...

Using a 3D modeling program can help you build more complex VRML objects. But once you have the object, you will want to optimize it to minimize the size of the .wrl file. And you will probably also want to have an environment to put it in.

To learn about creating your own VRML environments or to learn about how to use VRML so that you can optimize the output of your 3D modeler, see the following:

- If you are ready to start working directly in VRML, jump to [Part IV, "Creating a VRML Environment from Scratch."](#) These chapters provide a step-by-step description of how to create VRML environments as well as the standards and practices for VRML programming.
- If you want to look at the experiences of others who have created their own VRML environments and models, see Part V, "Learning from the Pros: Sample VRML Environments."
- If you want to look at some VRML environments and models, see [Chapter 23, "A Guide to Other Sample Environments on the Web."](#) It lists some of the VRML sites that are currently available.
- If you want to see what programs are available to help you create VRML environments, see [Chapter 25, "An Overview of Worldbuilding Toolkits."](#) That chapter discusses how these programs can help you build your VRML environment.
- If you want to see what utilities are available to help you convert non-VRML files to VRML, see [Chapter 26, "VRML Utilities."](#)
- If you feel the need to create your own VRML utilities, see [Chapter 32, "Writing VRML Software."](#) That chapter will give you the basic knowledge that you will need to create a VRML browser and utilities from scratch.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



25 - An Overview of Worldbuilding Toolkits

Once you have a good collection of VRML objects that you want to show off, you'll need a place where you can display them. While we showed you how to create environments by hand (see "[Putting Up Walls](#)," [Ch. 12](#)), you may want to create environments that are more complex. In these cases, you will then want to think about using a worldbuilding toolkit.

This chapter provides an overview of worldbuilding toolkits that can save VRML files. It also examines the following issues:

- What does a worldbuilding toolkit do and how does it work?
- What is the difference between a 3D modeler and a worldbuilding toolkit?
- Using a worldbuilding toolkit versus building by hand.
- An outline of some worldbuilding toolkits.

What Does a Worldbuilding Toolkit Do and How Does It Work?

A worldbuilding toolkit is a program that helps make the process of creating a virtual environment a much easier task. It lets you design your environment using a variety of drawing tools and save it in a 3D file format. Moreover, as you plan your environment, you can see what it would be like to navigate it. You can make changes in your design and immediately see what effect they will have on how users will interact with the space.

Most worldbuilding toolkits provide you with an overhead view with a grid overlay as the main input area. This interface follows the blueprint metaphor. You "draw" your walls and doorways on the grid, and then position other objects using this plan as your guide. You create walls, floors, and ceilings of varying sizes using the drawing tools. You can also apply texture maps and set colors and lighting using an intuitive interface. Most worldbuilding toolkits also allow you to import objects created using 3D modelers and position them in your environment.

The most important feature of this type of software, however, is its ability to show you what your environment looks like as you build it. You can add an object and then immediately navigate the environment to see how it has been affected. You can see if adding a partition to a room will enhance its design. And if you don't like it, you can remove it and try something else. In this way, you can design your environment directly on the screen and experience it from the users' point of view while you are building it.

Like 3D modelers, worldbuilding toolkits save objects as polygons as opposed to higher level primitives. This method of storing data takes up a lot of disk space, but it also makes it easier to let users apply colors and textures to an object. Moreover, these programs allow you to create complex spaces, make use of odd angles, and combine different shapes to form partitions.

Like 3D modelers, worldbuilding toolkits use different file formats to save their data. VRML files are saved in ASCII. WebSpace Author allows you to work with g-zipped files, but if you want to compress your files using the g-zip format with most other software, you will need to run a compression program after you have finished building your environment. I expect that you will eventually find more worldbuilding toolkits that incorporate g-zip compression as VRML becomes more popular.

3D Modelers and Worldbuilding Toolkits

As you have probably noticed, there are many similarities between 3D modelers and worldbuilding toolkits. Both save 3D data; and both store objects as sets of polygons rather than using higher level primitives. And in many cases, the programs can serve as both 3D modelers and worldbuilding toolkits. Caligari's Fountain is one example of this hybrid software.

One factor that you can use to differentiate between a program that is more suited for building objects and one that is better at building an environment is its interface. In a worldbuilding toolkit, you will find that the interface is good at laying out an environment. It will give you an overhead view with which to build your environment and a grid that lets you measure the proportion of width to length. Using this type of interface will give you a sense of drawing a blueprint: noting the position of walls, adding and positioning objects, and figuring out how users will move from room to room.

The tools for a worldbuilding toolkit will differ from a 3D modeler as well. In designing an environment, you need tools that are geared more for building such things as walls and columns. Thus, most worldbuilders will have tools for making lines, cubes and cylinders. However, these programs sometimes omit shapes that are more suited for objects like spheres and cones. Indeed, some worldbuilders only provide tools for drawing cubes and lines.

Having made these distinctions, you will need to decide whether to use both a 3D modeler and a worldbuilding toolkit to build your VRML environments, or to choose only one and the other in order to avoid learning a separate interface. While the option of using a hybrid program like Fountain is a possible compromise, you will find that these hybrids usually are better at one task than the other. My personal preference is to use a 3D modeler. You will find that building an environment by hand is usually easier than creating a complex object. Moreover, as is noted below, many environments can be built using geometric primitives rather than sets of polygons. This will make a huge difference in the size of your .wrl file.

Using a Worldbuilding Toolkit versus Creating a VRML Environment by Hand

As with 3D modelers, the first consideration when using a worldbuilding toolkit is deciding whether or not you really need to use one. As we have already seen, worldbuilders store their objects as a set of polygons. This can drastically increase the size of your VRML files. This factor becomes troublesome as the complexity of your environment increases. Indeed given the efficiency of using primitives to define your virtual space, it would appear that there is no real need to use a worldbuilding toolkit at all.

However, there are times when using a worldbuilding toolkit is not only preferable, but necessary. It is not always desirable to use primitives to create your walls and columns. In some cases, you want and need to use polygons.

For instance, using a worldbuilding toolkit would make more sense than building by hand when you wish to have curved walls. In the VRML 1.0 specification, the only way to create a curved wall without having to use an entire primitive like a cylinder is to use a series of smaller cubes or polygons. But while using the Cube and Transform nodes may be more efficient in terms of file size, trying to create an entire wall in this way is not practical. Using a worldbuilding toolkit lets you create narrow polygons and adjust their angles easily and quickly.

Finally, if you are creating a complex VRML environment with a lot of objects, using a worldbuilding toolkit to position them is much more practical than trying to do it by hand. In a simple environment, it is easy to keep track of where everything should go. But in a large environment with many objects, you will find it extremely difficult to position everything correctly. Therefore, even if your environment uses only four square walls, you should use a worldbuilding toolkit whenever you have more than just a few objects to put in it.

Home Space Builder (ParaGraph International)

ParaGraph International's Home Space Builder (HSB) is an inexpensive and powerful 3D authoring tool for middle and low-end computers. Its minimum requirements are a 386/33Mhz PC-compatible computer with 4MB RAM, Windows, and a color monitor with 256 or more colors. To achieve better rendering performance, you will need a 486/33 or higher and 8 MB RAM. The program files take about 1.5 MB of disk space.

The HSB interface uses two viewing screens and a number of toolbars (see Fig. 25.1). The Walker screen and toolbox let you see and navigate your environment as you build it. The Plane Builder window and Builder toolkit are where you actually construct your environment. The Pictures toolbar allows you to add color, textures, and pictures to your environment.

[Fig. 25.1](#)

The Home Space Builder.

Home Space Builder can save files in Home Space's MUS 3D file format or in the VRML 1.0 standard. Extended features available in the beta 2 version include:

Drawing Walls and Faces

Applying textures with Transparent Colors

Sound Attachments, Sound Recording (WAV), and Background Music (MIDI)

Attaching and Playing Movies and Video (AVI, FLI)

Designing and running Walking Shows

Extended function attachments (executable modules, embedding home spaces, etcetera)

Multifunction attachments

i An extended set of textures, pictures, and sample Home Spaces

You build virtual environments using HSB's internal MUS format. When you have completed your world, you can convert your MUS file to the VRML format. HSB cannot read VRML files; it can only write them.

Making a Virtual Space Using Home Space Builder

Creating a virtual environment using HSB is a fairly easy process. You begin by creating walls using the Building tool in the Builder toolbox. Start by clicking on the button with the rectangle. This activates the building mode. Then you can begin building walls in the Plane Building window. Choose where you want to have one corner of your wall and click once with the left mouse button. Then drag the mouse to where you wish to set the opposite corner. As you drag the mouse in the window, you will see a box that outlines the shape of the wall. When you see that the outline is the size you want, click on the left mouse button again. You will see a rectangle representing the wall in the Plane Building window and the actual wall in the Walker window (see Fig. 25.2).

Troubleshooting

Troubleshooting

Why do the walls disappear?

While you may be tempted to use a single, large cube to create a VRML room using HSB, you need to be aware that the polygons that this program creates are single-sided. Thus, you will notice that the walls disappear the moment you enter the cube. You can, however, use the Cutting tool to "hollow out" your cube. This will make the walls within the cube appear.

Fig. 25.2

Creating a wall in Building mode.

Now you can try building a wall of a different height. To do this, you need to adjust the Height tool in the middle of the Plane Builder window. Click and hold on the right side of the gauge and move your mouse down. You will see the gauge lowering the height setting. In this case, we will set it at 50 percent of the

full height. By repeating the wall building process described above, we can build a second but shorter wall that is perpendicular to our original one (see Fig. 25.3).

[Fig. 25.3](#)

Creating a shorter wall.

We can add a texture map to the shorter wall by using the Picture toolbar (see Fig. 25.4). Click on the Wallpaper icon. Then choose which texture map you wish to use by clicking on its image. If the texture map file is in another directory, you can click on the Change Directory icon and shift to the one you wish to use.

[Fig. 25.4](#)

The Picture toolbar.

Move the cursor to the Walker window. You will see that the cursor has changed to the wallpaper cursor (see Fig. 25.5). Move the cursor to where you wish to apply the texture map and click on the left mouse button. The texture map will then be applied to that face of the polygon (see Fig. 25.6).

[Fig. 25.5](#)

Using the Wallpaper tool.

Note that the texture map has been applied to only one face of the wall. If you want to have the same pattern on all sides of an object, you must repeat this process for each side.

[Fig. 25.6](#)

The short wall with the brick texture map applied.

We can also add color to an object using the Paint tool. Click on the Paint icon in the Picture toolbar. This will bring up the color palette. Click on the color you wish to use and move the cursor to the Walker window. It will now change to the Paint cursor. Move it to where you want to apply the color. For our example, we will use the first wall. Click on the object and the color will be applied to it (see Fig. 25.7). Note that, like texture maps, the color is applied to only one face of the wall. If we want to color the entire wall, we must repeat this process for each face.

[Fig. 25.7](#)

Applying color with the Paint tool.

HSB also lets you "hang" a picture on a wall. To do so, click on the Picture icon and then click on the graphic file you wish to use. For example, if you wanted to put the VRML logo on one of the walls, you would simply click on that graphic. You then move the cursor to the Walker window where it changes to the Picture cursor. Select the area you want to "hang" the picture and click again. The graphic will then be applied to the object (see Fig. 25.8).

If you make a mistake while building your environment, you can always change your previous actions. Click on the "Undo last operation button" (the lower left button on the Builder panel). This will undo the last change to your environment before that, and so on. Once you have removed all the undesired steps,

you can continue to build your environment.

Finally, you can cut sections out of your walls to add doorways, archways, and windows by using the Cutting tool in the Builder toolbar. To begin, click on the Cutting tool (the icon button showing a pair of scissors). Then choose the height of your opening using the height tool. Note that you can adjust both the top level and the bottom level. In this way, you can use this tool for making windows. Since we want to add an archway, we will only adjust the top level.

[Fig. 25.8](#)

"Hanging" a picture using Home Space Builder.

Move the cursor to the Plane Builder window. Select where you wish to place the left edge of your archway and click the mouse. Move the cursor to where you want to place the right edge. Then click on the mouse button again. You will now see an opening in the Walker window (see Fig. 25.9).

[Fig. 25.9](#)

Adding an archway in Home Space Builder.

Finally, we can save our environment. Click on the Paragraph icon (the button with the • symbol above the Walker window) and choose Save. This will call up the Save dialog box (see Fig. 25.10). If you want to save your environment as a VRML file, select the Virtual Reality Modeling Language file option in the file type field. Type in the name you wish to call the file and click on OK.

[Fig. 25.10](#)

The Home Space Builder Save dialog box.

CAUTION

In essence, you are working in HSB's internal 3D file format while you are building your virtual environment. Therefore, if you want to stop and save your work before it is finished, you should save it using the MUS file format. If you save it using VRML's WRL file format, you will not be able to resume working on your environment after leaving the program.

HSB will save your environment using the VRML format. The following is a listing of the HSB output for the above environment. Note that while HSB saves the environment using the VRML format, it adds an end-of-file marker as a comment. You can remove this marker using any ASCII editor.

VRML V1.0 ascii

Separator{

ShapeHints {vertexOrdering CLOCKWISE shapeType SOLID}

LightModel {model BASE COLOR}

Coordinate3{

point[

-4096 0 7424, -4096 0 7168, -4096 896 7168, -4096 896 7424, _1280 718 7424,
0 718 7424, 0 896 7424, 1280 896 7424, 0 0 7424, 4096 0 7424, _1280 0 7424,
4096 896 7424, 4096 0 7168, 4096 896 7168, -402 0 7168, 0 0 _7168, 0 331 7168,
-402 331 7168, -402 740 7168, 0 740 7168, 0 896 7168, -402 _896 7168, -1024 0 7168,
-1024 896 7168, 7 718 7168, 1280 718 7168, 1280 740 7168, 7 _740 7168, 1280 896 7168,
1280 0 7168, -2304 440 7168, -1024 440 7168, -2304 896 7168, _-2304 0 7168,
-2304 0 3840, -2304 440 3840, -1024 0 3840, -1024 440 3840, 0 _718 7168,

] # 39 points

}

TextureCoordinateBinding {value PER_VERTEX_INDEXED}

TextureCoordinate2{

point[

0 0, 26 0, 26 3.4375, 0 3.4375,

]

}

MaterialBinding {value PER_FACE_INDEXED}

Material{

ambientColor 0 0 0

emissiveColor 0 0 0

diffuseColor[

0.784314 0.564706 0.580392, 0.737255 0 0.737255, 0.54902 _0.501961 0, 0 0 0.737255,

0 0.737255 0, 0.611765 0 0, 0.25098 0.501961 0.454902, _0.329412 0.533333 0.705882,

0.54902 0.690196 0.8, 0.658824 0.752941 0.580392, 0.376471 _0.486275 0.643137,

0.407843 0.439216 0.423529, 0.25098 0.439216 0.172549,

] # 13 colors

}

Separator{

IndexedFaceSet{

coordIndex[

0,1,2,3,-1, 4,5,6,7,-1, 8,0,3,6,-1, 9,10,7,11,-1, _12,9,11,13,-1, 14,15,16,17,-1,
18,19,20,21,-1, 22,14,21,23,-1, 24,25,26,27,-1, _19,26,28,20,-1, 29,12,13,28,-1,
30,31,23,32,-1, 1,33,32,2,-1, 2,13,11,3,-1, 29,10,9,12, _-1, 1,0,8,15,-1,
33,34,35,30,-1, 34,36,37,35,-1, 35,37,31,30,-1, _34,33,22,36,-1, 10,29,25,4,-1,
15,8,5,38,-1, 38,5,4,25,-1,

] # 23 faces

materialIndex[

0, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 5, 5, 6, 7, 8, _9, 9, 10, 11,

]

}

Separator{

Texture2{

filename "c:\progs\bl50.RGB"

model DECAL

}

IndexedFaceSet{

coordIndex[

36,22,31,37,-1,

] # 1 faces

textureCoordIndex[

0,1,2,3,-1,

]

materialIndex[

12,

]

}

}


```

Separator{
Texture2{
filename "c:\progs\64x64.RGB"
model DECAL
}
IndexedFaceSet{
coordIndex[
17,16,19,18,-1, 38,24,27,19,-1,
] # 2 faces
materialIndex[
3, 3,
]
}
}
}
}
}
}
}
#EOF

```

For More Information....

To find out more about Home Space Builder, see
<http://www.us.paragraph.com/whatsnew/homespce.htm>.

Virtus WalkThrough (Virtus Corporation)

Virtus Corporation was among the first companies to offer commercial VR authoring tools for the PC and Macintosh platforms. The company announced that it would incorporate VRML into its Virtus WalkThrough Pro and Virtus VR products by the end of 1995. It also released its Virtus WalkThrough Special Edition/VRML software with Mark Pesce's VRML: Browsing and Building Cyberspace.

All Virtus VR authoring software for both the PC and Macintosh work in much the same way. The interface provides you with a Walk View window in which you can see and navigate your environment and a variety of working windows in which you add and position your objects (see Fig. 25.11). You can select different views for the working window: Top, Bottom, Left, Right, Front, and Back. The default setting is the Top View.

[Fig. 25.11](#)

The Virtus WalkThrough Special Edition/VRML interface.

To navigate in the Walk View window, click on the window to activate it. You will see the Walk toolbox appear on the left side of the screen. To move forward in the environment, click and hold the mouse button and then move the cursor above the crosshair in the middle of the window. To move to the right, click and hold the mouse button and move to the right of the cross-hair. Moving backwards and to the left, follow the same procedure. You can also use the mouse in conjunction with the Shift key to change your pitch (the up/down viewing angle) and use the mouse with the Control key to roll your view. If you lose track and become disoriented, select the View menu and choose Level Observer.

You will also notice a small circle in the working window. It indicates your point of view in the Walk View window. While you are building your environment, you can move the circle to the area where you are working by moving the cursor to it, clicking and holding the mouse button, and dragging the cursor to the desired location.

Creating an Environment Using Virtus WalkThrough

To start building your 3D environment in Virtus WalkThrough, click on the working window. The Design toolbox should appear on the left side of your screen. To build a rectangular or square wall, click on the square tool. Click and hold the mouse on a point in the working window and drag it until you get the size of wall you desire. Let go of the mouse button and your wall will appear (see Fig. 25.12).

[Fig. 25.12](#)

Creating a wall in Virtus WalkThrough.

We can then attach another wall to the first by following the same procedure. This time, however, we will align it perpendicular to the first wall (see Fig. 25.13).

[Fig. 25.13](#)

Adding another wall to our environment.

NOTE

Unlike Home Space Builder, Virtus WalkThrough produces two-sided polygons. Therefore, if you want to create a room using any of the shape tools, you can do so simply by making the shape the size of the room that you require. This feature is extremely useful if you want to create a round or irregularly shaped room.

We can also add a pillar to our environment. Click on the circle tool. Then move the cursor to where you want to place the pillar. Click and hold the mouse and drag it until you get the size you want. Then let go of the mouse button and your pillar will appear in the Walk View window (see Fig. 25.14).

Now that we have the basic elements of our environment, you may notice that the height of our walls and the pillar seems to be a bit short. Fortunately, Virtus WalkThrough lets you adjust the size of your objects even after you have created them. Select the View menu item. You will see the different views available for the working window. Choose Front. The Front View window will appear over the Walk View (see Fig. 25.15).

[Fig. 25.14](#)

Adding a pillar.

[Fig. 25.15](#)

Activating the Front View window.

Now click on the select tool in the Design toolbox: it's the one with an arrow. Move the cursor to the Front View window and click on the pillar. A bounding box will surround the pillar. Move the cursor to the top middle square, and then click and drag the box up. The pillar will become taller. When you have set the pillar to the desired height, let go of the mouse button. You can do the same with the rear wall (see Fig. 25.16).

[Fig. 25.16](#)

Increasing the height of the pillar and rear wall.

The Design tools for Virtus WalkThrough are easy to use and have a great deal of flexibility. As well as creating predefined shapes such as cubes and columns (using the circle and square tools respectively), Virtus also lets you create irregular-shaped rooms using the polygon tool.

Virtus WalkThrough's ability to edit your walls and objects after you have created them proved to be an invaluable asset. There were a number of times when I used Home Space Builder when I made a mistake and had to remove an entire wall and start all over again. Virtus allowed me to make adjustments after I created a wall. When you are trying to design your environment using a worldbuilding toolkit's interface, having the ability to edit your space after you have built it is extremely usefulóif not absolutely vital.

Importing Objects into Your Environment Using Virtus

One of the best features of Virtus WalkThrough is its ability to import objects and place them into your environment. Virtus provides you with a library of objects. To access them, select the File menu and choose Library. You can choose from the standard library or any of the bonus libraries. Choose Bonus 1. The Object Library window will appear (see Fig. 25.17).

[Fig. 25.17](#)

The Object Library window.

You can look through the library by clicking on any item in the list. Let's choose the console table. In order to place this object into your virtual environment, select Edit Copy from the main menu. You can

then close the Object Library window. Then go to the main menu and select Edit Paste. The console table will appear in both the working window and the Walk View window (see Fig. 25.18). You can now use the select tool to move the table to the position you desire.

Fig. 25.18

Adding the console table to your Virtual environment.

You can repeat this process as many times as you want to add and position more items in your environment. However, each item that you import will be stored as a separate set of coordinates and polygons. Moreover, copying the same object will result in a new set of data. Therefore, the more objects you include, the larger will be your file.

For More Information....

To find out more about Virtus WalkThrough, Virtus VR, and Virtus Walkthrough Pro, see <http://www.virtus.com/>.

WebSpace Author (Silicon Graphics, Inc.)

Silicon Graphics, Inc. (SGI) has developed its WebSpace Author software as a part of its WebFORCE suite of Internet tools. It is a sophisticated 3D authoring tool that allows you to create VRML environments. You can create your own 3D models using WebSpace Author, or you can import 3D models from other 3D software packages and incorporate them into your VRML environment.

WebSpace Author is different from other 3D modeler and worldbuilding toolkits in that it was designed for creating VRML environments from the start. It lets you import existing models into a scene or as Inline objects. It also includes a Shape Modeler, a 3D Text Modeler, and an LOD Editor. You can create links from any object in the scene to any URL, file, or viewpoint and save specific viewpoints to create "guided tours" of your environment.

WebSpace Author also provides you with a variety of viewing options. You can use perspective viewing modes with the program's Walk Viewer and Examiner Viewer. You may also use orthographic viewing modes to check specific objects. Moreover, the program will display the locations of WWWAnchor, WWWInline, and LOD nodes as well as showing Cull Volumes.

The program supports a variety of 3D file formats. It can directly import files saved with Open Inventor and VRML. It can also translate Alias, Wavefront, Softimage, 3D Studio, IGES, and DXF formats. It will save VRML files in ASCII or compress them using g-zip.

For more information about WebSpace Author, see <http://webpace.sgi.com/WebSpaceAuthor/index.html>.

From Here...

Using a worldbuilding toolkit can help you build more complex VRML environments. But once you have your environment, you will want to optimize it to minimize the size of the .wrl file.

To learn about creating your own VRML environments or to learn about how to use VRML so that you can optimize the output of your worldbuilding toolkit, see the following:

- If you are ready to start programming in VRML, jump to Part IV, ["Creating a VRML Environment from Scratch."](#) These chapters provide a step- by-step description of how to create VRML environments as well as the standards and practices for VRML programming.
- If you want to look at the experiences of others who have created their own VRML environments and models, see Part V, "Learning from the Pros: Sample VRML Environments."
- If you want to look at some VRML environments and models, see [Chapter 23, "A Guide to Other Sample Environments on the Web."](#) It lists some of the VRML sites that are currently available. Programs are available along with their URLs.
- If you want to see what programs are available to help you create VRML environments, see [Chapter 24, "An Overview of 3D Modelers."](#) This chapter discusses how these programs can help you build VRML objects.
- If you want to see what utilities are available to help you convert non-VRML files to VRML, see [Chapter 26, "VRML Utilities."](#)
- If you feel the need to create your own VRML utilities, see [Chapter 32](#). That chapter will give you the basic knowledge that you will need to create VRML utilities from scratch.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



26-VRML Utilities

One of the nice things about a standard like VRML is that it allows independent developers, researchers and hobbyists to create a variety of useful support tools. Even though VRML is a relatively new standard, there are already a number of translators, generators and miscellaneous utilities.

In this chapter, we will

- Learn how to use several different file format translators
- Discover how to use fractals to generate terrain
- Explore L-systems, and see how they can be used to generate organic shapes
- Take a look at a VRML authenticator

Most of the tools discussed in this chapter are available on the companion CD-ROM.

The Tower of Babel Meets the Translators

The reason we need a standard like VRML in the first place is because each modeling program, VR system, and object generator has its own way of representing data. It's a Tower of Babel, with everyone speaking a different language.

With the advent of VRML, we can see a certain amount of convergence starting to happen. However, there are still a lot of files out there in a wide variety of different formats. Even when each of the packages has VRML export capability, it would take a long time for everyone to get around to loading their objects and worlds into their packages and saving them out as VRML. What's needed is a translator.

A translator reads files in a number of different formats and writes them out in a variety of others. In our case, we're interested specifically in translators that deal with 3D file formats (like VRML).

The most popular and powerful of the conversion programs is InterChange, from Syndesis Corporation (<http://www.webmaster.com/syndesis/>). It reads and writes over 30 different formats, a remarkable accomplishment. It also does a good job of translating all the various formats into VRML. Unfortunately, this power comes at a price—InterChange costs over \$500!

Fortunately, there's a freeware alternative. A programmer named Keith Rule has created WCVT2POV, a Windows-based program that supports eight of the most popular formats and can convert them to (among other things) VRML. Incidentally, it can also convert to a format called POV, used by the Persistence Of Vision raytracing program. You can find WCVT2POV on the CD-ROM that accompanies this book.

Using WCVT2POV

When you first fire up WCVT2POV, you get the display shown in Figure 26.1.

[Fig. 26.1](#)

WCVT2POV before loading any data.

All the tasks can be performed using either the buttons or the menus. The first step in converting a file to VRML is to load it into WCVT2POV. After clicking File/Open, the display looks like Figure 26.2.

[Fig. 26.2](#)

Loading a file in WCVT2POV.

The menu shows a list of all the different formats that WCVT2POV version 2.6 knows how to read. Here's a quick synopsis of the most important ones:

AOFF

AOFF is the ASCII version of the OFF format. OFF stands for Object File Format; the reason ASCII is mentioned in the name is because there's also a binary version. OFF is unusual in that it sometimes has a number of separate files to represent a single object. One of those files, the GEO (for Geometry) file is the one that contains the information we're interested in.

DXF

DXF is the Drawing eXchange Format, originally designed as a format for exchanging AutoCAD drawings. It's an ugly format, but WCVT2POV does a reasonable job of converting it.

3DS

3DS is a format used by another Autodesk product, 3D Studio. It's a much better format than DXF, and it stores lots of useful information about materials and lighting.

NFF

NFF is a confusing name; there are actually two completely different formats called NFF, and in both cases the name stands for Neutral File Format! The one supported by WCVT2POV is the one designed by Eric Haines, not the one created by Sense8 for its WorldToolKit package.

RAW

The RAW file format is the simplest one of all; it's basically just a list of triangles, each stored on a separate line of the file, with three vertices for each triangle. Each vertex is just a set of X, Y, and Z values. Each triangle may also have color information associated with it. WCVT2POV is smart enough to merge identical vertices, which drastically reduces the size of the resulting VRML file and improves rendering performance as well (I described performance improvement back in Chapter 17).

OBJ

OBJ is the file format used by Wavefront's modeling package, which is popular on many UNIX workstations.

Others

There are a lot of other formats out there, but the ones above are certainly the most popular. Notably absent from WCVT2POV's list are Lightwave (used for professional 3D modeling on the Video Toaster and other systems) and Sense8's NFF. If you need to read Lightwave files, you can use Caligari's Fountain; if you need to read Sense8 NFF files, you can try the NFF2VRML converter described later in this chapter.

Loading the Object

Let's read in an OFF file of a teapot. We select the appropriate menu item, and choose the file Teapot.geo from the dialog box as shown in Figure 26.3.

Our reward for loading the file is a wireframe display of it, as shown in Figure 26.4.

[Fig. 26.3](#)

Choosing an input file in WCVT2POV.

[Fig. 26.4](#)

The object after it's been loaded.

At this point, we have a variety of options open to us. Let's start by finding out some basic information about our object; clicking on the Information icon brings up a box with all the relevant details in it. As you can see in Figure 26.5, this particular object contains close to 2,000 vertices and over 3,700 polygons.

[Fig. 26.5](#)

Getting information about the object.

We can use the plus and minus icons to zoom in and out, as shown in Figure 26.6.

[Fig. 26.6](#)

Zooming in on the teapot.

We can also flip the object over any of the X, Y, or Z axes; for example, Figure 26.7 shows the teapot after being flipped over both the X and Y axes.

Fig. 26.7

Flipping the object.

In addition to flipping, the object can also be rotated around the axes by any amount.

It's also possible to paint an object (or individual "groups" within an object, if the object was divided into separate groups when the file was first created). Figure 26.8 shows the Color and Texture Info dialog box.

Fig. 26.8

Adding some color.

The color, texture, and smooth-shading attributes of each "group" can be specified, and individual groups can also be renamed.

The final step is to save the object. Figure 26.9 shows the selection of file formats available to us; in our case, we're only interested in VRML.

Fig. 26.9

Saving the file.

Loading the teapot into a VRML browser gives us the result we want, shown in Figure 26.10.

Fig. 26.10

The VRML Teapot.

WCVT2POV is a very, very useful utility for converting existing files from several different formats into VRML.

WLD2VRML

One of the more popular formats used by VR hobbyists is called WLD, which simply stands for "world." It's much simpler than VRML, but serves a similar purpose: to describe the location, materials, and hierarchical attachment of lights and objects in a virtual world. WLD files reference PLG files, which store polygonal geometry information, and FIG files, which describe hierarchical figures.

When I designed these formats a number of years ago, there weren't any standards for describing virtual worlds. If I were doing it today, I'd simply adopt one of the existing formats (such as VRML!). Still, the WLD/PLG/FIG family of standards has served its purpose. Not only was it used in REND386, AVRIL and VR-386, but it's also been adopted by Jon Blossom's Gossamer for the Macintosh.

Now that VRML is becoming the standard for virtual world description, I've written a converter that

reads WLD files (and the PLG files and FIG files that the WLDs reference) and converts them to VRML.

The program runs under DOS, and is not the least bit interactive. It takes one command-line parameter, the name of the WLD file to load. It writes VRML to its standard output. To convert a file called "sample.wld" to VRML, use the following command:

```
wld2vrml sample.wld >sample.wrl
```

It's that simple. The conversion isn't perfect, especially when it comes to materials and lighting, but it's a good first step. The latest version of the program can be found online (the URL is <ftp://sunee.uwaterloo.ca/pub/rend386/converters/wld2vrml.zip>).

NFF2VRML

One of the first commercial VR libraries for C programmers was Sense8's WorldToolKit. It's a little pricey, but it works well and can be used to create powerful VR applications. No doubt Sense8 will soon have support for VRML import and export.

In the meantime, it would be useful to have a converter that reads Sense8 NFF files (not to be confused with the NFF format created by Eric Haines, which is supported by WCVT2POV) and writes them out as VRML. Such a converter is also handy for people who can't afford WorldToolKit.

I've written an NFF to VRML converter, called NFF2VRML. It's a quick hack, and doesn't support textures or 24-bit color specifiers (only 12-bit). It also doesn't handle the normals and shading quite right. On the plus side, I've included the complete source code; if you're a C programmer, you can make improvements yourself. The source is included on the CD-ROM that comes with this book; if you do make improvements, please drop me some e-mail to let me know (broehl@sunee.uwaterloo.ca).

Like WLD2VRML, NFF2VRML is a DOS program which reads an NFF file from its standard input, and writes VRML to its standard output. It takes an optional "scale" parameter on the command line; this is used to scale the X, Y, and Z coordinates, so you can make the object larger or smaller. For example, to convert a file called "test.nff" into VRML, scaling it to be two and a half times larger, you would use the following command:

```
nff2vrml 2.5 <test.nff >test.wrl
```

The internal workings of NFF2VRML are discussed in Chapter 32.

Creating Something from Nothing (or from very little...)

Translators like WCVT2POV, WLD2VRML, and NFF2VRML are useful if you already have objects or worlds in some existing format. What if you don't? Are you forced to do everything by hand, vertex by vertex?

Well, no. As Stephen describes in other chapters, you can use modeling programs (such as Caligari's "Fountain"). You can also use "object generators," which are powerful tools for creating all kinds of

different shapes. We'll take a look at two of them: FRGEN and LPARSER.

FRGEN

Fractals are mathematical constructs that can describe a variety of shapes found in nature. Fractal patterns can be found all around us, at both the macroscopic and the microscopic level.

FRGEN is a freeware program written by Steve Anger. It uses fractal techniques to create a variety of terrain, from gently sloping desert plains to rugged mountains. It can produce output in a variety of formats, but the one we're going to use is RAW. Once the RAW file is produced, we can use WCVT2POV to translate it into VRML.

FRGEN is a DOS program that reads an input file and writes an output file. The command line format is

FRGEN infile outfile options

For example, to generate a mountain range, you would use the following command:

frgen mountain.fr mountain.raw -or

The "-or" option means "output raw." The resulting file (mountain.raw) can be read into WCVT2POV and saved out as VRML. The result is shown in Figure 26.11.

Fig. 26.11

Fractal mountains.

The ".fr" and ".raw" extensions on the filenames given to FRGEN are optional; ".fr" is the default, and ".raw" is the default if "-or" is specified.

FRGEN has a variety of options, but before we can discuss them we need to look at how the program itself works.

Getting into Fractals

FRGEN is based on subdividing triangles. The input file describes one or more "base" triangles; each of these is subdivided into four smaller triangles.

Random noise is used to displace the vertices in three dimensions. The subdivision routine then calls itself again (a process called "recursion") to subdivide each of the new triangles and displace their vertices.

The Art of Noise

The displacement is done with "noise vectors." There are three sets of options that control how the noise vectors are modified. The first are the scale factors; a scale factor of 0.0 means that the noise vector should not be used at all. Good values are in the range 0.05 to 0.25; you can either use "-s" to set all the X, Y, Z scale factors to the same value, or use separate "-sx," "-sy," and "-sz" parameters. For example,

`frgen mountain.fr mountain.raw -or -sx0.2 -sz0.15`

would scale only the X and Z components of the noise vectors. The default X, Y and Z scaling factors are 0.1, which is a good value.

The noise vectors can also be biased in a particular direction; the "-bx," "-by," and "-bz" options are used to specify the bias vector. The default bias vector is (0 0 0). All three values (bx, by, and bz) can be set at once with the "-b" option.

The noise vectors can be made less significant at each level of triangle subdivision. To do this, the "falloff" factors are set using the "-fx," "-fy," and "-fz" options; they specify the amount by which the scaling of the noise should itself be rescaled at each level of subdivision. The default values are all 1.0, meaning that the scaling of the noise factor remains the same at each level of subdivision. The use of falloff produces a "weathered" effect, making for a more naturalistic look.

Previewing the Results

The -d option causes FRGEN to produce an on-screen preview of the generated fractal. This is a great time-saver; without it, you'd have to run FRGEN, run WCVT2POV, and load the result into a VRML browser, each time you made a change.

Figure 26.13 shows the display preview from FRGEN.

[Fig. 26.13](#)

FRGEN's on-screen preview.

By default, FRGEN produces an output file as well as the preview; if you only want the preview, not the output, add the "-on" option (for "output none") in place of "-or."

Controlling Recursion and Randomness

There are also options to control the randomness and recursion. The "-r" option specifies the maximum recursion depth; the default is 3. Remember that each triangle gets subdivided into four smaller triangles; therefore, the number of triangles increases as four to the power of the recursion factor (so -r5 will produce 4^5 or 1,024 triangles).

The "-e" option can be used to specify the seed value for the random number generator. By using the same seed, you can generate an identical fractal, perhaps with a different level of recursion. The resulting objects can be stored as children of an LOD node, for example.

To see how the combination of -e and -r can be used to control level of detail, consider the following three examples:

[Fig. 26.14](#)

Output from "frgen mountain.fr mountain.raw -or -e12 -r2."

[Fig. 26.15](#)

Output from "frgen mountain.fr mountain.raw -or -e12 -r3."

Fig. 26.16

Output from "frgen mountain.fr mountain.raw -or -d -e12 -r4."

Each of these could be separately processed by WCVT2POV, and referenced by an LOD node; as you move closer to the mountain range, more and more detail will appear.

Defining the Base Triangles

The input file format is very simple. Any line beginning with a semicolon (";") is a comment. The first non-comment line is a list of command-line options (so you don't have to keep typing them on the command line, and so you can keep them for future reference when you find a combination that works well). The rest of the line is a list of triangles, with three vertices on each line. Each vertex is a set of X, Y, and Z values; there are, therefore, 9 values on each line.

You can also specify a set of three flags at the end of the line, each of which is either 1 or 0. If the first value is 1, then the edge between the first two vertices of the triangle is kept in the same plane as the original triangle; if the second value is 1, then the edge between the second two vertices is kept in the same plane, and if the third value is 1 then the edge between the first and third vertices is kept in the same plane. The most common use of these values is to keep the edges of a mountain range in the same horizontal plane.

The complete format of each line is therefore

ax ay az bx by bz cx cy cz [fix_ab fix_bc fix_ca]

The square brackets should not be typed; they just indicate that the final three values are optional. If not specified, these "fix" values all default to 0 (meaning the edges are not locked down).

Some Interesting Terrain

The best way to see how FRGEN works is by examining the examples that are included with it. Here's a quick look at five different miniworlds created with FRGEN.

The Ice Cube

The first example starts off as a box, but after processing through FRGEN it begins to melt a bit.

Listing 26.1

; A box. With a transparent surface and lots of smoothing

; it makes a nice ice cube

-vx20 -vy20 -vz-30 -r4

;

-10 -10 10 10 -10 10 10 10

```

-10 -10 10 10 10 10 -10 10 10
-10 -10 10 -10 -10 -10 10 -10 -10
-10 -10 10 10 -10 -10 10 -10 10
10 -10 10 10 -10 -10 10 10 -10
10 -10 10 10 10 -10 10 10 10
10 10 10 10 10 -10 -10 10 -10
10 10 10 -10 10 -10 -10 10 10
-10 10 10 -10 10 -10 -10 -10 -10
-10 10 10 -10 -10 -10 -10 -10 10
-10 -10 -10 -10 10 -10 10 10 -10
-10 -10 -10 10 10 -10 10 -10 -10

```

The resulting object (after translation via WCVT2POV and viewing in a VRML browser) looks like Figure 26.17.

A Diamond in the Rough

The next example starts off as an eight-sided diamond, then gets "roughed up" a little in FRGEN.

Listing 26.2

; An eight faced diamond shape

```

-vx0 -vy5 -vz-30 -r3
0 0 +10 +10 0 0 0 +10 0
+10 0 0 0 0 -10 0 +10 0
0 0 +10 -10 0 0 0 +10 0
-10 0 0 0 0 -10 0 +10 0
0 0 +10 +10 0 0 0 -10 0
+10 0 0 0 0 -10 0 -10 0
0 0 +10 -10 0 0 0 -10 0
-10 0 0 0 0 -10 0 -10 0

```

Fig. 26.17

I'm melting!

Figure 26.18 shows our sparkling beauty.

[Fig. 26.18](#)

A girl's best friend.

The Hills Are Alive...

A gently rolling set of hills is our third example.

Listing 26.3

; Some rolling hills or eroded mountains. Shows how the falloff

; parameter (-f) affects terrain

-by0.55 -sx0.06 -sy0.2 -sz0.06 -f0.8 -vx0 -vy8 -vz-20 -r5

;

+10 0 0 0 0 +10 -10 0 0 1 1 0

-10 0 0 0 0 -10 +10 0 0 1 1 0

Notice that only two base triangles are required to produce the realistic effect seen in Figure 26.19.

[Fig. 26.19](#)

Over hill and dale.

Plain and Simple

The difference between those rolling hills and our next example, a mostly horizontal plane, is the lack of falloff or bias and a smaller "sy" value for lower hills.

Listing 26.4

; A horizontal rolling terrain

-sx0.06 -sy0.1 -sz0.06 -vx0 -vy8 -vz-20 -r5

;

+10 0 0 0 0 +10 -10 0 0 0

-10 0 0 0 0 -10 +10 0 0 0

This generates the pastoral scene in Figure 26.20.

[Fig. 26.20](#)

Plain as can be.

Kicking Asteroid

Just to show that not all mountains are earthbound, our next example creates the sort of "mountain" you'd find orbiting the Sun, out between Jupiter and Mars....

Listing 26.5

; Sphere made up of 48 triangles

-vx0 -vy10 -vz-30 -r2

;

0.00 8.80 0.00 6.39 6.38 0.06 4.48 6.38 4.56

6.39 6.38 0.06 8.99 0.00 0.08 6.30 0.00 6.42

6.39 6.38 0.06 6.30 0.00 6.42 4.48 6.38 4.56

8.99 0.00 0.08 6.39 -6.38 0.06 4.48 -6.38 4.56

8.99 0.00 0.08 4.48 -6.38 4.56 6.30 0.00 6.42

6.39 -6.38 0.06 0.00 -8.80 0.00 4.48 -6.38 4.56

0.00 8.80 0.00 4.48 6.38 4.56 -0.05 6.38 6.39

4.48 6.38 4.56 6.30 0.00 6.42 -0.08 0.00 8.99

4.48 6.38 4.56 -0.08 0.00 8.99 -0.05 6.38 6.39

6.30 0.00 6.42 4.48 -6.38 4.56 -0.05 -6.38 6.39

6.30 0.00 6.42 -0.05 -6.38 6.39 -0.08 0.00 8.99

4.48 -6.38 4.56 0.00 -8.80 0.00 -0.05 -6.38 6.39

0.00 8.80 0.00 -0.05 6.38 6.39 -4.56 6.38 4.48

-0.05 6.38 6.39 -0.08 0.00 8.99 -6.41 0.00 6.30

-0.05 6.38 6.39 -6.41 0.00 6.30 -4.56 6.38 4.48

-0.08 0.00 8.99 -0.05 -6.38 6.39 -4.56 -6.38 4.48

-0.08 0.00 8.99 -4.56 -6.38 4.48 -6.41 0.00 6.30

-0.05 -6.38 6.39 0.00 -8.80 0.00 -4.56 -6.38 4.48

0.00 8.80 0.00 -4.56 6.38 4.48 -6.39 6.38 -0.05

-4.56 6.38 4.48 -6.41 0.00 6.30 -8.99 0.00 -0.07

-4.56 6.38 4.48 -8.99 0.00 -0.07 -6.39 6.38 -0.05

-6.41 0.00 6.30 -4.56 -6.38 4.48 -6.39 -6.38 -0.05

-6.41 0.00 6.30 -6.39 -6.38 -0.05 -8.99 0.00 -0.07

-4.56 -6.38 4.48 0.00 -8.80 0.00 -6.39 -6.38 -0.05
0.00 8.80 0.00 -6.39 6.38 -0.05 -4.48 6.38 -4.56
-6.39 6.38 -0.05 -8.99 0.00 -0.07 -6.31 0.00 -6.41
-6.39 6.38 -0.05 -6.31 0.00 -6.41 -4.48 6.38 -4.56
-8.99 0.00 -0.07 -6.39 -6.38 -0.05 -4.48 -6.38 -4.56
-8.99 0.00 -0.07 -4.48 -6.38 -4.56 -6.31 0.00 -6.41
-6.39 -6.38 -0.05 0.00 -8.80 0.00 -4.48 -6.38 -4.56
0.00 8.80 0.00 -4.48 6.38 -4.56 0.04 6.38 -6.39
-4.48 6.38 -4.56 -6.31 0.00 -6.41 0.06 0.00 -8.99
-4.48 6.38 -4.56 0.06 0.00 -8.99 0.04 6.38 -6.39
-6.31 0.00 -6.41 -4.48 -6.38 -4.56 0.04 -6.38 -6.39
-6.31 0.00 -6.41 0.04 -6.38 -6.39 0.06 0.00 -8.99
-4.48 -6.38 -4.56 0.00 -8.80 0.00 0.04 -6.38 -6.39
0.00 8.80 0.00 0.04 6.38 -6.39 4.55 6.38 -4.48
0.04 6.38 -6.39 0.06 0.00 -8.99 6.40 0.00 -6.31
0.04 6.38 -6.39 6.40 0.00 -6.31 4.55 6.38 -4.48
0.06 0.00 -8.99 0.04 -6.38 -6.39 4.55 -6.38 -4.48
0.06 0.00 -8.99 4.55 -6.38 -4.48 6.40 0.00 -6.31
0.04 -6.38 -6.39 0.00 -8.80 0.00 4.55 -6.38 -4.48
0.00 8.80 0.00 4.55 6.38 -4.48 6.39 6.38 0.06
4.55 6.38 -4.48 6.40 0.00 -6.31 8.99 0.00 0.08
4.55 6.38 -4.48 8.99 0.00 0.08 6.39 6.38 0.06
6.40 0.00 -6.31 4.55 -6.38 -4.48 6.39 -6.38 0.06
6.40 0.00 -6.31 6.39 -6.38 0.06 8.99 0.00 0.08
4.55 -6.38 -4.48 0.00 -8.80 0.00 6.39 -6.38 0.06

Our extraterrestrial guest is shown in Figure 26.21.

As you can see, fractals are a great way of generating the kinds of shapes nature produces. However, it's not the only way....

[Fig. 26.21](#)

Look out!

Lindenmayer Systems

Lindenmayer systems, often called "L-systems," were first developed by a biologist named Aristid Lindenmayer back in the late '60s as a way of describing plants. Almost twenty years later, a computer scientist named Alvy Ray Smith realized that you could write computer programs to create virtual plants based on L-systems.

The basic idea of a Lindenmayer system is to take a starting string of characters called the "axiom" and transform it into another string by using a set of rules called "productions." The resulting string can then be transformed again by the same set of rules, producing yet another string. This process can go on indefinitely, just as fractals can be explored to any depth.

So you can turn one string of characters into another string of characters. So what? Well, the interesting thing is that you can treat that string of characters as a kind of "genetic code" that describes how to make something like a plant or an animal. The application of the productions to this genetic axiom can produce some beautiful and bizarre objects that look remarkably organic.

Expressing the Genes

In order to turn a string of characters into an object, we need to treat the characters in the string (sometimes called "symbols") as a very simple programming language. Imagine that you have a remote-controlled airplane with a smoke generator. You can send it some simple instructions: turn left, turn right, tilt left, tilt right, roll, fly forward, and so on. As it travels, it can leave a smoke trail; you can turn the smoke generator on and off by remote control as well. By flying the plane around, you can create some pretty complex figures from the smoke trails.

The string of characters from an L-system can be treated as a set of "flying instructions" for this imaginary airplane. The actual assignment of the instructions to the characters of the string is arbitrary; table 26.1 shows the particular instructions used by the L-system software we'll be examining in this chapter.

Table 26.1

Character	Assignment
+	turn left around current "up" vector
-	turn right around current "up" vector
^	tilt up around current "left" vector
&	tilt down around current "left" vector
<	roll counterclockwise around "forward" vector
>	roll clockwise around "forward" vector
	turn around 180 degrees
%	roll around 180 degrees

\$	roll until horizontal
~	turn/pitch/roll in a random direction
t	correction for gravity by 0.2
F	move forward, leaving smoke trail, record polygon vertex
Z	move forward, leaving smoke trail, record polygon vertex
f	move forward, record polygon vertex
z	move forward half-way, record polygon vertex
g	move forward
.	record polygon vertex (don't move)
[	push color, thickness, length, and angle onto the stack
]	pop color, thickness, length, and angle from the stack
{	start polygon
}	end polygon
"	multiply length by 1.1
'	multiply length by 0.9
;	multiply angle by 1.1
:	multiply angle by 0.9
?	multiply thickness by 1.4
!	multiply thickness by 0.7
c	increase color index value
*	reset color, thickness, length, and angle
@	end of object

For example, the string F+F+F+F would draw a square out of the smoke trail of the plane; move forward and draw, turn left, move forward and draw, turn left, and so on.

Some instruction can take a parameter; for example, +(5) would mean "turn left 5 degrees" instead of using the current turning angle.

The "stack" referred to in the "[" and "]" instructions is a way of keeping track of the previous state. When you push something onto a stack, the information is saved away; when you pop it, the information is brought back. If you push several sets of information, the one you pushed last is brought back first; it works like a stack of trays in a cafeteria.

The "recording" of vertex information is used with the "{" and "}" instructions. If you execute the "{" instruction, it begins to draw a new polygon; any instruction which records a vertex just keeps track of where that vertex was (in essence, it puts an entry into a Coordinate3 node). When a "}" instruction is encountered, it draws a polygon using all the recorded vertices (a little bit like an IndexedFaceSet) using the current color.

Going into Production

The production rules each have a left-hand side and a right-hand side. The left-hand side is a single character; if that character appears in the axiom, then it's replaced by the series of characters on the right-hand side. For example, if you have the following productions:

A=ABAB=BA

and the axiom is AB, then after one pass the string will be ABABA. That string can then be fed through the productions again, to get ABABAABABAABA, and so on.

LPARSER

LPARSER is a freeware L-system software package created by Laurens J. Lapre. It takes a filename on the command line; the grammar (axiom and production rules) are read from that file. There are a number of options, but the only one you have to worry about is "-d," which causes the output to be written in AutoCAD DXF format.

The output file is called "output.dxf," and it can be processed through WCVT2POV to generate VRML.

The input file format is straightforward. Anything after a # character on any line is treated as a comment, and ignored. The first three non-comment lines contain numbers that give the recursion depth to use, the initial turning angle, and the initial length (the distance to move forward). This is followed by the axiom, on a line by itself, and a series of lines that contain production rules.

For example, consider the file shown in Listing 26.6.

Listing 26.6

```
# ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóóóó
```

18 # recursion depth

10 # angle

10 # thickness as % of length

FFFFFFFFc[A][B]

$$A = [+A\{.\}.C.]$$
$$\underline{B = [-B\{.\}.C.]}$$
$$\underline{C = gC}$$

@

The axiom is "FFFFFFc[A][B]." The first production rule replaces the A with "[+A{.}.C.]," and the second replaces B with "[-B{.}.C.]" After one iteration, the axiom becomes "FFFFFFc[[+A{.}.C.]] [[-B{.}.C.]]." Starting with the next generation, the "C=gC" rule becomes

The Virtual Gardener

Of course, leaves are nice but we want more. How about complete plants? No problem:

Listing 26.8

```
#  L-System Parser/Mutator  Lj Lapre
5 # recursion depth
18 # angle
50 # thickness as % of length

p
p=i++[p+o]o>>[ol]i[++]o[po]+++po
i=Fs[>>&&l][>>^^]Fs
s=sFs
l=[c{++föffÅ97ÿf++|++föffóf}]
o=[&!ce>w>>>>w>>>>w>>>>w>>>>w]
e=FF
w=[c^!F][cc&&&&{óf+++f|óf+++f}]

@
```

Fig. 26.24

A plant.

Listing 26.9

```
#  plant01.ls  C.J. van der Mark jr.
10
20
20
q
q=[p][>(90)-(90)g(50).+(90)c(4)?(1)[x]]>(90)-(90)g(100).+(90)[p]
p=sp
s={ab[+Zh]<(90)ab[-Zh]}
a=Z+(5)a
```

b=Z-(5)b

d=Z{[+++e.][++ee.][+eee.][eeee.][-eee.][óee.][óe.]}

h=cZZZt(3.2)\$[++[++ZZd]ZZd][ó[óZZd]ZZd]ZZd

x=Z(60)!(0.90)x

e=t!g

@

Simple plants like these are all fine and good, but what about something more ambitious? Well, we can make a tree, as shown in Listing 26.10.

[Fig. 26.25](#)

Another plant.

Listing 26.10

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóó

#

Monopodial skeleton

#

10

45

15

#

FFAL

#

A=F[&'(.8)!BL]>(137)'! (.9)A

B=F[-'(.8)! (.9)\$CL]'! (.9)C

C=F[+'(.8)! (.9)\$BL]'! (.9)B

#

L=~cc{+(30)f(2)-(120)f(2)-(120)f(2)}

@

As you can see, it's a lot more complex than our previous examples. However, the object it produces is also a lot more complex, as shown in Figure 26.26:

Fig. 26.26

The Larch.

Springtime

Ah yes, the time when flowers bloom. Here are two flowers, and the L-systems which grew them:

Listing 26.11

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóó

20

7

50

#

&(10)c(4)Dc(6)[p>(72)p>(72)p>(72)p>(72)p]c(7)E # 5 petal elements on a branch

#

p=[&(90)[A]+(72)[B]] # one petal

#

A=[^gA{.}. # first 'curved line'

B=B^g.g.} # second 'curved line'

#

D=t(.1)Ft(.1)FD # branch

E=FE # stem

@

Fig. 26.27

A rose by any other name....

Listing 26.12

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóóóó

10

10

80

FS

S=WZ'!cS

W=R%R>>>>>>>>R%R>>>>

R=[da%a%g%g]

a=[{nmmn}]

g=[NMMNL]

m=+zm

n=-zn

M=+ZM

N=-ZoN

d=^d

L=Z';+Z';+Z';+Z';+ [&&&l]L

l=[c{ó z++z++z|ó z++z++z}]

o=[óóóó!!!"cZl]

@

Fig. 26.28

Virtual nature in all its splendor.

Enough of Nature

If you're like me, you can only take so much natural beauty. VR is high-tech, right? Let's make an industrial shape for a change.

Listing 26.13

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóó

2 # recursion depth

90 # angle

20 # thickness as % of length

X+cX+cX+cX

X=^F+F+F+F+&f

F=F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF

f=ffff

@

Fig. 26.29

Cold steel.

By the Sea

L-systems can not only create plants and industrial shapes, they can also make sea shells.

Listing 26.14

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóóóó

19

10

10

#

PPPPPPPPPPPPPPPPPPPPPPPPPP

#

P=S>(20)gg'

S=[&(90)[A]+(20)^(90)gg'&(90)[B]]

#

A=[^gA{.].

B=B^.g.}

@

Fig. 26.30

You can almost hear the ocean.

Animal Kingdom

We've already seen how to grow plants using L-systems; now it's time to move up the evolutionary ladder. Two of the most beautiful L-systems I've ever seen are those used to create the following odd creatures.

The first is called an Air Horse, sort of like a sea horse, but... different.

Listing 26.15

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóó

18 # recursion level

10 # basic angle

200 # starting thickness

C # axiom

#óóóóóóóóóóóóóóóóÅ97ÿóóóóóó Creature

C=LBW

#óóóóóóóóóóóóóóóóóóóóóóóó Body

B=[["aH]][g]]

a=Fs+;'a # upper part

g=Ft+;'g # lower part

s=[:cc!!!!&&[FFcccZ]^FFcccZ] # upper spikes

t=[c!!!!&[FF]^FF] # lower spikes

#óóóóóóóóóóóóóóóóóóóóóóóó Lungs

L=O # 8 recursions delay

O=P

P=Q

Q=R

R=U

U=X

X=Y

Y=V

V=[cc!!!&&&&&&&&[Zp]][Zp]]

p=h>>>>>>>>>>>>>>>>h>>>>>>>>>>>>>>>>h

h=[++++!F""p]

#óóóóóóóóóóóóóóóóóóóóóóóó Head

H=[cccci[>>>>>dcFFF][<<<<<ecFFF]]

d=Z!&Z!&:'d # left

e=Z!^Z!^:'e # right

i=-:'i

#óóóóóóóóóóóóóóóóóóóó Wing

W=[%[!!cb][<<<!!cb][>>>!!cb]]

b=Fl!+Fl+;'b # arc

l=[-cc{óz++z++zóz|óz++z++z}]

#óóóóóóóóóóóóóóóóóóóó

@

Fig. 26.31

The Air Horse.

The second is called a spider, but to me it looks more like a scorpion.

Listing 26.16

ó L-System Parser/Mutator ó Lj Lapre óóóóóóóóóóóóóóóóóóóó

18 # recursion level

10 # basic angle

80 # starting thickness

%C # axiom

#óóóóóóóóóóóóóóóóóóóó Creature

C=%LABHT

#óóóóóóóóóóóóóóóóóóóó Body

B=[???????zZZZ]

#óóóóóóóóóóóóóóóóóóóó Arms

A=[zf&&&&&&&&&""[pj>>>>>>>>S]][qj<<<<<<<<S]]

p=Ft+: 'p # left

q=Ft-:'q # right

j≡:'j

#óóóóóóóóóóóóóóóóóóóó Scorpers

S=["!r%^r]

L-systems are a powerful and compact way of creating complex organic shapes. There have been proposals to add L-system support to VRML, so that only the axiom and productions would have to be

transferred over the network, instead of the resulting geometry; that would greatly reduce the bandwidth required, at least for shapes like the ones we've looked at in this section.

There are many more examples of L-system input files included on the CD-ROM. If you'd like to learn more about L-systems in general, the definitive work is called "The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz and it should be available in the science section of any large library.

Other Utilities

While we're on the subject of trees, it's worth mentioning another package called TreeTool, from InWorld VR (e-mail inworld@rain.org). It's a commercial package, but it's reasonably priced and useful for generating trees with a small number of polygons. The trees used in the forest example back in Chapter 17 were created using TreeTool; Figure 26.33 shows the main screen. Various factors such as color and recursion depth can be set, and a series of random trees can be produced at the click of a button. The package doesn't support VRML, but it can produce a Sense8 NFF file which can be converted to VRML using NFF2VRML.

Fig. 26.33

The main screen of InWorld's TreeTool.

There are always new VRML tools being written; a good place to find a current list is at the VRML Repository (<http://sdsc.edu/vrml>). Look under Software, then Utilities. Be forewarned, however; some of the utilities there are better than others. Things like WRLGRID and TXT2WRL have been known to produce invalid VRML files.

Speaking of which, there's one nice utility there that's worth checking out: the VRML Authenticator.

The VRML Authenticator

It's a really clever idea. You point your Web browser to <http://www.geom.umn.edu/~daeron/docs/vrml.html>, and fill out the form resulting giving the URL of your VRML file (as shown in Figure 26.34).

Fig. 26.34

The VRML Authenticator form.

When you click on "submit," the Authenticator retrieves the VRML file and parses it, reporting back any errors. If your file is valid VRML, you'll get a message back that looks like Figure 26.35.

Fig. 26.35

All is well!

On the other hand, if you have a bogus VRML file, you'll get a message like the one in Figure 26.36.

Fig. 26.36

Uh-oh... problems!

The Authenticator is a good thing to run before announcing your VRML world to Net, since it'll warn you of any obvious problems. You may want to uncheck the box that tells the Authenticator to keep a link to your world.

That's It!

Well, you now have a mastery of a number of different tools and utilities for working with VRML. If you write any yourself, and want to share them with the world, just put them on a Web site and post a message to the VRML mailing list (www-vrml@wired.com).

From Here...

If you want to see how useful tools like FRGEN, LPARSER and especially WCVT2POV are, then check out [Chapter 22](#).



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



27-Proposed Changes for VRML 2.0

The dividing line between the changes planned for VRML 1.1 (described in [Chapter 19](#)) and those intended for future revisions (1.x, 2.0) is blurry. Some of what we'll be discussing in this chapter may appear as early as 1.1, and some of it may not appear until much later or not at all!

In this chapter, we'll be reviewing

- Areas in the specification that need to be clarified
- Proposed new VRML nodes
- New or changed fields in existing VRML nodes
- Miscellaneous other changes

Again, remember that all of this is pure speculation; a lot more information should be available by the time you read this.

Clarifications

One of the problems with VRML is that no two browsers give you the same image for any given scene, and no two support exactly the same subset of the VRML specification. One of the most important changes being made to VRML isn't really a change at all; it's a clarification of the spec. There are several key areas where clarification is needed.

Matrix Orientation

The MatrixTransform node fails to point out that the matrix is oriented differently than in most rendering systems; the rows are the columns, and the columns are the rows! This is mentioned under the description of the SFMatrix data type, but it's easy to miss; the revised spec will spell it out right in the description of MatrixTransform.

Transformation Order

Contrary to what a lot of people might assume, the transformations are not applied to an object in the order in which they appear; they're carried out backwards, starting with the one syntactically closest to the object. This will be explained clearly in the revised specification. The description I gave in Chapter 11 is the correct one.

Indexed Bindings

The whole idea of indexed bindings is confusing, but it will be made clear in future versions of the spec. In particular, many people assumed that `PER_VERTEX_INDEXED` normal binding would specify a normal for each vertex, indexed according to the `normalIndex` field. This is not the case! `PER_VERTEX_INDEXED` binds one normal to each point of each face. Confusing? You bet!

Think of it this way. Coordinate values are always bound `PER_VERTEX_INDEXED` (which is why an `IndexedFaceSet` always has a `coordIndex` field); anything else that's bound the same way should have a one-to-one correspondence with entries in the `coordIndex` field. In other words, if `NormalBinding` is `PER_VERTEX_INDEXED`, there'll be one entry in the `normalIndex` array for each entry in the `coordIndex` array, and there'll be a one-to-one relationship between the two.

New Nodes

There are several new nodes that have been proposed for VRML. They add important features to the language which are missing from version 1.0.

The Environment Node

There are various aspects of a virtual world that are not related to any specific object; instead, they're related to the environment itself. The Environment node gives world-builders a means of describing them. It allows the ambient light level and color to be specified, and allows fog of various types and any color to be added to the world (although not all browsers will necessarily support fog). It also contains coefficients for the attenuation (softening or weakening) of light and sound over distance.

The Sound Nodes

VRML 1.0 worlds are silent. Without sound, the world seems dead and empty; with sound, it comes alive. The sound nodes let you specify the URL of a file anywhere on the Internet that contains a sampled (digitized) sound. You can also set the location, orientation, and intensity of the sound, as well as the minimum and maximum range and angle of effect. I've proposed adding a description field to the node, so that people on machines without sound capabilities (or hearing-impaired users) can at least get a textual description of the sound ("you hear a train whistle").

The exact file formats for the sound samples haven't been specified, but with any luck it will be a small set; this is similar to the problem of supporting different image file formats for texture maps.

The WorldInfo Node

Many browsers can display a title for a world (usually in the title bar of their main window). The WorldInfo node allows this string to be specified, along with additional information such as author, copyrights, and so on.

The ElevationGrid Node

One of the more common things to model in a virtual environment is terrain. You may remember the fractal terrain generator I described back in Chapter 26; the terrain it produces works well, but it requires you to send a lot of vertices and polygons down the wire whenever the user downloads the world.

An alternative approach is the ElevationGrid node, which specifies a "heightfield" for the terrain. A heightfield is just a rectangular array of numbers giving the elevation (i.e., Y value) above the X-Z plane at each grid intersection; from that information, the browser is able to construct a polygon-based representation. The advantage is that a heightfield is much smaller and easier to store and transmit than the equivalent set of polygons. The actual polygonal landscape generation is done in the browser, based on the ElevationGrid node.

The GeneralCylinder Node

A lot of objects, especially man-made ones, can be described by "surfaces of revolution." You start off with a two-dimensional cross-section, or "profile," and you revolve it around some axis. The shape that gets swept out is called a "surface of revolution"; it can be used for things like vases, cups, rockets, or anything that has axial symmetry. As with heightfields, it's a lot more compact to send the profile and revolution information than to send all the generated vertices and faces.

The Background Node

Many of the VRML worlds you find floating around the net are doing just that: floating. They exist in a timeless, empty void with no sense of a surrounding environment. One proposed VRML node will solve that problem, by allowing a world-builder to specify colors for the sky and the ground; this Background node will also contain information about the "backdrop" image to use.

New or Changed Fields

Several of the features being proposed for VRML don't require new nodes, but they do require that existing nodes gain additional fields to store new information. Others require fresh interpretations for existing fields.

Texture Mapping

A transparencyColor field has been proposed for the Texture2 node, which would allow a kind of "chroma-key" effect on texture-mapped surfaces. The specified color in the texture map would be treated as transparent, allowing you to see through to the other side. This is not required for those texture file

formats (such as PNG) that have an alpha channel.

A quality field has been proposed for the Texture2 node; this would allow world-builders to specify whether simple bi-linear texture interpolation is sufficient, or whether correct (but slow) perspective mapping should be used.

A type field in the Texture2 node would indicate whether the texture should be "blended" with the computed surface color, or whether it should simply replace it.

The filename field in the Texture2 node might become an MFSTRING, so that multiple "frames" of an animated texture can be referenced. Another use for this would be multiple texture maps per object (currently, VRML can only have one texture map for each object). If this is implemented, a textureIndex field would be added to the IndexedFaceSet node so that the textures can be associated with individual faces.

Another way of specifying animated texture maps would be to allow a Texture2 node to reference an MPEG, QuickTime, AVI, or FLC file.

New ShapeHints

Currently, all VRML browsers tessellate Spheres, Cones, and Cylinders however they choose; a new tessellation field has been proposed for the ShapeHints node, to control the number of faces to generate.

A primitiveVisibility field would allow world-builders to indicate whether a primitive (Sphere, Cone, Cylinder, Cube) is visible only from the outside, only from the inside, or both.

Other Fields

The map field of the WWWAnchor node would become an SFBitmask, which would specify which parameters should be sent to the far end when the anchor is selected. Any combination of POINT, ORIENTATION, and TEXTURE could be sent.

Miscellaneous Changes

The name field of WWWInline would become an MFString, so that alternative locations for the data can be specified.

A prototype node would allow the creation of an object without instancing it.

Support for internationalization would let you use characters not found in the English language.

So What's In and What's Out?

Good question. VRML 1.0 is not very old; predicting exactly what will be in VRML 2.0 is next to impossible.

However, there are two areas where VRML will definitely be developing: behavior, and multiuser

worlds. I'll be discussing these in the next two chapters.



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



28-Behaviors

People enjoy making fun of acronyms. IBM once earned the title "Itty Bitty Machines," the programming language LISP became "Lots of Incredibly Silly Parentheses," and PCMCIA has come to stand for "People Can't Memorize Computer Industry Acronyms."

VRML's nickname is the "Virtual Reality Museum Language," since a surprising number of VRML worlds are in fact museums and art galleries. The reason for this is simple: most real-world museums are relatively static, unchanging places where you're not allowed to touch anything. Early VRML worlds are the same way: there's no action, no life, and you can't really "touch" anything in a way that affects it. All you can do is wander around and look at the pretty pictures.

In a way, first-generation VRML isn't really VR at all. Back in Chapter 3, I gave a definition of virtual reality; part of that definition required that you not only be able to view the world, but manipulate it as well. Motion and responsiveness are essential if we're to build anything in cyberspace besides museums. What we want to do is give objects behavior.

In this chapter, we'll look at

- What we mean by behavior
- How behavior might be added to VRML
- What a "browser API" is
- What "events" are and where they come from
- The different types and "levels" of behavior
- Some examples of how behaviors might work

If you're not a programmer, you might find parts of this chapter challenging; that's a good thing. If you're planning on adding behavior to your VRML worlds, you'll eventually have to bite the bullet and learn how to write some simple computer programs.

What Is Behavior?

In the context of VR and VRML, behavior is change. Among the things that can change about an object are its location, its orientation, its properties, and its relationship to other objects. Behavior also encompasses the notion of "responsiveness," the ability of objects to react to the user and to each other.

There have been a wide variety of proposals for adding behavior to VRML. The first draft of the VRML specification deliberately avoided dealing with the issue of behavior, because it was recognized as being a thorny and potentially divisive subject.

However, there is a growing feeling of consensus within the community. A number of the behavior proposals are similar enough to each other that it seems clear that there will soon be a convergence.

At the moment, many of the details of the final proposal are still unclear. In this chapter, we'll take a look at the one possible implementation of behavior in VRML systems; however, it's not the only approach, and it's not clear by any means just how the final behavior proposal for VRML will look.

How Will Behavior Fit into VRML?

It's quite possible that the only real change to the VRML specification itself will be the addition of a single behavior field to a Separator. The field value will be a URL (or URN) for a behavior "script" written in some programming language.

The programming language itself will not be specified; this is a deliberate (and wise!) choice, since no one language is ideal for all applications. It also avoids the danger of "language wars," in which everyone campaigns for their favorite programming language to be the chosen one.

Having said that, it's important to also say that Java is a leading contender for a behavior language for VRML. It's free, widely available, and Web-friendly. However, none of what follows is in any way specific to Java or any other programming language; the concepts are independent of their implementation.

Sidebar

Java

Since Java is a complete object-oriented programming language, it's not practical to describe it in any detail here. However, the basic concepts are important.

Most languages (like C) compile to machine instructions, which are specific to a particular processor (like the Intel x86 family, for example). A program compiled for a PC won't run on a Macintosh, and one compiled for an SGI won't run on a Sun.

Java, by contrast, compiles into a "bytecode" language that is run by an interpreter on the target machine. The bytecode language itself is platform-independent, so it can run without change on any computer and on any operating system. You can compile your program once, and have it run on a PC, a Macintosh, an SGI, or a Sun.

Java bytecode data can be downloaded over the Web, and the runtime interpreter is usually built right into the user's Web browser. Behavior can be embedded into any document (including a VRML file) in the form of Java "applets." The tight integration of Java into Web browsers gives it a tremendous advantage as a behavior language for VRML.

When the browser loads a VRML file, and encounters a Separator containing a behavior field, it retrieves the behavior script the same way it would a texture map or another VRML file; it establishes an HTTP connection and requests the data. As part of that transaction, the browser gets informed of the MIME type of the data; that MIME type specifies what language the script is written in.

Based on that information, the browser then loads a language interpreter (assuming it doesn't have one already built-in). It requests that the interpreter load the script and parse it; once this is done, the behavior is ready to go.

Note that the scripts are only loaded as needed, subject to LOD (see Chapter 17) and object culling. This means that a behavior will probably not be loaded right away, and may be "unloaded" when an object is no longer relevant.

Sidebar

What About ActiveVRML?

Microsoft has recently announced "ActiveVRML," a language based on a functional programming paradigm (rather than a procedural one like Java or C). ActiveVRML is not some variant on VRML; it's a completely different language, syntactically and conceptually. The only relationship it has to VRML (aside from the name) is that it uses the VRML file format to describe object geometry.

ActiveVRML duplicates much of the functionality of the existing VRML format for things like transformations and lighting. It adds behavior implicitly, and is intended more as a replacement for VRML than as an extension to it.

ActiveVRML is an extremely powerful language. It already has advanced features that VRML itself won't have for some time to come. It can leverage the existing base of VRML files by importing them as static geometry. What's more, Microsoft is planning to make the complete source code for an implementation of ActiveVRML available for free, in the hope that it will become a standard.

Despite all this, there is strong resistance to ActiveVRML within the VRML community, and it remains to be seen if ActiveVRML will gain the wide acceptance that VRML itself has. There are also many, many issues surrounding the integration of VRML and ActiveVRML.

In a sense, ActiveVRML turns the model of how things should work inside-out. Instead of the geometric description language (VRML) loading behaviors (in a language like Java), the behavior language (ActiveVRML) loads geometry (in VRML). These are two fundamentally different approaches to the problem of integrating behavior with geometry, and it remains to be seen which one prevails. In the remainder of this chapter, I'll be discussing behavior referenced from VRML, rather than VRML referenced from a behavior language, simply because there's a lot more information available about that approach.

The API

Since behavior is all about change, there has to be some way for the behavior script to actually make changes to the world itself and the objects it contains. The programmer who constructs the behavior script needs access to an APIó an Application Programmer's Interface, which will allow changes to be made to the data structures inside the browser.

This is where the controversy really begins. Since the internal data structures vary from browser to browser, it's hard to find a level of abstraction that works for all of them. This is likely to be a source of much discussion within the VRML community in the future.

At the very least, we can expect that a browser API will allow behavior scripts (or even external application programs written in other languages) to make changes to individual objects. Obvious changes include the location and orientation of an object, as well as its material properties and texture.

There needs to be some way to move an object to a different point in the hierarchy (usually called "re-parenting"); for example, if you put a box on a cart then it should remain on the cart, regardless of how the cart itself moves. There should also be some way to create objects (either from raw VRML or by giving the URL of an existing VRML file), and some means of destroying them.

In addition, scripts should be able to access information about the browser (such as which version is being run), the world (such things as the title and URL) and the user (such as their current navigation mode). There should also be a way of finding out the current simulation time and the current (real-world) wall-clock time.

Events

A browser API will also allow a script to register its interest in certain "events." If you've ever written a program for a window system of any kind, you're already familiar with event-driven program structure. When the user clicks a mouse on the screen, it causes a "mouse event" to be sent to the current application program. The same thing happens when the user presses a key, selects a menu item, opens or closes a window, and so on.

These events are handed to the application program, which decides what to do with them. In many cases, they can simply be ignored; however, that's for the programmer to decide. If he or she wants to catch every keystroke and mouse click, they certainly can.

In a VRML browser, there are a number of different "events" that can occur. Table 28.1 gives a partial list.

Table 28.1

PICK	the mouse has been clicked on an object
GRAB	the mouse is being held down on an object
RELEASE	the mouse is no longer being held down on an object
DRAG	an object which has been GRAB'ed is being moved

ENTER	the user has entered the bounding box of an object
LEAVE	the user has left the bounding box of an object
COLLISION	the user has collided with an object
ACTIVATE	the object has become visible (not culled, or LOD'd out)
DEACTIVATE	the object has ceased to be visible (culled, or LOD'd out)
FRAME	occurs once every frame
INIT	occurs once when the behavior is initialized
END	occurs once when the behavior is de-initialized

TIP

The COLLISION event is not very well defined right now, since collision detection isn't really addressed in VRML.

These events are sent to the behavior script, in just the same way that mouse clicks and keystrokes are sent to an application program in a window system.

Although the exact mechanism for sending the events hasn't been spelled out, the most likely one is to have functions in the script that are called by name for certain events. For example, the INIT event might call a function called `Init( )` within the script. The parameters would include the event itself, the object it occurred on, and possibly additional information. If a particular function isn't defined in the script, the event is ignored. For example, a script that didn't need to perform any initialization would simply not have an `Init( )`; this would not be considered an error.

However, almost all behavior scripts would have an `Init( )` function, since that's most likely where the script would inform the browser of what events it's interested in. It might say (for example) that it's interested in PICK events that occur on it, or any of its children.

By default, any events for which no script has been specified are simply passed on to the parent of the object, and from there to the grandparent, and so on.

Characteristics Fields

The main advantage of referencing scripts by their URL is that they can be re-used. However, this can result in multiple objects having identical behavior; it would be nice if there were some way to "customize" objects. For example, let's say you wanted a dozen tumbleweeds to be rolling through an Old West town; if they all move in perfect synchronization, it would look very odd and unnatural. What you want to do is specify a "phase offset" for each one, to introduce some variation; it would be a nuisance to have to do that in the behavior script, so it's been proposed that an open-ended characteristics field be defined. It would have an MFSTRING value, in which each string is a "keyword=value" pair; for example, "tumble_offset=7." The scripts could access these characteristics and make decisions based upon them. They would be script-specific, so there's no danger of the same name being used for two different purposes.

Passing Events

Each behavior script has the option of ignoring an event, dealing with it directly, or passing it on up the object tree or to another object altogether. It might even map it into a different kind of event.

For example, let's build a simple robot. We want it to dance whenever we touch it; we therefore have a behavior script that registers an interest in the PICK event. When any part of the robot is touched, a PICK event is generated; since the individual body parts don't know about PICK events, they just hand the event up the tree until it gets to the top-level object that knows how to deal with it. The robot's script receives the event, and causes the "dance" behavior to take place (the behavior script uses the API to rotate the various limbs of the robot).

Now, let's say that when we touch the robot's hand, we don't want it to dance; instead, we want the hand to change color. No problem; we just add a behavior script to the hand that receives the PICK message and performs the color change. It doesn't forward the message up the tree, so the dancing behavior never happens. What if we want both? Again, no problem; the script for the hand does the color change, and then relays the message up the tree.

In effect, our robot has a virtual "nervous system," with signals (events) being sent up to the brain (the robot's behavior script).

Levels of Behavior

Notice that we have (so far) defined two distinct interface layers: the API is a "level 0" interface, between the behavior script and the browser. The events that invoke functions in the behavior script are a "level 1" interface. This distinction is important; as we'll see later, there are even higher levels of behavior.

Some Examples

In order to get a handle on how all this fits together, we're going to look at a few examples of behavior. Rather than use a real scripting language (since that would require you to have some programming knowledge), we'll do it all in a kind of "pseudo-code."

The simplest behavior, of course, is no behavior at all. This is the "behavior" exhibited by all VRML 1.0 objects, and it's also the behavior that the vast majority of objects in any virtual world will have; just as in the real world, most things don't move on their own. However, that's not a very interesting case; let's begin with a simple rotation behavior.

The Spinning Cube

Let's say we create a cube, and we want it to spin on its axis.

Separator {

behavior "http://somewhere.someplace.org/~somebody/spin.lng"

Transform {

translation 12 7 42

rotation 0 1 0 0

}

Material { diffuseColor 0 0 1 }

Cube { }

}

The ".lng" extension is fictitious, and is meant to represent a language. In practice, it might be ".tcl" for TCL, or ".pbs" for Perl Binary Script. The behavior script would have an Init() function, and a Frame() function:

Init()

register interest in Frame events for this object

Frame()

increment the rotation angle of the object

The Init() function would tell the browser to call the script's Frame() function every frame. On every frame, the cube would be rotated slightly more than in the previous frame, causing it to spin.

That's great, but there's a problem: the cube won't spin at a constant rate! Since the frame rate varies with the complexity of the scene, a more complex scene would make the cube's rotation slow down. What if we wanted it to remain constant? It turns out that we can use the simulation time for that purpose.

Init()

register interest in Frame events for this object

Frame()

compute the current rotation angle using the current simulation time and the rotation period

set the object's rotation angle

The computation is simple; we take the current simulation time, modulo the rotation period, divide by the rotation period and multiply by 2π to convert to radians. For example, if the current time is 1795 ticks, and the rotation period is 100 ticks, we're 95/100ths of the way around the circle. That's 5.969 radians.

The interesting thing about using the simulation time is that we can predict the orientation of the cube at any time. When we start building multi-participant worlds, we can be sure that everyone sees the cube in exactly the same position (since the simulation time would be synchronized across machines). Using the simulation time makes it easier to build multi-participant worlds.

The rotation period could be a constant, or it could be specified in the characteristics field described

earlier. Using the characteristics field would allow us to have lots of cubes, each spinning at a different speed, without having to have different scripts for each one.

A Clock

Let's say we want to build a virtual wall clock. It'll have three hands: hour, minute, and second. There are two approaches we can take; either we could have a script for each hand, or one script for the entire clock. Let's start with the first approach.

```
DEF Clock Separator {  
DEF Hour_hand Separator {  
behavior "spin.lng"  
characteristics "period=43200" # seconds per twelve-hour _period  
Transform {  
rotation 0 0 1 0  
}  
Material { diffuseColor 0 0 0 } # black  
WWWInline { name "clock_hand.wrl" }  
}  
DEF Minute_hand Separator {  
behavior "spin.lng"  
characteristics "period=3600" # seconds per hour  
Transform {  
rotation 0 0 1 0  
scaleFactor 1 0.75 1 # minute hand is shorter  
}  
Material { diffuseColor 0 0 0 } # black  
WWWInline { name "clock_hand.wrl" }  
}  
DEF Second_hand Separator {  
behavior "spin.lng"  
characteristics "period=60" # seconds per minute
```

Material { diffuseColor 1 0 0 } # second hand is red

WWWInline { name "clock_hand.wrl" }

}

}

The "spin" behavior script is used for all three hands; it's basically the same as the script we looked at for the spinning cube. The period (in seconds, this time) for the three hands is different, so they progress around the face at a different rate.

Instead of having a script on each hand, we could have had one script for the entire clock:

DEF Clock Separator {

behavior "clock.lng"

DEF Hour_hand Separator {

Transform {

rotation 0 0 1 0

}

Material { diffuseColor 0 0 0 } # black

WWWInline { name "clock_hand.wrl" }

}

DEF Minute_hand Separator {

Transform {

rotation 0 0 1 0

scaleFactor 1 0.75 1 # minute hand is shorter

}

Material { diffuseColor 0 0 0 } # black

WWWInline { name "clock_hand.wrl" }

}

DEF Second_hand Separator {

Material { diffuseColor 1 0 0 } # second hand is red

WWWInline { name "clock_hand.wrl" }

}

```
}
```

In this case, the clock behavior script would do the following:

Init()

register an interest in Frame events

find the child object called "Hour_hand" and store a pointer _ to it

find the child object called "Minute_hand" and store a pointer _ to it

find the child object called "Second_hand" and store a pointer _ to it

Frame()

get the "seconds" value of the current wall-clock time

divide it by 60

multiply it by 2p to get rotation angle in radians

set the second hand's rotation

get the "minutes" value of the current wall-clock time

divide it by 60

multiply it by 2p to get rotation angle in radians

set the minute hand's rotation

get the "hours" value of the current wall-clock time

divide it by 12

multiply it by 2p to get rotation angle in radians

set the hour hand's rotation

Which approach to programming the clock is better? Depends on your application.

Responsiveness

So far, all our examples have been for objects whose behavior is strictly a function of time; the only events they respond to are FRAME events. Let's look at an object that responds to user interaction.

We've already seen an example of one such object, our dancing robot. Now we're going to create a sphere that starts off blue, but turns red when you touch it.

Separator {

behavior "color_change.lng"

```
Material { diffuseColor 0 0 1 }
```

```
Sphere { }
```

```
}
```

The color_change script does the following:

```
Init()
```

```
register an interest in PICK events
```

```
Pick()
```

```
set diffuseColor to 1 0 0
```

That's it. The first time you touch the sphere, it turns red. It will continue to receive PICK events, and will continue to turn red each time it does, but you won't notice any change. There's no Frame() function, since the object doesn't change over time except in response to being PICKed.

Notice that we begin to see a kind of hierarchy emerging: the most primitive kinds of objects are those that don't have any behavior at all (such as mountains and trees). The second kind are objects whose behavior is strictly a function of time; this includes our spinning cubes and our clocks, as well as such natural phenomena as the moon orbiting the earth, rushing waterfalls, and the daily day/night cycle. We've now seen that there's a third kind: objects that respond to stimuli.

State

One term that comes up a lot when discussing behavior is "state." It simply refers to changeable information about an object; we speak of the object's "current state" as being the set of values that describe it at a given moment in time.

For example, consider a sphere that toggles between red and blue each time it's PICKed.

```
Separator {
```

```
behavior "color_change.lng"
```

```
Material { diffuseColor 0 0 1 }
```

```
Sphere { }
```

```
}
```

The script for changing colors is

```
Init()
```

```
register an interest in PICK events
```

```
Pick()
```

if diffuseColor is 1 0 0, set diffuseColor to 0 0 1

otherwise, set diffuseColor to 1 0 0

This will effectively toggle between the two states.

TIP

It's not hard to see that we could easily set up a traffic light, just using what we've discussed so far; the simulation time would be used to control whether each of the three lights (analogous to the clock hands) are dark or lit.

State is a really important concept. Let's look at an example that has more than two states: a door.

Separator {

behavior "door.lng"

Transform {

rotation 0 1 0 0

}

WWWInline { name "door.wrl" }

}

The door's script maintains the state information for the door. The door has four states: open, closed, opening, and closing. The behavior script implements what's called a "state machine."

State machines are very powerful, and easy to implement.

Init()

set the state to "closed"

set the door's rotation angle to zero

Pick()

if state is "opening" or "closing", do nothing;

otherwise...

if state is "closed", set state to "opening"

if state is "open", set state to "closing"

save current time in a variable called "start_time"

Frame()

do something different depending on the current state...

open: do nothing

closed: do nothing

opening:

take the current time

subtract the start_time

divide by the length of time it takes for the door to open

multiply by the maximum angle of the door

set the rotation angle of the door to that value

if the angle is greater than or equal to the maximum angle, set the state to "open"

closing:

take the current time

subtract the start_time

divide by the length of time it takes for the door to open

multiply by the maximum angle of the door

subtract that angle from the maximum angle of the door

set the rotation angle of the door to that value

if the angle is less than or equal to zero, set the state to "closed"

If you click on a closed door, it will enter the "opening" state and continue opening (on each frame) until it's fully open, at which point it enters the "open" state. If you click on an open door, it will enter the "closing" state and continue closing (on each frame) until it's fully closed, at which point it enters its "closed" state.

Notice that "state" is different from "characteristics"; characteristics are things that make each instance of an object different, whereas state is what makes a single object different as a function of time.

For example, consider a virtual squirrel. I use virtual squirrels as examples because the University of Waterloo campus is infested with them, so they're always on my mind. All the squirrels would share the same behavior script and the same geometry, but each would have its own unique characteristics. For example, one characteristic might be the distance that a potential predator has to be at before triggering the "run away" behavior. The squirrel also has state information; for example, how hungry or tired it is. Its decision-making will be based on some combination of those factors.

Relaying MessagesóA Flashlight

For our next example, let's consider a flashlight. When you click on its switch, the light comes on; click again, the light goes off. The interesting thing is that the switch itself also changes from an "open" position to a "closed" position.

```
DEF Flashlight Separator {  
behavior "flashlight.lng"  
DEF Bulb Separator {  
WWInline { name "bulb_geometry.wrl" }  
DEF Light SpotLight {  
on FALSE  
}  
}  
DEF Toggle_switch {  
behavior "switch.lng"  
Transform {  
translation 0 0 0  
WWWInline { name "switch_geometry.wrl" }  
}  
WWWInline { name "flashlight_body_geometry.wrl" }  
}
```

The behavior field references a file called "switch.lng."

The script for the switch would be:

```
Init()  
set switch_state to be zero  
Pick()  
if switch_state is zero, set it to one  
otherwise, set it to zero  
if switch_state is zero, set Z translation to -0.1
```

otherwise, set Z translation to 0.0

forward "pick" message up the tree to the flashlight behavior _script

The flashlight script would be:

Init()

set the light to be off

Pick()

if the message was not from the switch, ignore it

otherwise...

if the light is off, turn it on

otherwise, turn it off

The switch script moves the switch when it's touched, and the flashlight script responds to the same PICK message by toggling the state of the light.

NOTE

In some browsers, the flashlight may not work as you would expect. If the browser strictly complies with the VRML 1.0 specification, then light stops at Separators; that means the flashlight can never shine on anything! Fortunately, many browsers are ignoring this peculiar "feature." Still, this is an issue that will have to be addressed in future releases of the VRML specification.

There are certainly other ways to create a flashlight; for example, you might decide not to have a script on the switch, and instead have the flashlight script move the switch to a new position when a PICK message is received.

You could make the flashlight more realistic, of course; for example, you could give it a virtual "battery" that runs down.

Init()

read battery's initial charge from its characteristics field

store it in a variable called "energy"

Frame()

if the flashlight is on, decrement energy by depletion_rate _times elapsed time

if energy is less than 50%, scale the light's intensity by _the energy remaining

Each flashlight you create could of course have a different initial charge in its battery.

Sending Events Between Objects

So far, events (sometimes called "messages") have been travelling upwards within an object tree. That's fine for a flashlight or other self-contained object, but what if we wanted to create a ceiling light fixture controlled by a switch? The fixture and the switch would probably be in different sub-trees.

Well, event messages can be sent to other trees as easily as they can be to the trees their associated objects are in.

Separator {

DEF Fixture Separator {

behavior "fixture.lng"

WWWInline { ... }

}

DEF Lightswitch Separator {

WWWInline { ... }

}

}

The fixture script would do the following:

Init()

register an interest in PICK events on the Lightswitch node

Pick()

if the event is from the Lightswitch node

if light is off, turn it on

otherwise, turn if off

Registering an interest in events on another object requires you to identify the object by name; that raises an interesting question. What if the Lightswitch had been WWWInline'd? What if it had been a child of a WWWInline'd node? The current VRML specification says that DEF'd names are file-scoped, so you wouldn't be able to address nodes across a WWWInline. That will probably be changed in future releases of the specification.

What if we wanted to have a second switch? Say, one at the top of the stairs and one at the bottom? No problem.

Separator {

DEF Fixture Separator {

behavior "fixture.lng"

WWWInline { ... }

}

DEF Lightswitch1 Separator {

WWWInline { ... }

}

DEF Lightswitch2 Separator {

WWWInline { ... }

}

}

The fixture script would now check for either switch:

Init()

register an interest in PICK events on the Lightswitch1 node

register an interest in PICK events on the Lightswitch2 node

Pick()

if the event is from the Lightswitch1 node or from the _Lightswitch2 node

if light is off, turn it on

otherwise, turn it off

Pretty simple.

A Virtual Nervous System

Notice that the state information is maintained in exactly one place: the fixture. The decision-making about state changes is also centralized; the individual switches don't each have their own opinion as to whether the light is on or off, they simply send messages to the "brain" (in this case, the behavior script for the fixture) which makes the decision and updates the state. Since the script can perform operations on other objects (like making our robot dance), what we have is essentially a sensor/brain/effector model, a virtual "nervous system." This will be very important later, when we start dealing with multi-user worlds.

The nervous system approach is important for another reason: keeping track of state. To understand what we mean by this, let's take another look at our two-lightswitch example.

Let's say that you have a very large world. The light switches are far away from each other, and from the

fixture. They're so far, in fact, that LOD and culling means you would not necessarily have both switches and the fixture loaded simultaneously. But if the fixture isn't loaded when one of the switches is PICKed, what happens?

A related question has to do with the flashlight; if you turn it on, then wander off (far enough away so that the flashlight is no longer being kept in memory in the browser) and you return again later, is the flashlight still on? Does its state get preserved, even when the flashlight is not active?

One approach to answering these questions (not the only approach, and not necessarily even the best, but an informative one) would be to simply have one behavior script for the entire world. The world would look like this:

```
Separator {  
behavior "world.lng"  
WWWInline { name "world.wrl" }  
}
```

The world.wrl file would describe the world's geometry, and the world.lng file would describe its behavior and keep track of its state. Since the default behavior for all events is to get passed up the tree, they would all arrive at the world.lng script. The world.lng script could then respond (based on what object an event is from) and make changes to the world.

That one script would have access to all the state information, and it wouldn't matter which parts of the world got loaded or unloaded; the state of the flashlight, the state of the light fixture, and all the other state information would be kept in one place. If you decided to leave this world, all the state information could be saved in a file; when you re-entered the world later, the state could be restored.

The world.wrl file could be reused by countless different worlds, each with its own unique state and behaviors (sort of like the "parallel universes" that are so popular in science fiction).

This all works fine, but there's a problem: it doesn't scale very well. It's also a maintenance nightmare; every time a new kind of entity is added to the world, additional code has to be added to the ever-growing behavior script. The script would have to know how to make the robot dance, the flashlight switch slide, the sphere change color, and so on.

You could partition things up, of course. For example, you might have a single behavior script for each room; it would keep track of which doors are opened and closed, which lights are on and off, and so on.

However, there's a better way. There's an old saying in the computer field: every problem can be solved by adding one more level of abstraction. That's not really true in every case, but it's certainly true here.

Another Level of Behavior

Instead of simply referring to "behavior" in general, let's subdivide it into three distinct levels:

- level 0: updates the object properties directly
- level 1: updates the object properties (using level 0) as a function of time

- level 2: responds to input and maintains state information

The level 0 behavior is what the browser API provides. The level 1 behavior is provided by the methods (functions) of the behavior scripts on each object. Level 2 behavior is actually something quite distinct from level 1, in that it keeps state information around and makes decisions.

You can think of the level 2 behavior as the higher level of brain function; receiving and processing information and making decisions. In a human being, this would be the part that decides to walk to the kitchen to make a sandwich. Level 1 behavior is a much lower-level; it deals with things like knowing how to move your legs when you walk. Level 0 behavior is the simplest: moving or rotating individual body parts.

These behavior levels perform the same function as the multiple layers of the ISO (International Standards Organization) reference model do for networking.

Let's look at an example.

Level 1 behavior scripts could be written that handle things like the door opening and closing or the robot dancing, but the actual decision-making (when to start opening, closing, or dancing) would be left to a separate, higher-level behavior.

Separator {

behavior "world.lng"

DEF Robot Separator {

behavior "robot.lng"

WWWInline { name "robot.wrl" }

}

DEF Flashlight Separator {

behavior "flashlight.lng"

WWWInline { name "flashlight.wrl" }

}

}

The robot's script would detect PICK events and send a message up the tree to the world.lng script, which would then send a message to the robot, saying "start dancing."

The flashlight script would detect PICK events on its switch, and send them up the tree; all other events (such as PICKing the body of the flashlight) would be discarded. The world.lng script would receive the PICK event from the flashlight and keep track of the flashlight's state; it would also send a message back down to the flashlight telling it to turn on.

If the flashlight became inactive (for example, because of LOD), it wouldn't matter; its state is maintained by the world.lng script. When the flashlight gets re-activated, its Activate() function would

send a message to the world.lng script asking for update information, and the world.lng script would send back a message giving the flashlight's current state.

This ensures that state information exists in only one place, and avoids having the "brain" of an object deal with the mechanics of its operation. The functions defined in an object's behavior script are truly "methods" in an object-oriented programming sense: ways of responding to messages. The details of the robot (including its geometry and how to access it) are kept hidden inside the robot itself, and only messages like "dance, robot, dance!" are used to control it.

This approach also extends cleanly to a multi-user networked environment, in which the "level 2" behavior (the "brain," if you will) is on a different computer.

For example, consider an airplane in flight. If you wanted to do an accurate simulation of aircraft flight using computational fluid dynamics, you could certainly do it; however, that's not going to run inside a browser on a 486. So you put the complex calculations (the level 2 behavior) on a massively parallel supercomputer, and have it send out simplified approximations of the plane's behavior; these simpler approximations are executed in the browser as a level 1 behavior, which in turn calls the level 0 behavior (the API) to actually re-position the plane on every call to the Frame() function.

When a bird hits the plane (or gets sucked into the engine!) the browser detects the interaction and sends a message up to the level 2 behavior script (still running on the supercomputer), which updates the flight model and sends fresh updates down.

For a human user, the same thing happens; however, the level 2 behavior is in "wetware"óthe user's brain. The user communicates to the level 1 behavior scripts using various input devices; if they tilt a joystick forward, it invokes the "move forward" function in their avatar's behavior script. That function in turn might move the virtual legs, swing the virtual arms and so on, as well as translate the avatar in the correct direction at the correct velocity. This is an extension of the "dead reckoning" concept that we'll be discussing in the next chapter.

From Here...

Although the details of how behavior will be implemented in VRML still aren't crystal clear, we know in general terms how it will work. Once the standards are established, you'll be able to start designing worlds in which objects move and respond and give the appearance of being alive.

However, some of the objects will do more than just appear to be alive; they actually will be, since they'll be controlled by other human beings. The place to go from here is [Chapter 29](#), in which we'll take a look at the technology behind multi-user virtual environments.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



29-What Comes Next?

The title of this chapter poses a simple question, and in this case there's a simple answer. What comes next in the evolution of VRML are fully interactive, multiuser virtual environments. In the jargon of the industry, this is also known as "distributed" VR.

In this chapter, we'll be discussing

- What distributed VR is all about
- Some of the challenges it poses
- Different approaches to meeting those challenges
- Existing protocols, including DIS
- Dead reckoning and behavior
- Broadcasting and multicasting

As with the previous chapter, none of this is etched in stone; there are as yet no standards for sharing VRML worlds over networks. In fact, there's much less of a consensus in this field than there is in the area of behavior.

In this chapter, I'll simply describe some of the issues and proposals. Most of what follows is based on a paper I wrote for VRML '95, an annual conference dealing with issues surrounding the future of the VRML standard. You can find this document online, at <http://sune.uwaterloo.ca/~brochl/distrib.html>.

What Is Distributed VR?

The idea behind distributed VR is very simple; a simulated world runs not on one computer system, but on several. The computers are connected over a network (such as the Internet) and people using those computers are able to interact in real time, sharing the same virtual world. In theory, people can be sitting at home in London, Paris, New York, and Edmonton, all interacting in a meaningful way in VR.

It's clear that there are a number of obstacles to be overcome in achieving this goal. The fact that we want people to be able to access it from their homes means that we have to be able to run over relatively limited-bandwidth links, such as 28.8k modems. The fact that we want to run over the Internet means

that we have to tolerate a certain amount of latency in the delivery of update information. Finally, the fact that people are running on different computer systems with different hardware and different software means that we must design the system for portability.

Some Jargon

Each of the computers participating in the simulation is called a "host." On each host there are a number of "entities" (things in the virtual environment) that communicate their changing state by sending "update messages." The specific entity that corresponds to a human participant's virtual body is called its "avatar."

The Naive Approach

The simplest approach, and the one that some multiplayer computer game systems use, is to simply have each host broadcast the location of each entity that it maintains. These broadcasts are received by every host in the simulation, and are used to update their local copy of the "world database."

This approach works acceptably on small, dedicated networks; however, there are a number of problems with it. The most important problem with broadcasting is that every machine on the subnet must receive and process every update packet; this includes machines which aren't participating in the simulation! That's not a problem on a dedicated LAN, but networks that are being used for other things can't have huge amounts of broadcast traffic on them; this is why so many companies and universities adopted a "no network DOOM" policy. (DOOM, incidentally, switched to point-to-point messages for subsequent releases in order to be more friendly to non-dedicated networks.)

The other problem is that simply broadcasting your current location and orientation requires a lot of bandwidth, since those (potentially) change every time through your simulation loop. When you're dealing with a large number of entities and limited bandwidth, the system just stops working.

In short, the naive approach doesn't scale up.

Central Servers

For many years, the Internet has had MUDsóMulti-User Dungeons (or Multi-User Dimensions, depending on who you ask). Users connect to them via TELNET or with a special-purpose "front end" program, and share text-based virtual environments with each other.

MUDs are simple, since each one is based on a single central server. The world is guaranteed to be consistent and up-to-date, since it's being maintained in one place. Unfortunately, that also produces a bottleneck; in fact, it produces several bottlenecks. Since all the traffic goes to one host, there's a huge amount of network congestion. Since a single server is doing all the work, the CPU also gets bogged down. The MUD server also has to maintain the entire world in memory at once, or spend a lot of time swapping parts of it out to disk.

In short, a central server doesn't scale well either.

How Have Other People Solved These Problems?

In designing a standard for representing scene geometry, the VRML group chose to use a subset of an existing format as its jumping-off point, rather than create yet another new format. This approach has worked well; maybe we can do the same thing in selecting standards for distributed VR?

The largest, best-known, and most successful standard for distributed VR has been DISóthe Distributed Interactive Simulation protocol. As its name suggests, it addresses the very issues we're concerned about. Perhaps we could simply adopt it, and declare our task complete?

Well, as it turns out, it's not quite that simple. DIS, like its predecessor SIMNET, is designed for a very specific application domain: military simulations. As we'll see, it's extremely well-optimized for that application, but is not an ideal choice for our needs. However, we can benefit from a lot of the work done on DIS; they've already encountered and dealt with a lot of the same challenges we'll be facing in designing a distributed VR standard.

More information about DIS is available from the Naval Postgraduate School:
<ftp://taurus.cs.nps.navy.mil/pub/mosaic/nps_mosaic.html>.

The Basic Ideas of DISóPDUs and Dead Reckoning

As mentioned earlier, the three basic challenges are:

- Compatibility
- Limited bandwidth
- Latency (lag)

DIS addresses the first of these problems, compatibility, by defining a standard message format for interchanging information between simulation hosts. This standard format is called a PDU, for Protocol Data Unit. There are many different types of PDU, most of which (like requests for resupply of munitions) are not applicable to our needs. However, one of the formats, the Entity State PDU or ESPDU, contains information very similar to what we need to send between hosts.

The second and third problems, limited bandwidth and update latency, are dealt with in DIS using a technique called "dead reckoning." The idea behind dead reckoning is simple: instead of just sending an entity's location, a host sends a message (an ESPDU) that contains the entity's location, a timestamp, and a velocity vector. Using that information, each host in the network can then extrapolate the entity's location without additional updates. This can be done for entity orientation as well as location.

Each entity runs its own full-up simulation, and also runs the simple dead reckoning model for itself. It keeps track of the actual location predicted by the two models; when they diverge by more than a certain amount, it sends out another update to bring everyone's model of the entity back in line with what the entity is actually doing.

In addition, each entity sends out periodic update messages (say once every five seconds or so) that act as "keep-alives." These keep-alives actually serve three purposes: if someone enters the simulation after it's started, they'll get caught up with every entity's state within a few seconds. If update messages get missed

for any reason, the system recovers gracefully; within a few seconds, a new update message will arrive and bring the host up to date. If no updates are received from an entity within a certain period of time, the other hosts on the network can assume the entity is no longer around.

Dead reckoning produces impressive results. Since updates are only sent when needed, the amount of traffic is greatly reduced. The system works particularly well for the problem domain it's designed for: military simulation. Since every entity is either a vehicle or projectile, the dead reckoning approach works nicely.

Limitations of DIS

However, a more general distributed VR system doesn't work nearly as well with standard DIS. Let's say we want to simulate a park, for example; we want to have birds fluttering around overhead, squirrels foraging for nuts, and so on. Despite the fact that birds do occupy airspace, their behavior is surprisingly unlike that of a jet fighter aircraft. And despite the fact that squirrels are land-based, they're not very much like tanks.

Perhaps more importantly, people don't move anything like vehicles; nor do they behave like projectiles (unless you fire them out of a cannon, which is an exceptional case).

A proposed extension to DIS by Pratt, Barham, and a number of other people is described in "Insertion of an Articulated Human into a Networked Virtual Environment"; the Postscript version of that document is online (at <ftp://taurus.cs.nps.navy.mil/>, in the directory pub/NPSNET_MOSAIC; believe it or not, the filename is "Insertion.of.an.Articulated.Human.into.a.Networked.VE.ps").

The paper attempts to deal with the people problem. It considers people to be dismounted infantry, and sends information about their configuration; small packets of information giving the rotation angles of each limb are sent as part of the PDU.

This works well, but is (by design) highly anthropomorphic. The packet format actually stores information about the rotation of each hip joint; it has no provision for body configurations other than human. In terms of our park example, the data structures they propose make no provision for a bird's wings or a squirrel's big fluffy tail. Treating birds and squirrels as "dismounted infantry" isn't really the best approach.

(This is not meant as a criticism of the work done on dismounted infantry; it's certainly better than the standard DIS approach, which would require us to treat each of the squirrel's legs as a tiny gun turret!)

Beyond Dead Reckoning

Dead reckoning by itself, even when supplemented by anthropomorphic joint information, isn't enough. Does that mean we should be looking for a completely different approach? Not necessarily.

After a little thought, it becomes clear that dead reckoning is just a special case of a more general idea: instead of sending location and orientation updates, we should be sending updates about an entity's behavior. In a sense, we're generalizing dead reckoning, and extending it to meet our needs.

The update messages an entity sends out will specify a "behavior" and some behavior parameters; the dead reckoning approach then becomes just one of a repertoire of behaviors, which might be arbitrarily complex. In the case of the dead reckoning behavior, the parameters are the initial time, initial location, and initial velocity vector (direction and magnitude).

In a VRML browser, these update messages would be sent as events to the behavior script for an object. The parameters of the behavior (as sent in the message) are passed to the function in the script which then implements that behavior.

The local events received by behavior scripts in a browser would be sent via the network to the master copy of an object, which would respond by sending back behavior messages. In effect, level 2 behavior would be moved off onto another host on the network; the network exists at the interface layer between level 2 and level 1.

A Brief Digression TCP/IP

As mentioned earlier, broadcasting update messages is a bad idea. In order to understand the alternatives, it's worth taking a quick look at the underlying network infrastructure that's available to us.

The Internet is based around a family of protocols called TCP/IP. It is described in detail in Chapter 1, but here's a quick review.

The IP (Internet Protocol) part is the lowest level; it handles addressing and routing. IP packets can be sent over Ethernet, fiber optics, telephone lines, radio links, tin cans and string, or any other medium that can move bits.

Above IP are two other protocols: TCP and UDP. TCP, the Transfer Control Protocol, provides a connection-oriented, reliable byte stream form of communication. Applications can use TCP to open a logical connection to another host on the Internet, send data down that connection, and receive data back. TCP handles the complex chores of breaking the stream of data into smaller pieces, doing checksums on the data to make sure it arrives intact, sorting the packets as they arrive to make sure they're in the correct order, requesting retransmission when needed, handling flow control, and lots more. TCP uses IP to actually move the packets to other hosts.

TCP is the protocol on which most Internet applications run. The Web uses HTTP (Hypertext Transport Protocol) which runs over TCP; when you click on a link, a connection is opened to another host using TCP, the URL is sent down the wire, the data is received back, and the connection is closed. Similar techniques are used for Finger, SMTP (electronic mail), and NNTP (USENET news). In all cases, the hard parts are handled transparently by TCP.

UDP, the User Datagram Protocol, is quite different from TCP. It provides a connectionless, unreliable way of sending "datagrams" from one host to another. The word "unreliable" in this context doesn't mean that UDP is somehow flakey or error-prone; it just means that there's no guarantee that a specific packet will arrive. However, if a packet does arrive, it will arrive intact. UDP is used by NFS (the Network File System), NTP (Network Time Protocol), and several others.

There are some important advantages in using UDP instead of TCP. The problem with TCP lies in its strengths. The fact that it does all kinds of flow control, error checking, and sequencing means that it has

a fair amount of overhead; this translates into lag, which is one of our enemies.

UDP packets are relatively "lightweight." Although they get lost occasionally, there'll never be any congestion; they won't clog things up, they'll just be discarded. Also, unlike TCP, UDP packets can be broadcast on subnets (since there's no "logical connection" involved).

However, because UDP is "unreliable," it requires the use of "stateless" protocols; in other words, each message must be complete and self-contained, and make no assumptions about previous messages having been received. NFS is an example of a stateless protocol; so is DIS.

DIS uses UDP broadcast packets to send its PDUs to other hosts. Each ESPDU has the complete state of the entity, and they're rebroadcast every few seconds (or more often, if the entity determines that more frequent updates are needed).

Why Broadcasting Doesn't Scale

The broadcast approach works for DIS, because it has its own dedicated network; unlike DOOM, there's no need to be concerned about the broadcast activity playing havoc with other users of the network.

However, the broadcast approach still doesn't scale. Even with dead reckoning, the problem of limited bandwidth is still there; as the number of entities grows larger, the slower-speed links become saturated. Even if bandwidth weren't an issue, the processing of updates for a large number of entities begins to exceed the capacity of the slower hosts, especially since they must also do all the rendering, input device processing, and so on.

The designers of DIS are well aware of these problems, and they're looking at solutions. The central idea, common to DIS and many other approaches as well, is the use of "update filtering."

The basic idea is again very simple. The virtual world is divided up into a large number of "cells," much like the way the cellular telephone system works. These cells are used as the basis for filtering update messages.

Each host participating in the simulation determines an Area of Interest (AOI), consisting of a number of cells within its range of vision.

The DIS work on the project is described in the document "Exploiting Reality with Multicast Groups" by Macendonia, Zyda, Pratt, Brutzman, and Barham. The Postscript version of the paper is online; the URL is <[ftp://taurus.cs.nps.navy.mil/pub/NPSNET_MOSAIC/macedonia_exploit.ps.Z](http://taurus.cs.nps.navy.mil/pub/NPSNET_MOSAIC/macedonia_exploit.ps.Z)>.

The paper uses an example where the cells are hexagonal, somewhat like a strategy board game or certain types of military simulation computer games. This hex grid is well-suited to military simulations, and is a closer model for circular Areas of Interest than a square grid would be.

As participants move around, cells will enter and leave their Area of Interest; at any given time, they're only receiving updates for cells they can see, resulting in a small, manageable number of updates. The combination of AOI filtering and dead reckoning produces significant bandwidth savings.

Now clearly, hexagonal grid cells are better-suited for military simulations than for real-world VR. Our cells (perhaps better described as "zones") wouldn't necessarily be hexagonal grid cells; they would

probably be defined by other criteria. For example, a virtual mansion might be divided up into a large number of rooms, each of which would be its own zone. Since no regular grid would be used, there would need to be some way to specify zone-to-zone visibility.

Other than that, the basic idea is sound; just as we extended the idea of dead reckoning to the more general concept of behavior, we're extending the idea of update filtering from hexagonal grid cells to arbitrarily shaped volumes of space.

Multicasting

Given that we want to filter messages, what mechanism should we use for doing so? The DIS system uses "multicasting," and that approach seems promising for more general distributed VR applications as well.

Multicasting is an attempt to have the best of two worlds: broadcasting and point-to-point links. The idea is that any given host, in addition to having its own Internet address, can belong to a number of "multicast groups." Each multicast group has its own special Internet address; IP addresses in a certain range have been set aside for this purpose. When any host sends a message to a multicast address, that message is sent to all the hosts that belong to that multicast group. In effect, it's like broadcasting to a subnet that spans continents.

Sending a multicast message is better than having to send point-to-point messages to every host in the simulation, so it's easier on the sender. Hosts that aren't in a particular multicast group will ignore the packet at a very low level, with minimal overhead; this makes it easier on every host on the network.

There are some problems with multicasting; the most important is that it's not universally implemented. Some systems have it, some don't. An effort is underway to link small pockets of multicast-capable machines to each other over the Internet; the result is a "multicast backbone" or MBONE. The MBONE system uses "tunneling"; it wraps the IP packets destined for a multicast address inside of another IP packet, which travels through the regular Internet from one MBONE subnet to another.

Multicasting and Zones

How does multicasting fit in with Area of Interest management? In the DIS system, each cell has its own multicast group address; in other words, there's a one-to-one correspondence between cells and multicast addresses.

As participants move around, they enter and leave cells; this corresponds to entering and leaving multicast groups. Since participants only receive messages for multicast groups they're in, the multicast system itself does our message filtering for us.

The implementation of multicasting for DIS also does away with the "keep-alive" messages, which (according to their figures) account for a high percentage of the total traffic. Instead, a "group leader" is chosen for every group (i.e., cell); the group leader is the simulation host which has been in the cell the longest. When a host first enters a cell, it sends a message to the multicast address for that cell asking the group leader to identify itself; the response is a message informing the newly arrived host where they might obtain the complete current set of Entity State PDUs for this zone. The newcomer then requests

and receives that set.

Another, simpler approach might be to send an "identify yourself" message to the multicast address; all the hosts would respond by sending their most recent update message, and perhaps (in the absence of any kind of "registry") a message containing the URI of a file containing more detailed information about them, including the VRML description of the object.

Multicasting is not the only approach to filtering, but it's a good one; it doesn't involve any higher-level functionality to do the filtering, and therefore imposes minimal overhead.

Open Questions About Zones

There are a number of questions to be answered relating to zones:

- How do we define a zone?
- How do we find out what zone we're in?
- How do we find out what zones we can see?
- How do we map zones to multicast addresses?

Clearly, some additional work is needed in these areas.

Why Not Just Use DIS?

It certainly sounds like DIS has addressed most of the issues we need to deal with. Why not just use DIS?

Well, as mentioned several times above, DIS is designed for a very specific application. Most of the PDUs are inappropriate for general VR use (e.g., even the most aggressive squirrels are unlikely to issue "munition resupply request" PDUs). The Entity State PDU is too specific, and doesn't generalize well to our concept of sending "behavior." It also includes a great deal of superfluous information, such as which army the squirrel belongs to. This makes the individual PDUs very big, which greatly (and needlessly) increases the bandwidth requirements. A leaner, simpler format is needed.

As a minimum, a message must contain the entity ID, a timestamp, and a behavior description. In fact, that may be all it needs to contain.

What We Can Get from DIS

DIS brings us several powerful ideas:

- Standard message formats
- Fully decentralized modelóno central server
- Dead reckoning (generalized to behavior and parameters)
- Areas of Interest (based not on cells but on "zones")
- Possible use of multicasting for filtering of update messages

Accessing Entity Information

One area that DIS doesn't address, which we must, is that of initially getting information about the entities in the virtual environment. We're not just dealing with tanks, planes, and projectiles here; we want to be able to have objects with arbitrary geometry (and other attributes) in our simulation.

DIS assumes everybody has the complete database before the simulation starts; we need something more dynamic. We do need a way to uniquely identify every entity in the simulation, and provide a way of accessing their geometric description (specified using VRML) as well as other information about them.

To Register or Not to Register?

Essentially, we need a way of generating short, unique identifiers for every entity in the simulation. One approach (suggested by Eric Van Hensbergen) would be to use the IP address of the entity's host, plus a short (16-bit) number to identify the entity on that host.

This avoids many of the problems of a central registry; there's not much overhead, since the messages would contain the originating IP address in any case. A mechanism would have to be established on each host for generating the per-entity sequence numbers, but that's a far easier problem to solve.

There would need to be some mechanism to map an entity ID to a URI giving more details about the entity. Perhaps it could be as simple as sending a request to the entity's host asking for more information about the entity, and receiving a response message back containing a URI.

About Time

It's important that all the systems remain synchronized; in other words, they must all agree as to the current simulation time. Existing protocols should be used wherever possible, and NTP (the Network Time Protocol) should be sufficient for this purpose.

Summary

It will be quite a while before any really solid standards begin to emerge for distributed VR and multiuser VRML worlds. There's going to have to be a period of experimentation, during which different approaches will be tried by various groups. Hopefully this chapter has given you some insight into some of the challenges that lie ahead.

From Here...

Discussing distributed VR in isolation from behavior is a bad idea; if you haven't already read [Chapter 28](#), then go back and do so now. If you've had enough technical jargon for the time being, then move on to [Chapter 30](#) for a bit of philosophy.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



30-The Zen of Virtual Reality

This is not the first book I've worked on that has been a collaborative effort. After my first book came out, I received e-mail from people asking me who wrote which chapters; to avoid having to reply to similar e-mail this time around, I'll put a breakdown at the end of this chapter. However, this particular chapter may come as a surprise to you; since it focuses on the philosophy of VR, it's a lot less "techie" than my previous ones.

This chapter has just one simple goal:

- To introduce you to the philosophy of virtual reality

In this chapter, I won't be mentioning bandwidth, polygon counts, or transformation matrices; instead, I'll don my philosopher's cap. Or perhaps touch the Philosopher's Stone, to transmute myself from base computer geek to gleaming, golden VR visionary.

The Orb of Agomotto

I've always loved comic books. When I was young, one of my favorite heroes was named Dr. Strange; he was Earth's Sorcerer Supreme, and he was in charge of fighting any supernatural threat to mankind.

One of the unique aspects of Dr. Strange was that he fought many of his greatest battles from the comfort of his own home. He would gaze into the Orb of Agomotto, a desktop-sized hemisphere that would allow him to see into mystical dimensions and spot trouble brewing. He would then enter a trance-like state and send his astral body into those other worlds to set things right. His spirit-self lacked substance, and was able to float and fly through those bizarre, surreal worlds; he could even pass, wraith-like, through solid objects. All this would take place while his physical body sat in the relative safety of his Sanctum Sanctorum.

Sound familiar?

The Orb of Agomotto has been replaced by a 16-inch color monitor, and instead of entering a trance state, one puts on a head-mounted display. Other than that, the similarities are striking; science and sorcery have indeed met. As Arthur C. Clarke put it, "Any sufficiently advanced technology is indistinguishable from magic."

Escape

As I sit here, the first November snow has just fallen. I look out the window, and see it swirling. It's like staring into a snow globe, or into the Orb of Agomotto. For a flickering moment, I transport myself to another place and time and begin a mini-adventure in my mind. And as I return to my computer and this manuscript, it dawns on me that VR and VRML will allow those mini-adventures to actually happen, on the "astral plane" of cyberspace.

This has been a dream for a long time. From Ray Bradbury's "The Veldt" in 1950 to the high-tech dreams of today's computer games, we've been striving to build imaginary worlds that we can disappear into.

What will we do when we get there? Well, one of the dreams has been to "crawl inside the computer" and have more intimate contact with the data that resides there.

Becoming One with the Data

Scientific Visualization. How's that for a dry subject? And yet, it's one of the most important uses for virtual reality.

Of course, it doesn't have to be boring. Taking columns of numbers and trying to give them meaning, striving to endow them with a kind of vitality and life that they sorely lack, can be exciting and challenging. It requires you to be part scientist, part computer programmer, and part artist; you need to understand the underlying reality of those numbers, and virtualize it. You must give it shape and form, color and movement.

We are visual creatures. Our perception of the world starts with sight, not numbers, and to really understand what's going on in a system, we need to see it and experience it first-hand. That's one of the things VR offers: it gives us a means of metamorphosis, a way to transform abstract data into something we can dive into and wander around in. By becoming one with a virtual universe, we become one with the data it contains.

Reality and Surreality

All virtual worlds have a certain surreal quality to them. There are no shadows. There is no dust. There is no dirt. It's not "reality" at all, but virtual reality—neither better nor worse than the real thing, but certainly different.

VRML and Environmental Art

I recently met someone who is involved in "environmental art." Don't let the name fool you; it doesn't have anything to do with ecology. Environmental art involves creating a three-dimensional scene, which people can wander around in. They don't look at the art, so much as move through it.

In the real world, environmental art is rare. It can be expensive to create. It requires a larger area than

most art forms, and as a result it must compete with other exhibits. Quite often, environmental pieces only have a short "run" before they're taken down to make room for other projects.

VRML is a tool for environmental art. By building the environmental piece in cyberspace, rather than real space, all the limitations are removed. People are still free to move through the art, but there are no constraints as to its size. There are also no limits as to what it might contain; whatever the artists can imagine (and model in 3D) they can include in the space. They're also not limited by what can "really" exist; in fact, they're not even limited by gravity or physics.

They're also not limited by time. Their exhibit can last forever, and can be placed on the Internet in order to reach an audience larger than any physical environmental art piece ever could.

Home, Home on the Homeworld

Artists aren't the only people who create art. Just as the World Wide Web has brought about a "democratization" of publishing, VRML will bring about a democratization of art. People will not just have a "home page"; they'll have a "homeworld." Each such homeworld will be unique, and each will be a piece of environmental art that expresses something about the person who created it.

You can learn a lot about someone by visiting his or her home. His or her choice of furnishings, lights, and taste in music and art, all speak volumes about the kind of person he or she is.

However, there's a problem. Everyone is taught how to read and how to write; creating text is not difficult for the average person. Creating art, on the other hand, is not something that people are used to doing.

Of course, this creates a new market; just as there are companies out there right now designing home pages, there'll be companies that specialize in building home worlds. They'll become adept at reading your personality and constructing a space that is uniquely your own, one that captures your essence in polygons and pixels.

Tools

The main obstacle to effective world-building right now is the lack of tools. That can be expected to change, and soon; there's suddenly a market for tools that are powerful, inexpensive, and simple to use.

It's true that it's a poor craftsman who blames his tools. On the other hand, even Da Vinci would have been limited if all he had was a six-pack of Crayolas. Better tools make for more effective expression, and better tools are on their way.

You shouldn't have to be a CAD expert to create a homeworld for yourself. You should be able to select from a catalog of "clip objects" (analogous to clip art), apply colors and textures, and assemble them to form the world inside your head.

Welcome to My Nightmare

Speaking of the world inside your head...

VRML is a tool for creating reality. Anything you can imagine, you should be able to build, and anything you can build, you can share with others. Your innermost thoughts, dreams, and nightmares can be given three-dimensional form. You can invite other people to share in your visions, to become part of your fantasies, to explore your world in order to discover who you are.

The world as seen in your mind's eye can be brought through the eyes and into the minds of people across the face of the globe. People you've never met can enter the darkest recesses of your psyche, provided that you've chosen to share those dark regions with them through VRML.

From bizarre alien landscapes to mystical demon-filled worlds, from beautiful Elysian fields to the depths of Tartarus, from the realism of a living room to the surrealism of a room that lives, VRML is the common language for the ultimate form of self-expression: creating reality itself.

The Creative Urge

World-building is not architecture. There's more to creating a world than just putting up some walls and dropping in some furniture. We want to be able to fill our worlds with creatures that move, and respond, and think, and live. We want to create life.

At first blush, this may sound rather disturbing. Surely we don't want to follow in Dr. Frankenstein's footsteps? Well, fortunately this kind of life-creation doesn't require us to stitch together body parts and wait for a thunderstorm. Instead we can simply learn to program, and discover the magic of state machines and genetic algorithms, of pattern recognition and machine learning. While the artificial intelligence community has fallen short of the lofty goals it once set for itself, there have been some remarkable achievements by these mind-makers, and we can benefit by learning their craft.

A World of Walter Mittys

VR isn't just about creating worlds, or creatures to populate them. It's about re-creating ourselves.

James Thurber wrote a wonderful story called "The Secret Life of Walter Mitty." In it, a shy and timid man leads a fantasy life full of action, adventure, intrigue, and danger. In his mind, he is everything that he is not in life. And so it will be with VR.

We will each have our avatars, our virtual selves, that will be our embodiment in cyberspace. The word "avatar" comes from Hindu, and refers to a manifestation of god on the earthly plane. In a sense, it's almost the opposite of the meaning that's intended: in VR, our avatars are manifestations of man on the heavenly plane.

Of course, you are indeed the god of your own private worlds; you are your own "word made flesh." After all, what is a god but a creator of worlds?

You are not limited to a single avatar. You can keep a repertoire of them on hand, and select one that

suits your mood or your task. You might have a "body wardrobe"óone body for every day, one for special occasions, and so on.

Just as we show one side of ourselves to our friends, another to our employer, another to our children, and yet another to strangers, we will have avatars appropriate to every occasion.

Your avatars allow you to become who you really are, or who you really want to be. They allow you to leave the constraints of your physical body and let other people see you as you see yourself, or how you would have them see you at that precise moment.

Theatre

All the world may indeed be a stage, but the virtual world is even more than that: it has no "fourth wall," and no backstage; it is an all-encompassing theatre of the mind, in which we all will play our parts, and write them as well. To quote Brenda Laurel, "In a theatrical view of human-computer activity, the stage is a virtual world [...] The technical magic that supports the representation is behind the scenes. Whether the magic is created by hardware, software, or wetware is of no consequence."

As an actor crafts his or her character, so too will we build personalities for ourselves to go with our avatars. We will be able to discover different facets of ourselves, and explore how they interact with the virtual world around us.

Masks

Some of my acting teachers over the years have been involved in mask work; the most recent have been Paulette Hallich and Glenys McQueen-Fuentes. Mask work has been part of theatre for thousands of years; there are a lot of theories behind mask, but the most basic is that it removes an actor's inhibitions. They are no longer on stage as themselves, but as the mask; the mask has a mind, an essence, a personality of its own. It has greater freedom than the actor, and greater license to do outrageous things. It's liberating, and powerful.

Actors who do a lot of mask work speak of the masks as being "inhabited," and when you first listen to them it sounds like they're talking about some bizarre form of demonic possession! In reality, they're talking about the character suggested by the mask itself; on some deep, subconscious level we all believe that one's face reflects one's essence, and so wearing a different face (and a different body) gives us an essence that is not entirely our own.

Everyone in a virtual environment is wearing a mask, adopting a character, playing a part. Our "real" selves are left behind, just as Dr. Strange left his physical body at home while his astral self went out exploring. We have the freedom to design our own masks, and the bodies to go with them.

Gathering in Cyberspace

In a virtual environment, you can be anyone, go anywhere, do anything. You can interact with people who you will never meet in the real world.

In a surprising way, this produces a kind of intimacy. I once was on a long flight (Toronto to San Jose), and got to know the person next to me. After some pleasantries, it became clear that we were not going to the same place; indeed, it was very unlikely we would ever meet again. Within an hour, she was sharing the most intimate details of her life, and I was giving her advice that I would never consider giving to even a close friend. The anonymity, the fact that we would never ever see each other again, or learn each others' "true names," was liberating.

That same freedom exists on the Net. Once we begin bringing more of ourselves into cyberspace, through VRML worlds and our personal avatars, that new kind of intimacy and closeness will grow.

The Net, the Web, cyberspace, the "astral plane"... whatever you call it, it's an increasingly important component of our lives. As more and more people gather there, and as better technology allows us greater freedom of expression, it will become a vibrant and exciting new environment. No longer will we be restricted to our small groups of nearby friends and neighbors; we will suddenly have an entire world of diversity and variety to explore.

The Oncoming Train

In the early days of the motion picture industry, a cinematographer shot footage of an oncoming train. It was a simple idea, but it had a surprising result. In the darkened theatre, people saw the train rushing toward them. They panicked. They screamed. They fainted.

Even though it was grainy, black-and-white, and completely silent, it was realistic enough to have a profound effect on the audience.

VR technology is at roughly the same point now that motion pictures were when that early film was shot. The images are crude, the resolution is low, there's often no sound and certainly no real sense of immersion. That will change in the next few years; already people are talking about head-mounted displays whose resolution approaches that of the human eye, and hardware accelerators that will allow us to render reality as never before. The oncoming train is about to arrive, and with it will come a whole new level of realism.

Who Wrote What

As I promised at the start of this chapter, here's a summary of who wrote what. Basically, all the "techie" chapters are mine: 3, 4, 6, 7, 11, 14, 16, 17, 19, 22, 26, 27, 28, 29, 32 and most of the appendixes (A, B, C, D, and E). And of course, I also wrote this chapter—a change of pace for me, and one which I hope you enjoyed.

All Aboard!

In this chapter, I've touched on some of the philosophical underpinnings that lie just beneath the surface of VR technology. If you remember nothing else from what you've just read, bear in mind that VR and VRML are about to change the nature of society and of human communication. Get on board; it's going to be an exciting ride!



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



31-Architecture, Design, and VRML

In the previous chapter, Bernie outlined the philosophy of virtual reality. In this chapter, we will look at the philosophies underlying architecture and design, and how they relate to designing VRML environments. This chapter will not deal with specific architectural and design features such as column types or decorations. Instead, it looks at incorporating theories about architecture and design. In other words, we will discuss the ideas behind a design rather than the design itself.

What Is Architecture and Design?

Architecture, as defined by the dictionary, is the art and science of building. Design is related to the construction of an object. But whether you are talking about architecture or design, you are going through a process of describing why something looks the ways it does. Indeed for all intents and purposes, architecture and design are one and the same.

Behind both architecture and design is a philosophy and an idea. In *Form, Function and Design*, Paul Jacques Grillo stated that design is a form of poetry in that it associates forms into new meanings. When you look at any building or object, you can "read" it in the same way that you read a novel or poem. It has a narrative, it has a structure, and it expresses the ideas of those who created it.

Le Corbusier

Le Corbusier (whose real name was Charles-Edouard Jeanneret) was one of the most influential writers of architecture. He was both a practicing archi-tect and a theorist who saw a need to break away from the classically based schools of architecture. His book, *Towards a New Architecture*, helped to define the modern movement. It expresses not so much a technique, but a guiding philosophy and principles to design. He wrote that architecture is "the masterly, correct and magnificent play of masses brought together in light" (*Towards a New Architecture*, 1927). And it is this play of mass and light that creates and expresses the ideas of the designer/architect. The trick is to make that play "correct and magnificent."

In *Towards a New Architecture*, Le Corbusier states that architecture is "a thing of art, a phenomenon of emotions, lying outside questions of construction and beyond them." How we arrange the design

establishes our intentions and ideas. And we manifest these intentions and ideas by using mass and surface, both of which are determined by "the Plan." He goes on to construct these three elements into "three reminders to architects": things that architects must always keep in mind when designing a building.

The first reminder is Mass. Our eyes are designed to let us see forms in light. And we give a building its mass by putting forms together: it can be a simple mass of cubes, or it can be symphony of different shapes. Moreover, the most beautiful forms are the primary ones—cubes, cones, spheres, cylinders, and pyramids—because they can be clearly appreciated.

The second reminder is Surface. Le Corbusier notes that a mass is "enveloped" in its surface, with each surface divided according to the lines of the mass. It is surface that gives a mass its individuality.

The third reminder is the Plan. The Plan holds the "essence of sensation." It is what gives a design its order and willfulness. It is, according to Le Corbusier, the "generator."

With these principles in mind, we can turn to the function of the architect. For Le Corbusier, the product of the architect is more than simply design. He writes:

"The Architect, by his arrangement of forms, realizes an order which is a pure creation of his spirit; by forms and shapes he affects our senses to an acute degree and provokes plastic emotions; by the relationships which he creates he wakes profound echoes in us, he gives us the measure of an order which we feel to be in accordance with that of our world, he determines the various movements of our heart and our understanding; it is there that we experience the sense of beauty." (Towards a New Architecture, p. 7)

Those "plastic emotions" do not mean artificial ones; plastic in this sense means malleable or changeable. When we see the products of great architects like Frank Lloyd Wright, Walter Gropius, and I. M. Pei, we undergo a process very much like the one we experience when we read the works of William Shakespeare, Yukio Mishima, and Toni Morrison. We do not feel only one emotion; our emotions change as we explore the design.

And it is affecting our users' senses "to an acute degree" and provoking these "plastic emotions" that should be our goals as architects in a VRML world.

Designing for a VRML World

If we take Le Corbusier's ideas as our starting point, we can begin to construct some guiding principles for design in a VRML world. But whereas he was constrained by the physical limitations of the real world—construction material, climate, location, etc.—designing for a virtual world gives us a greater degree of flexibility.

While designing a VRML environment, we need to think about the following:

- Design and function
- Originality
- Proportion

Design and Function

Design is form, and good design is never divorced from function.

All good design is dynamic: that is, it makes us aware of the life and action that will take place in a building or with an object. Indeed, a good design will makes us aware of the purpose of the building, room, and/or object before us. If we come across a design that does not make its intentions clear, we are left confused and unable to function within that design. How many times have you looked at an object and puzzled over what it does? How often have you become lost in a building because you could not figure out how you were supposed to move through it?

For a design to be effective, it must make us become aware of the life and action that will take place within it. When you go to Philadelphia, for example, you can visit where Benjamin Franklin lived and worked. You can go through the building where he had his printing press. But when you go through the courtyard, you see simple, steel structures that represent buildings that once stood there. These structures are like three dimensional outlines: they give a sense of a building without the building having to be there. The steel is a stark, contemporary contrast to the brick and mortar of the historical buildings surrounding these outlines.

Now these missing buildings could have been reconstructed. But the designers of the museum decided to give us only their "ghosts": ethereal representation of what once was. The steelóespecially on a sunny dayónot only reflects light; it reflects the fact that one is standing in the midst of history.

This idea, as expressed by mass and light, is both correct and magnificent in that the design of these structures gives us an understanding of history in a historical site. The Franklin historical site does not try to pretend that you have traveled back in time. The staff is not dressed in historical costumes; the site does not look exactly like it did over 200 years ago. The steel structures make you acutely aware that you are in the present. What you gain from this is an awareness of this site's place in history. You gain the realization that the importance of this place does not lie in what it was like in the past, but in its importance in relation to the present. Thus the design of the Franklin museum makes it the ideal historical site: its design does not simply present history; it makes you aware of the function of history.

Originality

While there will be times when we want to recreate the designs of the past, our primary goal in the VRML world is to create new designs for a virtual world. What makes a new building interesting and exciting is the way it breaks new ground. How many times have you gone to a "new" mall or cafe only to be disappointed to see more of the same old thing? But if you think about when that design did seem exciting, you will probably discover that it was when you saw it for the first time.

In a VRML world, we have the opportunity to do so many new things. Gravity will no longer be a factor. People can fly through a VRML space instead of simply walking through it. Walls, floors, and ceilings no longer have to be connected to each other. We do not have to have light fixtures to have light. Our designs should take these factors into consideration.

Grillo states that we must avoid borrowing any element from another design unless we intend to use it for the same function. And yet we will see people trying to recreate objects and environments from the real

world. This may be useful as a way to connect the virtual world to the real one, but it is not going to be the main attraction of the virtual universe. Why go to a virtual mall that looks like the mall that is just down the street? If we want to entice people to our VRML worlds, we must break free from the pull of the past and create something new.

Antiperspective Three-Dimensionality

One way of making something more original is to think of 3D without the trappings of perspective. In *The Modern Language of Architecture*, Bruno Zevi calls this style of thinking antiperspective three-dimensionality.

According to Zevi, the problem with perspective is that it rarely performs what it is supposed to do. Perspective is a drawing technique for representing three-dimensional objects in two dimensions. It should, therefore, give us an awareness of three dimensions in design. But what has happened is that designers have created three-dimensional objects with two-dimensional perspectives.

For example, let's say we are building three cubic structures in an inverted u-shape. Classical schools of architecture would position these buildings at 90 degree angles to each other (see Fig. 31.1).

[Fig. 31.1](#)

The rear building positioned at a ninety-degree angle to the other two. (a) Top view. (b) Perspective.

But when we look at the rear building, we do not see a three-dimensional object. Instead, the three-dimensional object looks flat: that is, it has taken on a two-dimensional appearance. If, however, we shift the angle of the rear building, it becomes a three-dimensional object (see Fig. 31.2).

We do the same with objects in a VRML environment. For example, instead of setting walls perpendicular to each other, we can set them at different angles to create a better sense of the three-dimensional nature of our environment (see Fig. 31.3).

[Fig. 31.2](#)

The rear building positioned at a different angle to the other two. (a) Top view. (b) Perspective.

[Fig. 31.3](#)

Using different angles to create a greater sense of 3D.

Four-Dimensional Decomposition

Another element we can use to enhance the originality of our design is to think about what Zevi calls four-dimensional decomposition. He notes that the traditional joining of four walls, a floor and a ceiling creates a "perspective block." He likens it to a coffin: confining the occupant in an enclosed space.

A way of freeing ourselves from this perspective block is to "eliminate the third dimension by decomposing the box": that is, by breaking it up into panels (see Fig. 31.4). Thus our rooms are no longer left to "a cubic void." Instead, they are fluid; they flow in a continuum. As Zevi puts it, "The static quality of classicism is replaced by a dynamic vision, with the element of time added or, if you will, with

a fourth dimension" (The Modern Language of Architecture, 31).

Fig. 31.4

An example of four-dimensional decomposition.

Proportion

Another element in any design is proportion. However, it is open to question whether proportion is important to good design.

For Grillo, good design needs good proportion. He believes that a keen sense of proportion is "one of the most precious qualities of a good designer." For him, good proportion has rhythm and harmony. There is a recurrence of certain patterns that combines an appreciation of form, rhythm, and scale which, when put together, creates harmony.

In contrast, Zevi sees proportion as "another tumor that needs to be cut out." For him, the rhythms of proportion that establish a binding relationship to different parts of a building are artificial: a "neurotic longing for 'synthesis'." If different parts of a building have different messages, why unify them through proportion and reduce the various messages to only one? As Zevi puts it, "Proportion freezes the vital process and masks falsity and waste."

So what should we do about proportion in our VRML world? The key is flexibility. Since proportion unifies the ideas and messages we wish to convey, why not use it when we want to do so? But remember that a VRML world does not have to be made up of a single room and, conversely, a single message. If we want to convey a multitude of ideas, we need not be as concerned about proportion.

A Need for the New

Of course, everything I have outlined in this chapter is my own opinion. Just as Bernie's vision of VR can be subjected to other viewpoints, my views about design are vulnerable to other philosophies and approaches. Some people will demand symmetry instead of an anti-perspective three-dimensionality. Some will want to copy from the real world rather than use the virtual aspects of virtual reality. Deconstructionalists and Postmodernists will say that I have not done enough to break free from the artificial constraints of order.

But in many ways, dissonant visions are the keys to VRML's success. In VRML: Browsing and Building Cyberspace, Mark Pesce notes that VRML is so new that there is only "a tiny body of work" for us to look at. For him, there must be and will be more experimentation with VRML.

I too think that experimentation will be a key to the success of VRML. But we need the freedom to discover mistakes as well as successes. That is the nature of experimentation. And I don't believe that VRML will fail because of bad content design. I am confident that it is robust enough to survive the horrors in design that will inevitably appear. We need only to look at how HTML has not only survived but flourished with both good and bad design.

When we look at the real world, we get a glimpse of how the VRML universe will unfold. I visited Prague a few years ago and what struck me were the variety of styles that existed within a few blocks of

each other. I could visit a gothic cathedral and then pop in to a cafe with an Art Deco design. Sitting beside the 18th-century opera house where Mozart's Don Giovanni was first performed was an ultra-modern arts center built during the Soviet occupation. It was clear that as each style became obsolete, there were always several new designs vying to take its place. Sometimes the old style would be torn down to make way for the new. The buildings using the old styles survived if they maintained some kind of significance. But the process of trying something new was constant.

This is the way Cyberspace will be built. We will build our virtual worlds and open them for users to explore. Some of our worlds will endure. Those that no longer have any relevance will be torn down to make way for the new. And we must always encourage the new or we will never be able to go beyond the mistakes of the past. It is as Shakespeare reminds us:

If there be nothing new, but that which is

Hath been before, how are our brains beguiled,

Which, laboring for invention, bear amiss

The second burden of a former child!



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



32-Writing VRML Software

Well, it's been a long road. If you've made it this far, you must truly be a VRML fanatic; join the club!

If you also happen to be a C programmer, then this chapter is a must-read; if you're not, go learn C and come back later.

In this chapter, we're going to

- Write a translation program that produces VRML
- Write a "world generator" that produces VRML
- Examine some of the issues in writing a VRML browser

I'm going to assume you already know how to program in C. If you don't, there are a lot of excellent books on the market; check out your local library or computer bookstore.

NFF2VRMLóThe Inside Story

We took a look at the NFF2VRML translator back in Chapter 26. Now it's time to open it up and look at its guts. The guts aren't very pretty; I literally wrote this program in an afternoon, and there are many things it doesn't yet support. Of course, since you have the source code, you should be able to add whatever additional features you may require. You can also use it as a skeleton for writing translation programs for other formats.

The NFF File Format

Before we can write a translation program, we need to understand the format of its input and output files. The output file is VRML; you already know everything there is to know about it from having read the rest of this book. The input format is NFF (the Sense8 one, not the Eric Haines one).

Like VRML (and most other object file formats), NFF is pure ASCII. That makes our job easier, since we don't have to worry about machine word lengths, byte ordering, or any of the other nasties you encounter when reading binary files that may have been produced on a different kind of computer system.

Everything after a pair of slashes ("/") on any line of an NFF file is considered to be a comment, and ignored. This is similar to the "#" character in a VRML file.

Unlike VRML, an NFF file is line-oriented; line boundaries are important. The first line in a valid NFF file should be "nff." After that, the version number is specified on a line containing (for example) "version 2.0"; this allows programs that parse NFF files (like ours) to recognize that the file format is different from what they were expecting.

Following this are some optional specifications for the camera location and the point the camera is looking at; these are specified using "viewloc" and "viewdir" lines.

Next comes the object name on a line by itself (possibly followed by "shading=y" or "shading=n," to indicate whether the object should be shaded or not).

After all this preamble, we get to the meat of the data itself: the lists of vertices and polygons. The first line after the object name is a single number, which gives the number of vertices in the object. This is followed by that number of lines, each of which contains the X, Y, and Z values of the vertex. The NFF file supports normals, in a true per-vertex form (not to be confused with VRML's idea of PER_VERTEX). If a particular vertex has a normal, then the X, Y, and Z coordinates are followed by the word "normal" and then the X, Y, and Z components of the normal vector. If just the letter "n" is specified, then a normal should be computed automatically for the vertex.

The next item in the file (following the vertex list) is a number telling us how many polygons are to follow. After this come the polygon lines; there's quite a bit of information for each polygon, but there are only three things we care about. The first number on the line is a count of the number of points in the polygon; for example, an octagon would have the number 8, and a triangle would have the number 3. After this is a series of numbers that indicates which vertices from the vertex array are to be used in this polygon. Finally, there's the polygon's color.

By now, the similarity to VRML should be clear; the list of vertices is exactly what you would find in the Coordinate3 node, the normals are what you would find in the Normal node, and the numbers on each polygon line are the same numbers you would find in the coordIndex field of the IndexedFaceSet node.

The color value is generally a three-digit hex number, with a C-style "0x" in front of it. Each digit is a four-bit value giving the intensity of one of the color components (Red, Green, or Blue). For example, 0x5BF would have a red value of 5, a blue value of 11 (the decimal equivalent of B hex), and a green value of 15 (the decimal equivalent of F hex).

There's other information on a polygon line, but the only remaining thing that we're going to worry about is the keyword "both," which indicates that the polygon should be visible from both sides.

The following listing shows a typical NFF file, in this case a tree produced by InWorld's TreeTool.

nff

version 2.0

//óóóóóóóóóó

viewpos 0.0 -50.0 -250.0

viewdir 0.0 0.0 1.0

None

//Total Vertexes

//óóóóóóóó

408

//All Vertexes

//óóóóóóóó

-42.6515388 -112.7189484 -8.4177876 N

-44.2316589 -112.9305420 -18.2898922 N

-36.7058830 -115.0833130 -24.5130959 N

-27.5999851 -117.0244904 -20.8641949 N

-26.0198669 -116.8128967 -10.9920902 N

-33.5456429 -114.6601257 -4.7688866 N

-38.4551849 -94.0324860 -13.2722073 N

-39.8372307 -94.2710266 -13.8371801 N

-35.8305359 -114.9881439 -14.0817490 N

-35.0013084 -114.8450165 -13.7427654 N

-37.2805634 -93.8384476 -14.2042780 N

-38.4551849 -94.0324860 -13.2722073 N

-35.0013084 -114.8450165 -13.7427654 N

-34.2965355 -114.7285919 -14.3020077 N

-37.4879837 -93.8829575 -15.7013216 N

-37.2805634 -93.8384476 -14.2042780 N

-34.2965355 -114.7285919 -14.3020077 N

-34.4209900 -114.7552948 -15.2002344 N

-38.8700256 -94.1215057 -16.2662945 N

-37.4879837 -93.8829575 -15.7013216 N

-34.4209900 -114.7552948 -15.2002344 N

-35.2502136 -114.8984299 -15.5392179 N

-40.0446510 -94.3155441 -15.3342237 N

-38.8700256 -94.1215057 -16.2662945 N

-35.2502136 -114.8984299 -15.5392179 N

-35.9549866 -115.0148468 -14.9799757 N

-39.8372307 -94.2710266 -13.8371801 N

-40.0446510 -94.3155441 -15.3342237 N

-35.9549866 -115.0148468 -14.9799757 N

-35.8305359 -114.9881439 -14.0817490 N

-43.4737320 -115.4475327 -15.0166225 N

-53.0794563 -112.8556213 -16.0226345 N

-56.3105659 -108.8711853 -24.6065826 N

-49.9359512 -107.4786453 -32.1845169 N

-40.3302269 -110.0705566 -31.1785069 N

-37.0991173 -114.0549927 -22.5945587 N

-37.3116837 -94.7495575 -14.6754007 N

-38.2979927 -94.8721008 -13.5359612 N

-46.4860687 -111.9401627 -22.8605976 N

-45.8942871 -111.8666306 -23.5442619 N

-37.6763000 -93.9544449 -15.9086885 N

-37.3116837 -94.7495575 -14.6754007 N

-45.8942871 -111.8666306 -23.5442619 N

-46.1130562 -111.3895645 -24.2842331 N

-39.0272217 -93.2818756 -16.0025368 N

-37.6763000 -93.9544449 -15.9086885 N

-46.1130562 -111.3895645 -24.2842331 N

-46.9236069 -110.9860229 -24.3405418 N

-40.0135307 -93.4044266 -14.8630981 N

-39.0272217 -93.2818756 -16.0025368 N

-46.9236069 -110.9860229 -24.3405418 N

-47.5153961 -111.0595551 -23.6568794 N

-39.6489143 -94.1995392 -13.6298103 N

-40.0135307 -93.4044266 -14.8630981 N

-47.5153961 -111.0595551 -23.6568794 N

-47.2966232 -111.5366211 -22.9169064 N

-38.2979927 -94.8721008 -13.5359612 N

-39.6489143 -94.1995392 -13.6298103 N

-47.2966232 -111.5366211 -22.9169064 N

-46.4860687 -111.9401627 -22.8605976 N

-21.0258598 -78.5839386 -0.4750996 N

-23.0190811 -77.2819214 0.3508730 N

-39.7689743 -94.9180374 -13.8068161 N

-38.5730400 -95.6992416 -14.3023996 N

-19.9515648 -77.9830627 -2.6740174 N

-21.0258598 -78.5839386 -0.4750996 N

-38.5730400 -95.6992416 -14.3023996 N

-37.9284630 -95.3387222 -15.6217499 N

-20.8704929 -76.0801697 -4.0469632 N

-19.9515648 -77.9830627 -2.6740174 N

-37.9284630 -95.3387222 -15.6217499 N

-38.4798203 -94.1969833 -16.4455185 N

-22.8637142 -74.7781525 -3.2209916 N

-20.8704929 -76.0801697 -4.0469632 N

-38.4798203 -94.1969833 -16.4455185 N

-39.6757507 -93.4157791 -15.9499350 N

-23.9380074 -75.3790283 -1.0220737 N

-22.8637142 -74.7781525 -3.2209916 N

-39.6757507 -93.4157791 -15.9499350 N

-40.3203278 -93.7763062 -14.6305847 N

-23.0190811 -77.2819214 0.3508730 N

-23.9380074 -75.3790283 -1.0220737 N

-40.3203278 -93.7763062 -14.6305847 N

-39.7689743 -94.9180374 -13.8068161 N

-34.1370506 -120.5217361 -15.5466824 N

-39.5248947 -120.2066498 -23.9652252 N

-35.1472893 -117.7508011 -32.6142349 N

-25.3818359 -115.6100540 -32.8447037 N

-19.9939899 -115.9251404 -24.4261627 N

-24.3715973 -118.3809891 -15.7771521 N

-29.7412949 -100.5763855 -12.3066597 N

-31.2516060 -100.6470947 -12.2964773 N

-30.1892338 -118.4875412 -23.5170860 N

-29.2830467 -118.4451141 -23.5231972 N

-29.0249767 -99.8736420 -13.4376717 N

-29.7412949 -100.5763855 -12.3066597 N

-29.2830467 -118.4451141 -23.5231972 N

-28.8532562 -118.0234680 -24.2018032 N

-29.8189697 -99.2416229 -14.5585022 N

-29.0249767 -99.8736420 -13.4376717 N

-28.8532562 -118.0234680 -24.2018032 N

-29.3296509 -117.6442490 -24.8743019 N

-31.3292809 -99.3123322 -14.5483198 N

-29.8189697 -99.2416229 -14.5585022 N

-29.3296509 -117.6442490 -24.8743019 N

-30.2358379 -117.6866760 -24.8681927 N

-32.0456009 -100.0150757 -13.4173069 N

-31.3292809 -99.3123322 -14.5483198 N

-30.2358379 -117.6866760 -24.8681927 N

-30.6656303 -118.1083221 -24.1895847 N

-31.2516060 -100.6470947 -12.2964773 N

-32.0456009 -100.0150757 -13.4173069 N

-30.6656303 -118.1083221 -24.1895847 N

-30.1892338 -118.4875412 -23.5170860 N

-31.0248489 -120.5793839 -14.4746866 N

-39.3199196 -117.8232040 -19.3321838 N

-38.8585510 -115.3513412 -29.0108738 N

-30.1021080 -115.6356583 -33.8320694 N

-21.8070374 -118.3918381 -28.9745712 N

-22.2684078 -120.8637009 -19.2958813 N

-29.2277622 -100.3319473 -12.7745743 N

-30.5390873 -100.7131805 -12.1255493 N

-30.5657578 -118.5688171 -23.3722172 N

-29.7789631 -118.3400803 -23.7616310 N

-29.2239647 -99.5631256 -14.0765104 N

-29.2277622 -100.3319473 -12.7745743 N

-29.7789631 -118.3400803 -23.7616310 N

-29.7766838 -117.8787842 -24.5427933 N

-30.5314903 -99.1755295 -14.7294216 N

-29.2239647 -99.5631256 -14.0765104 N

-29.7766838 -117.8787842 -24.5427933 N

-30.5611992 -117.6462250 -24.9345398 N

-31.8428135 -99.5567627 -14.0803967 N

-30.5314903 -99.1755295 -14.7294216 N

-30.5611992 -117.6462250 -24.9345398 N

-31.3479939 -117.8749695 -24.5451241 N

-31.8466129 -100.3255920 -12.7784615 N

-31.8428135 -99.5567627 -14.0803967 N

-31.3479939 -117.8749695 -24.5451241 N

-31.3502731 -118.3362656 -23.7639637 N

-30.5390873 -100.7131805 -12.1255493 N

-31.8466129 -100.3255920 -12.7784615 N

-31.3502731 -118.3362656 -23.7639637 N

-30.5657578 -118.5688171 -23.3722172 N

-20.4144096 -77.9971466 -0.3393202 N

-22.7724075 -77.4897232 0.3905926 N

-31.2691555 -101.0721588 -12.4041595 N

-29.8543587 -101.3766174 -12.8421078 N

-19.5867863 -77.1884766 -2.5779552 N

-20.4144096 -77.9971466 -0.3393202 N

-29.8543587 -101.3766174 -12.8421078 N

-29.3577843 -100.8914108 -14.1852884 N

-21.1171608 -75.8723755 -4.0866766 N

-19.5867863 -77.1884766 -2.5779552 N

-29.3577843 -100.8914108 -14.1852884 N

-30.2760086 -100.1017456 -15.0905218 N

-23.4751568 -75.3649521 -3.3567638 N

-21.1171608 -75.8723755 -4.0866766 N

-30.2760086 -100.1017456 -15.0905218 N

-31.6908054 -99.7972946 -14.6525736 N

-24.3027802 -76.1736298 -1.1181288 N

-23.4751568 -75.3649521 -3.3567638 N
-31.6908054 -99.7972946 -14.6525736 N
-32.1873779 -100.2825012 -13.3093929 N
-22.7724075 -77.4897232 0.3905926 N
-24.3027802 -76.1736298 -1.1181288 N
-32.1873779 -100.2825012 -13.3093929 N
-31.2691555 -101.0721588 -12.4041595 N
3.2248201 -50.4195023 0.9447946 N
0.8513327 -48.8067055 4.0116234 N
-22.1969681 -78.2257385 0.4946769 N
-20.7728748 -79.1934204 -1.3454205 N
2.3734884 -49.5127983 -3.0668287 N
3.2248201 -50.4195023 0.9447946 N
-20.7728748 -79.1934204 -1.3454205 N
-21.2836723 -78.6493912 -3.7523947 N
-0.8513308 -46.9932938 -4.0116234 N
2.3734884 -49.5127983 -3.0668287 N
-21.2836723 -78.6493912 -3.7523947 N
-23.2185650 -77.1376877 -4.3192711 N
-3.2248192 -45.3804932 -0.9447945 N
-0.8513308 -46.9932938 -4.0116234 N
-23.2185650 -77.1376877 -4.3192711 N
-24.6426582 -76.1700134 -2.4791739 N
-2.3734875 -46.2872009 3.0668287 N
-3.2248192 -45.3804932 -0.9447945 N
-24.6426582 -76.1700134 -2.4791739 N
-24.1318588 -76.7140350 -0.0721999 N
0.8513327 -48.8067055 4.0116234 N

-2.3734875 -46.2872009 3.0668287 N
-24.1318588 -76.7140350 -0.0721999 N
-22.1969681 -78.2257385 0.4946769 N
5.5405164 -115.7872772 47.2944183 N
-0.4892600 -110.2509460 53.0381775 N
-9.1180182 -106.7931747 49.3519440 N
-11.7170010 -108.8717346 39.9219437 N
-5.6872244 -114.4080658 34.1781845 N
2.9415340 -117.8658371 37.8644180 N
11.5753965 -98.4262924 37.3703270 N
12.0363140 -98.3494644 38.8083115 N
-2.4133034 -112.8623276 43.8972588 N
-2.6898537 -112.9084244 43.0344696 N
10.4504986 -97.5382462 36.8885345 N
11.5753965 -98.4262924 37.3703270 N
-2.6898537 -112.9084244 43.0344696 N
-3.3647928 -112.3756027 42.7453918 N
9.7865171 -96.5733795 37.8447227 N
10.4504986 -97.5382462 36.8885345 N
-3.3647928 -112.3756027 42.7453918 N
-3.7631812 -111.7966766 43.3191071 N
10.2474346 -96.4965515 39.2827072 N
9.7865171 -96.5733795 37.8447227 N
-3.7631812 -111.7966766 43.3191071 N
-3.4866309 -111.7505798 44.1818962 N
11.3723335 -97.3845978 39.7644997 N
10.2474346 -96.4965515 39.2827072 N
-3.4866309 -111.7505798 44.1818962 N

-2.8116918 -112.2834091 44.4709740 N

12.0363140 -98.3494644 38.8083115 N

11.3723335 -97.3845978 39.7644997 N

-2.8116918 -112.2834091 44.4709740 N

-2.4133034 -112.8623276 43.8972588 N

23.3476677 -105.7027969 55.3202019 N

15.7969303 -100.4295883 59.2163124 N

6.5442820 -104.2226486 59.2515335 N

4.8423719 -113.2889023 55.3906441 N

12.3931103 -118.5621109 51.4945335 N

21.6457596 -114.7690506 51.4593124 N

12.0097942 -98.2020950 37.5977402 N

12.3385134 -96.9685822 38.4080162 N

14.9512796 -109.2001419 55.4043198 N

14.7540474 -109.9402542 54.9181595 N

10.5826960 -98.6949310 37.5162468 N

12.0097942 -98.2020950 37.5977402 N

14.7540474 -109.9402542 54.9181595 N

13.8977890 -110.2359543 54.8692627 N

9.4843168 -97.9542694 38.2450256 N

10.5826960 -98.6949310 37.5162468 N

13.8977890 -110.2359543 54.8692627 N

13.2387609 -109.7915573 55.3065262 N

9.8130360 -96.7207565 39.0553017 N

9.4843168 -97.9542694 38.2450256 N

13.2387609 -109.7915573 55.3065262 N

13.4359932 -109.0514526 55.7926941 N

11.2401342 -96.2279205 39.1367950 N

9.8130360 -96.7207565 39.0553017 N
13.4359932 -109.0514526 55.7926941 N
14.2922516 -108.7557449 55.8415909 N
12.3385134 -96.9685822 38.4080162 N
11.2401342 -96.2279205 39.1367950 N
14.2922516 -108.7557449 55.8415909 N
14.9512796 -109.2001419 55.4043198 N
8.6802378 -76.3112793 20.9165840 N
8.8179770 -74.7943268 22.9241467 N
12.3629818 -97.4835587 39.1926270 N
12.2803383 -98.3937302 37.9880905 N
6.4595704 -77.2736130 20.2144814 N
8.6802378 -76.3112793 20.9165840 N
12.2803383 -98.3937302 37.9880905 N
10.9479389 -98.9711304 37.5668297 N
4.3766422 -76.7190018 21.5199375 N
6.4595704 -77.2736130 20.2144814 N
10.9479389 -98.9711304 37.5668297 N
9.6981812 -98.6383591 38.3501053 N
4.5143814 -75.2020569 23.5274982 N
4.3766422 -76.7190018 21.5199375 N
9.6981812 -98.6383591 38.3501053 N
9.7808247 -97.7281952 39.5546417 N
6.7350488 -74.2397156 24.2296028 N
4.5143814 -75.2020569 23.5274982 N
9.7808247 -97.7281952 39.5546417 N
11.1132259 -97.1507950 39.9759026 N
8.8179770 -74.7943268 22.9241467 N

6.7350488 -74.2397156 24.2296028 N
11.1132259 -97.1507950 39.9759026 N
12.3629818 -97.4835587 39.1926270 N
13.7910271 -120.7351608 20.6695518 N
4.1971264 -121.2534485 23.4423580 N
-2.4965649 -114.9624786 19.4903717 N
0.4036446 -108.1532211 12.7655802 N
9.9975452 -107.6349335 9.9927731 N
16.6912365 -113.9259033 13.9447584 N
16.9100456 -100.9339142 29.6950741 N
16.1316986 -101.5491638 30.8360329 N
7.4597278 -115.1209641 17.2009525 N
7.9267359 -114.7518158 16.5163765 N
16.3060589 -99.8059616 28.8894272 N
16.9100456 -100.9339142 29.6950741 N
7.9267359 -114.7518158 16.5163765 N
7.5643444 -114.0750427 16.0329895 N
14.9237251 -99.2932587 29.2247391 N
16.3060589 -99.8059616 28.8894272 N
7.5643444 -114.0750427 16.0329895 N
6.7349443 -113.7674179 16.2341766 N
14.1453781 -99.9085083 30.3656998 N
14.9237251 -99.2932587 29.2247391 N
6.7349443 -113.7674179 16.2341766 N
6.2679362 -114.1365662 16.9187527 N
14.7493649 -101.0364609 31.1713448 N
14.1453781 -99.9085083 30.3656998 N
6.2679362 -114.1365662 16.9187527 N

6.6303277 -114.8133392 17.4021416 N

16.1316986 -101.5491638 30.8360329 N

14.7493649 -101.0364609 31.1713448 N

6.6303277 -114.8133392 17.4021416 N

7.4597278 -115.1209641 17.2009525 N

21.5949135 -126.3909836 32.7611084 N

14.6334457 -124.8420792 39.7710228 N

6.2791796 -119.8366394 37.5012550 N

4.8863802 -116.3801041 28.2215748 N

11.8478470 -117.9290085 21.2116623 N

20.2021141 -122.9344482 23.4814301 N

16.5942993 -100.5609360 28.9678383 N

16.9781971 -100.5706635 30.4302578 N

14.1109381 -121.4752121 30.7312660 N

13.8806000 -121.4693756 29.8538132 N

15.1438131 -100.4114914 28.5679646 N

16.5942993 -100.5609360 28.9678383 N

13.8806000 -121.4693756 29.8538132 N

13.0103073 -121.3797150 29.6138916 N

14.0772228 -100.2717743 29.6305141 N

15.1438131 -100.4114914 28.5679646 N

13.0103073 -121.3797150 29.6138916 N

12.3703537 -121.2958755 30.2514191 N

14.4611206 -100.2814941 31.0929337 N

14.0772228 -100.2717743 29.6305141 N

12.3703537 -121.2958755 30.2514191 N

12.6006918 -121.3017120 31.1288719 N

15.9116077 -100.4309387 31.4928055 N

14.4611206 -100.2814941 31.0929337 N

12.6006918 -121.3017120 31.1288719 N

13.4709845 -121.3913803 31.3687954 N

16.9781971 -100.5706635 30.4302578 N

15.9116077 -100.4309387 31.4928055 N

13.4709845 -121.3913803 31.3687954 N

14.1109381 -121.4752121 30.7312660 N

8.6281805 -75.4858246 20.7548676 N

8.6908817 -74.7048645 23.1499844 N

17.0305347 -100.4714279 30.8028393 N

16.9929161 -100.9400024 29.3657684 N

6.5346084 -76.5376205 19.8269272 N

8.6281805 -75.4858246 20.7548676 N

16.9929161 -100.9400024 29.3657684 N

15.7367716 -101.5710831 28.8090038 N

4.5037360 -76.8084641 21.2941017 N

6.5346084 -76.5376205 19.8269272 N

15.7367716 -101.5710831 28.8090038 N

14.5182476 -101.7335892 29.6893082 N

4.5664368 -76.0275116 23.6892185 N

4.5037360 -76.8084641 21.2941017 N

14.5182476 -101.7335892 29.6893082 N

14.5558681 -101.2650146 31.1263790 N

6.6600089 -74.9757156 24.6171589 N

4.5664368 -76.0275116 23.6892185 N

14.5558681 -101.2650146 31.1263790 N

15.8120117 -100.6339417 31.6831436 N

8.6908817 -74.7048645 23.1499844 N

6.6600089 -74.9757156 24.6171589 N
15.8120117 -100.6339417 31.6831436 N
17.0305347 -100.4714279 30.8028393 N
4.0867462 -47.6858253 -0.9447947 N
2.5585761 -45.4737396 2.2818718 N
8.3618317 -75.2694397 24.3637867 N
9.2787342 -76.5966873 22.4277878 N
1.5281700 -50.1120872 -3.2266674 N
4.0867462 -47.6858253 -0.9447947 N
9.2787342 -76.5966873 22.4277878 N
7.7435880 -78.0524445 21.0586624 N
-2.5585761 -50.3262634 -2.2818727 N
1.5281700 -50.1120872 -3.2266674 N
7.7435880 -78.0524445 21.0586624 N
5.2915401 -78.1809540 21.6255398 N
-4.0867462 -48.1141815 0.9447947 N
-2.5585761 -50.3262634 -2.2818727 N
5.2915401 -78.1809540 21.6255398 N
4.3746381 -76.8536987 23.5615387 N
-1.5281701 -45.6879196 3.2266665 N
-4.0867462 -48.1141815 0.9447947 N
4.3746381 -76.8536987 23.5615387 N
5.9097838 -75.3979492 24.9306641 N
2.5585761 -45.4737396 2.2818718 N
-1.5281701 -45.6879196 3.2266665 N
5.9097838 -75.3979492 24.9306641 N
8.3618317 -75.2694397 24.3637867 N
7.0000000 0.0000000 0.0000000 N

3.5000000 0.0000000 6.0621777 N

2.1000001 -50.0000000 3.6373069 N

4.2000003 -50.0000000 0.0000000 N

3.5000000 0.0000000 -6.0621777 N

7.0000000 0.0000000 0.0000000 N

4.2000003 -50.0000000 0.0000000 N

2.1000001 -50.0000000 -3.6373069 N

-3.5000000 0.0000000 -6.0621777 N

3.5000000 0.0000000 -6.0621777 N

2.1000001 -50.0000000 -3.6373069 N

-2.1000001 -50.0000000 -3.6373069 N

-7.0000000 0.0000000 0.0000000 N

-3.5000000 0.0000000 -6.0621777 N

-2.1000001 -50.0000000 -3.6373069 N

-4.2000003 -50.0000000 0.0000000 N

-3.5000000 0.0000000 6.0621777 N

-7.0000000 0.0000000 0.0000000 N

-4.2000003 -50.0000000 0.0000000 N

-2.1000001 -50.0000000 3.6373069 N

3.5000000 0.0000000 6.0621777 N

-3.5000000 0.0000000 6.0621777 N

-2.1000001 -50.0000000 3.6373069 N

2.1000001 -50.0000000 3.6373069 N

//Total Polygons

//óóóóóóó

98

//All Polygons

//óóóóóóó

6 0 1 2 3 4 5 0x0a0 both id=2

4 6 7 8 9 0x770 id=1

4 10 11 12 13 0x770 id=1

4 14 15 16 17 0x770 id=1

4 18 19 20 21 0x770 id=1

4 22 23 24 25 0x770 id=1

4 26 27 28 29 0x770 id=1

6 30 31 32 33 34 35 0x0a0 both id=2

4 36 37 38 39 0x770 id=1

4 40 41 42 43 0x770 id=1

4 44 45 46 47 0x770 id=1

4 48 49 50 51 0x770 id=1

4 52 53 54 55 0x770 id=1

4 56 57 58 59 0x770 id=1

4 60 61 62 63 0x770 id=1

4 64 65 66 67 0x770 id=1

4 68 69 70 71 0x770 id=1

4 72 73 74 75 0x770 id=1

4 76 77 78 79 0x770 id=1

4 80 81 82 83 0x770 id=1

6 84 85 86 87 88 89 0x0a0 both id=2

4 90 91 92 93 0x770 id=1

4 94 95 96 97 0x770 id=1

4 98 99 100 101 0x770 id=1

4 102 103 104 105 0x770 id=1

4 106 107 108 109 0x770 id=1

4 110 111 112 113 0x770 id=1

6 114 115 116 117 118 119 0x0a0 both id=2

4 120 121 122 123 0x770 id=1

4 124 125 126 127 0x770 id=1

4 128 129 130 131 0x770 id=1

4 132 133 134 135 0x770 id=1

4 136 137 138 139 0x770 id=1

4 140 141 142 143 0x770 id=1

4 144 145 146 147 0x770 id=1

4 148 149 150 151 0x770 id=1

4 152 153 154 155 0x770 id=1

4 156 157 158 159 0x770 id=1

4 160 161 162 163 0x770 id=1

4 164 165 166 167 0x770 id=1

4 168 169 170 171 0x770 id=1

4 172 173 174 175 0x770 id=1

4 176 177 178 179 0x770 id=1

4 180 181 182 183 0x770 id=1

4 184 185 186 187 0x770 id=1

4 188 189 190 191 0x770 id=1

6 192 193 194 195 196 197 0x0a0 both id=2

4 198 199 200 201 0x770 id=1

4 202 203 204 205 0x770 id=1

4 206 207 208 209 0x770 id=1

4 210 211 212 213 0x770 id=1

4 214 215 216 217 0x770 id=1

4 218 219 220 221 0x770 id=1

6 222 223 224 225 226 227 0x0a0 both id=2

4 228 229 230 231 0x770 id=1

4 232 233 234 235 0x770 id=1

4 236 237 238 239 0x770 id=1

4 240 241 242 243 0x770 id=1

4 244 245 246 247 0x770 id=1

4 248 249 250 251 0x770 id=1

4 252 253 254 255 0x770 id=1

4 256 257 258 259 0x770 id=1

4 260 261 262 263 0x770 id=1

4 264 265 266 267 0x770 id=1

4 268 269 270 271 0x770 id=1

4 272 273 274 275 0x770 id=1

6 276 277 278 279 280 281 0x0a0 both id=2

4 282 283 284 285 0x770 id=1

4 286 287 288 289 0x770 id=1

4 290 291 292 293 0x770 id=1

4 294 295 296 297 0x770 id=1

4 298 299 300 301 0x770 id=1

4 302 303 304 305 0x770 id=1

6 306 307 308 309 310 311 0x0a0 both id=2

4 312 313 314 315 0x770 id=1

4 316 317 318 319 0x770 id=1

4 320 321 322 323 0x770 id=1

4 324 325 326 327 0x770 id=1

4 328 329 330 331 0x770 id=1

4 332 333 334 335 0x770 id=1

4 336 337 338 339 0x770 id=1

4 340 341 342 343 0x770 id=1

4 344 345 346 347 0x770 id=1

4 348 349 350 351 0x770 id=1

4 352 353 354 355 0x770 id=1

4 356 357 358 359 0x770 id=1

4 360 361 362 363 0x770 id=1

4 364 365 366 367 0x770 id=1

4 368 369 370 371 0x770 id=1

4 372 373 374 375 0x770 id=1

4 376 377 378 379 0x770 id=1

4 380 381 382 383 0x770 id=1

4 384 385 386 387 0x770 id=1

4 388 389 390 391 0x770 id=1

4 392 393 394 395 0x770 id=1

4 396 397 398 399 0x770 id=1

4 400 401 402 403 0x770 id=1

4 404 405 406 407 0x770 id=1

Notice that the vertex lines contain an N, indicating that automatic normal calculation is required. Also notice that there are really only two color values used, 0x0a0 and 0x770 (green and brown); that makes sense, because it's a tree.

Data Structures

Now that we know what our input file looks like, we can start thinking about what kind of data structures we're going to use. Clearly, we'll need to be able to represent a vertex:

typedef struct { float x, y, z; } Vertex;

and a polygon:

typedef struct

{

int npts; /* number of points in this polygon */

int *points; /* indices into the vertex array */

int color; /* index into the array of color values */

} Poly;

For each vertex, we have the X, Y, and Z components; the normals will be stored separately. For each

polygon, we have a count of the number of points it uses and its color value. We also have an array to store the indices into the vertex array for the points making up this polygon. Notice that we don't use a fixed-size array; if we did, we would not only limit the number of points per polygon to whatever the size of the array was, we would also waste a lot of memory. Instead, we have a pointer to an array that will be dynamically allocated.

For the color, we could just store the 12-bit value (three hex digits is twelve bits) in the color field; however, as we noticed in the sample NFF file above, the same colors get used over and over. What we'll do is keep an array of all the different colors used in the file, and use the color field in each polygon as an index into that array. We'll need a type for storing the color values:

```
typedef unsigned long Color;
```

Now we have all our basic types defined. Let's create the actual data structures themselves. There'll be several arrays: one for the colors, one for the vertices, one for the normals, and one for the polygons. They'll all be

dynamically allocated, so we'll need pointers to each (set to NULL initially) and a count of the number of elements in each.

```
Color *colors = NULL; /* one entry for each *unique* color */
```

```
int ncolors = 0; /* number of unique colors defined */
```

```
Vertex *vertices = NULL; /* the array of vertices */
```

```
int nvertices = 0; /* number of vertices defined */
```

```
Vertex *normals = NULL; /* the array of normals */
```

```
int nnormals = 0; /* number of normals defined */
```

```
Poly *polys = NULL; /* polygons */
```

```
int npolys = 0; /* number of polygons defined */
```

Since the NFF2VRML program allows objects to be scaled to a different size, we'll need to keep track of our scale factor; by default, it's 1.0 (meaning that the coordinates shouldn't be scaled at all).

```
float scale = 1.0; /* scale factor */
```

We also want to keep track of the name of the object given in the input file.

```
char objname[256]; /* the name of the object */
```

The information about one-sided versus two-sided polygons needs to be preserved unfortunately. VRML is unable to specify "sidedness" on a face-by-face basis; an entire IndexedFaceSet is either made up of single-sided polygons (shapeType SOLID in the ShapeHints node) or double-sided ones (shapeType UNKNOWN_SHAPE_TYPE). We'll just keep a single flag around that indicates which it is; by default, we'll assume it's SOLID (so faces are visible only from the "outside"). Remember from Chapter 11 that inside/out is defined in terms of vertex ordering; NFF files always list the points in a polygon in a counterclockwise direction when seen from the "outside" (i.e., the visible side).

```
int solid = 1; /* set if backface removal okay */
```

We're going to be parsing and storing the camera location and direction; at the moment, we won't be using them for anything, but it doesn't do any harm to keep them around.

```
Vertex camloc = { 0, 0, 0 }; /* camera location (currently not _used) */
```

```
Vertex camdir = { 0, 0, 0 }; /* camera direction (currently not _used) */
```

This is our first use of the Vertex type to store X, Y, Z information.

We'll keep track of the name of our program in a variable called "progrname". If we wanted to, we could probably set this variable to be argv[0] in main().

```
char *progrname = "NFF2VRML"; /* name of this program */
```

Utility Routines

Well, that's all our global data. Now we need a couple of small routines to help us along; the first one will read a single line of input from the file, stripping off comments and skipping over blank lines.

```
int getline(char *buff)
```

```
{
```

```
do
```

```
{
```

```
char *p;
```

```
if (gets(buff) == NULL) /* end of file */
```

```
return 0;
```

```
p = strstr(buff, "//"); /* look for start of comment */
```

```
if (p)
```

```
*p = '\0'; /* truncate the line at that point */
```

```
} while (buff[0] == '\0'); /* skip blank lines */
```

```
return 1; /* not end of file */
```

```
}
```

The line is read from standard input using gets(). The first occurrence (if any) of the "//" sequence is found, and a null byte is written there to truncate the line. If the line was empty (or is now empty, after the comment has been stripped off) we continue reading; if gets() ever returns NULL then we return 0 to signal end of file. Once a non-empty line is found, we return 1.

We're going to be allocating a lot of arrays in this program (vertices, normals, polygons, point lists,

colors), and each time we do, we're going to have to check if the allocation failed due to lack of available memory. To save a lot of duplication, let's create a function that does the allocation, checks to see if it worked, and exits with an error message if there was a problem:

```
void *ecalloc(int nitems, int size)  
  
{  
  
void *p = calloc(nitems, size);  
  
if (p == NULL)  
  
{  
  
fprintf(stderr, "%s: calloc() failed ó not enough _memory\n", progname);  
  
exit(1);  
  
}  
  
return p;  
  
}
```

Our final utility routine is used to position a camera in the world. We could have just used the viewpos and the viewdir, but there are three good reasons not to. The first is that they're optional, and therefore may not have been specified in the input file. The second is that it's a little tricky to compute the orientation field of a PerspectiveCamera node based on the viewdir; they're not the same thing, and there's a bit of math involved. The third is that it's much more interesting to compute a "good" camera location based on the object's geometry.

```
void make_camera(void)  
  
{  
  
int i;  
  
float minx, maxx, miny, maxy, minz, maxz;  
  
/* compute bounding box */  
  
minx = maxx = vertices[0].x;  
  
miny = maxy = vertices[0].y;  
  
minz = maxz = vertices[0].z;  
  
for (i = 1; i < nvertices; ++i)  
  
{  
  
if (vertices[i].x < minx)  
  
minx = vertices[i].x;
```

```

if (vertices[i].x > maxx)
    maxx = vertices[i].x;
if (vertices[i].y < miny)
    miny = vertices[i].y;
if (vertices[i].y > maxy)
    maxy = vertices[i].y;
if (vertices[i].z < minz)
    minz = vertices[i].z;
if (vertices[i].z > maxz)
    maxz = vertices[i].z;
}

printf("\tPerspectiveCamera { position %f %f %f }\n",
(minx + maxx) / 2, -(miny + maxy) / 2, maxz + (maxz - minz));
}

```

The `make_camera` function starts by computing a "bounding box" around the object. This is useful if you ever decide to write a VRML browser, since it's sometimes used in culling objects. A bounding box is just the smallest (axis-aligned) box which completely contains all the vertices in the object. To compute it, we use six variables: a minimum and a maximum value for each of the three axes (X, Y, and Z). We start by setting all of them to coincide with the first vertex in the file; then we run through the remaining vertices, and update the min and max values. For example, if we find a vertex whose X value is less than the minimum x, that value becomes the new minimum X.

Once we've run through all the vertices, we'll have the limits of our bounding box. We then position the camera so that it's half-way along the box in X and Y, and set back a bit in Z. The $(\text{minx} + \text{maxx}) / 2$ is just a shorthand way of saying $(\text{minx} + (\text{maxx} - \text{minx}) / 2)$; work it out for yourself. The Y coordinate gets its sign flipped, because NFF and VRML have different opinions about which way their coordinate systems are oriented.

The expression $(\text{maxz} - \text{minz})$ is the "depth" of the box; since Z is pointing towards us, we subtract that distance from the maxz (i.e., nearest) value.

Parsing the Input

It's now time to look at the main program itself. We start by checking the command line for a scale parameter

```

void main(int argc, char *argv[])

```



```

{
char buff[256];

int i, j;

if (argc > 1)
sscanf(argv[1], "%f", &scale);

```

Next, we check for the "nff" signature at the start of the file; if it's not found, we issue an error message and give up.

```

getline(buff);

if (strnicmp(buff, "nff", 3))
{
fprintf(stderr, "No header line ó not a valid NFF _file\n");

exit(1);
}

```

The strnicmp() function is found in the libraries of most C compilers for Windows and DOS. It does a case-insensitive comparison of the first n characters of two strings.

If your compiler doesn't have it, you'll have to include the following:

```

#include <ctype.h>

int strnicmp(char *a, char *b, int n)
{
int i;

for (i = 0; i < n; ++i)
{
if (a[i] == 0 && b[i] != 0)
return -1; /* string 'a' is shorter */

if (b[i] == 0 && a[i] != 0)
return 1; /* string 'b' is shorter */

if (a[i] == 0 && b[i] == 0)

return 0; /* both are identical, and shorter than 'n'*/

if (toupper(a[i]) != toupper(b[i])) /* mismatch */

```

```

return toupper(a[i]) - toupper(b[i]);
}

return 0; /* strings are identical in the first 'n' characters*/
}

```

Now we keep reading the header of the file; we record the viewpos and viewdir, and issue a warning if the version number of the input file is not 2.0. We also treat anything else (besides "viewdir," "viewpos," and "version") as being the object name and optional shading parameter. If we encounter a line that begins with a numeric digit, it's the vertex count; we break out of the loop.

```

while (getline(buff))
{
    if (isdigit(buff[0]))
        break;

    if (!strnicmp(buff, "viewpos", 7))
        sscanf(buff, "viewpos %f %f %f", &camloc.x, &camloc.y, &camloc.z);
    else if (!strnicmp(buff, "viewdir", 7))
        sscanf(buff, "viewdir %f %f %f", &camdir.x, &camdir.y, &camdir.z);
    else if (!strnicmp(buff, "version", 7))
    {
        float version;
        sscanf(buff, "version %f", &version);
        if (version != 2.0)
            fprintf(stderr, "Warning: NFF version is %f, not 2.0\n", version);
    }
    else
    {
        char tempstr[100];

        /* pick up object name */

        if (sscanf(buff, "%s %s", objname, tempstr) == 2)
            shading = toupper(tempstr[9]) == 'n';
    }
}

```

```
}
```

```
}
```

We read the number of vertices, and call our error-checking version of `calloc( )` to allocate an array of vertices. We also allocate a matching array of normals, once for each vertex. We leave the count of the number of normals (`nnormals`) at zero, though; more about that later.

```
nvertices = atoi(buff);
```

```
vertices = calloc(nvertices, sizeof(Vertex));
```

```
normals = calloc(nvertices, sizeof(Vertex));
```

Our next step is to read in the vertices themselves. If there were six (or more!) numbers on the line, then a normal was specified; we read it into the normals array, and bump the number of normals we've stored.

```
/* read the vertices themselves */
```

```
for (i = 0; i < nvertices; ++i)
```

```
{
```

```
float x, y, z;
```

```
int m;
```

```
getline(buff);
```

```
m = sscanf(buff, "%f %f %f norm %f %f %f",
```

```
&vertices[i].x, &vertices[i].y, &vertices[i].z,
```

```
&normals[nnormals].x, &normals[nnormals].y, &normals[nnormals].z);
```

```
if (m >= 7)
```

```
++nnormals;
```

```
}
```

After this, we pick up the number of polygons and allocate an array of them.

```
getline(buff);
```

```
npolys = atoi(buff);
```

```
/* allocate the polygon array */
```

```
polys = calloc(npolys, sizeof(Poly));
```

We also allocate our color array. Remember, there'll be one entry in this array for each unique color in the input file. The worst case is for every polygon in the file to have its own unique color; therefore, the size of the color array will be the same as the size of the polygon array. As with the normals, however, we don't set `ncolors` yet; it will get incremented once for each unique color we encounter.

```
colors = calloc(npolys, sizeof(Color));
```

We can now read in the polygons.

```
/* read the polygons */
```

```
for (i = 0; i < npolys; ++i)
```

```
{
```

```
Color col;
```

```
char *p;
```

```
/* read the next line of input */
```

```
getline(buff);
```

```
/* if any of the polys are double-sided, the object can't be solid */
```

```
if (strstr(buff, "both"))
```

```
solid = 0;
```

```
/* read the number of points */
```

```
polys[i].npts = strtoul(buff, &p, 0);
```

```
/* allocate the array of points for this polygon */
```

```
polys[i].points = calloc(polys[i].npts, sizeof(int));
```

```
/* read the points themselves */
```

```
for (j = 0; j < polys[i].npts; ++j)
```

```
polys[i].points[j] = strtoul(p, &p, 0);
```

```
/* read the color (assume 12-bit) */
```

```
col = strtoul(p, &p, 0);
```

```
/* see if we've already got this color in the array */
```

```
for (j = 0; j < ncolors; ++j)
```

```
if (colors[j] == col)
```

```
break;
```

```
/* if it's a new color, add it to the array of colors */
```

```
if (j >= ncolors)
```

```
colors[ncolors++] = col;
```

```
polys[i].color = j;
```

```
}
```

Several things to notice here. If any polygon line contains the string "both", then we must make the entire object double-sided (since VRML can't do it on a face-by-face basis). We simply set the solid variable to zero; we'll use it later.

For each polygon, we read the number of points (the strtoul() function is very handy; it just picks up the next number, converts the number base if necessary, and updates a pointer into the input line). We allocate an array to store the points in, and store it in the points field of the polygon. Reading the points themselves is straightforward. We pick up the (12-bit) color, and check to see if it's one we've already seen; if not, we add it to the array. In any case, the color field of the polygon winds up getting set to be the index of that color in the colors[] array.

Generating the Output

We've now successfully read the entire NFF file into memory, and we're ready to write out the corresponding VRML. The first step is to emit the standard VRML version 1.0 header, and an outside Separator. We also call our make_camera() function to position the camera to a suitable location, and put out a named Info node to give the object name as the title of the world; many VRML browsers will then display this in their title bar.

```
printf("#VRML V1.0 ascii\n# Converted by NFF2VRML\n\n");
```

```
printf("Separator {\n");
```

```
make_camera();
```

```
/* put the object name out inside an Info node */
```

```
if (objname[0])
```

```
printf("\tDEF Title Info { string \"%s\" }\n", objname);
```

Since we know the vertices in the original NFF file were counterclockwise, we can mention that fact in a ShapeHints node. We also know whether the object can be treated as SOLID or not, so we mention that too. We also establish that the materials (which correspond to colors, in our case) are bound to each face through an index value.

```
/* faces are counterclockwise; if none of the faces were _double-sided,
```

```
then it's a SOLID shape */
```

```
printf("\tShapeHints { vertexOrdering COUNTERCLOCKWISE %s _}\n",
```

```
solid ? "shapeType SOLID" : "shapeType _UNKNOWN SHAPE _TYPE");
```

```
printf("\tMaterialBinding { value PER_FACE_INDEXED }\n");
```

Next come the coordinates themselves. Since NFF and VRML have different coordinate systems orientations, we negate the Y value. We also use our scale factor to increase or decrease the size of the

object if necessary, by scaling each individual vertex.

```
/* emit the coordinates */  
printf("\tCoordinate3 {\n\t\tpoint [\n");  
for (i = 0; i < nvertices; ++i)  
printf("\t\t\t%f %f %f,\n",  
vertices[i].x * scale,  
-vertices[i].y * scale, /* coordinate system is _flipped */  
vertices[i].z * scale);  
printf("\t\t]\n\t}\n");
```

Since colors correspond to materials, we'll go through the color array and generate a material for each entry.

```
/* put out the unique colors as materials */  
printf("\tMaterial {\n\t\tdiffuseColor [\n");  
for (i = 0; i < ncolors; ++i)  
printf("\t\t\t%f %f %f,\n", ((colors[i] >> 8) & 0x0F)/_15.0, ((colors[i] >> 4) & 0x0F)/15.0, (colors[i] &  
0x0F)/  
_printf("\t\t]\n\t}\n");
```

The red, green, and blue components are obtained by extracting the appropriate nybbles (half-bytes, or four-bit values) from the 12-bit color and scaling them into a floating-point number between zero and one.

Finally, we're ready to emit our IndexedFaceSet. Notice that we stick a -1 at the end of each polygon's list.

```
printf("\tIndexedFaceSet {\n");  
/* put out the actual polygons */  
printf("\t\tcoordIndex [\n");  
for (i = 0; i < npolys; ++i)  
{  
printf("\t\t\tt");  
for (j = 0; j < polys[i].npts; ++j)  
printf("%d,", polys[i].points[j]);
```

```
printf("-1,\n");
```

```
}
```

```
printf("\t\t]\n");
```

The material indices are handled similarly, one per polygon. We make the formatting a bit nicer by starting a new line after every ten values are written out.

```
printf("\t\tmaterialIndex [\n");
```

```
for (i = 0; i < npolys; ++i)
```

```
{
```

```
if (i % 10 == 0)
```

```
printf("\t\t\t");
```

```
printf("%d, ", polys[i].color);
```

```
if ((i+1) % 10 == 0) /* see if we've filled a line */
```

```
printf("\n");
```

```
}
```

```
if (i % 10 != 0)
```

```
printf("\n");
```

```
printf("\t\t]\n");
```

The last thing we do is put out the curly braces that end the IndexedFaceSet node and the outermost Separator node.

```
printf("\t}\n");
```

```
printf("}\n");
```

That's it! The complete, unabridged listing for this program can be found in Appendix E.

Making a Forest

Our second program is a world generator. It produces a pseudo-random forest, made out of trees; the trees themselves were produced using TreeTool and converted using the NFF2VRML program we just wrote.

There are several challenges; we want to randomly position the trees, but we don't want them intersecting with each other. To figure out how to do this, we take a look at the natural world; trees never get very close to each other, partly because their root systems will be feeding off the same nutrients in the soil. We'll use a simplified model in which the world is a grid in the X-Z plane; each grid cell may or may not have a tree in it, but it is guaranteed to never have more than one tree.

We'll use a random number generator to "roll the dice" and see whether a particular cell has a tree or not. We'll use a global threshold value for this. If the random number for a particular cell is greater than the threshold, we place a tree; otherwise, we don't. The lower the threshold value, the higher the density of trees. The threshold value can be anywhere between 0.0 and 1.0; if the value is (for example) 0.75, then a tree will only be placed 25 percent of the time.

```
float placement_threshold = 0.75; /* if rand is greater than this, _place a tree */
```

We'll also need variables for the number of rows and columns in our grid, and for the horizontal and vertical sizes of the cells. This allows us to define both the size of our generated world and the "coarseness" of the grid.

```
int nrows = 32, ncols = 32; /* number of rows and columns in the _grid */
```

```
float horizontal_spacing = 5, vertical_spacing = 5; /* sizes of _the cells */
```

We want to be able to set these parameters, so we'll write a little utility routine that processes command-line arguments.

```
void procarg(char *arg)
```

```
{
```

```
switch (*arg++)
```

```
{
```

```
case 'r': nrows = atoi(arg); break;
```

```
case 'c': ncols = atoi(arg); break;
```

```
case 'p': placement_threshold = atof(arg); break;
```

```
case 'h': horizontal_spacing = atof(arg); break;
```

```
case 'v': vertical_spacing = atof(arg); break;
```

```
case '?':
```

```
fprintf(stderr, "Usage: forest [options]\noptions _include:\n\t-rrows -ccols -pplacement_threshold\n\t-hhorizontal_spacing -vvertical_spacing _-?\n");
```

```
exit(0);
```

```
break;
```

```
default: fprintf(stderr, "Warning: unrecognized option\n"); break;
```

```
}
```

```
}
```


Seeing the Forest Through the Trees

The essence of making a forest is making trees, so we'll write a routine that makes a tree at a particular cell location.

```
void make_tree(int row, int col)  
  
{  
  
float size_factor = (rand() / (float) RAND_MAX) * (2.0 - 0.25) + 0.25;  
  
printf("\tSeparator {\n");  
  
printf("\t\tTransform {\n");  
  
printf("\t\t\t scaleFactor %f %f %f\n", size_factor, _size_factor, size_factor);  
  
printf("\t\t\t rotation 0 1 0 %f\n", (rand() * 2 * 3.1415926) / _RAND_MAX);  
  
printf("\t\t\t translation %f 0 %f\n",  
horizontal_spacing * (col + rand() / (float) RAND_MAX - 0.5),  
vertical_spacing * (row + rand() / (float) RAND_MAX - 0.5));  
  
printf("\t\t}\n");  
  
printf("\t\tUSE tree\n");  
  
printf("\t}\n");  
  
}
```

Each tree is a separate object, so we surround it by a Separator. We compute a random scale factor; the expression `(rand() / (float) RAND_MAX)` will be a value between zero and one; the sizes of the trees will vary from a minimum of 0.25 to a maximum of 2.0 times a tree's "natural" height. Those values for the trees are arbitrary (that's sort of a pun, if you're multilingual). Any values could be used, but the 0.25 to 2.0 range seems to give the best looking results.

We emit a Transform node to do the scaling, along with a random rotation around the Y axis (between zero and 2 PI, or 180 degrees). The location is just the column (or row) in the grid, times the appropriate spacing. We also add a bit of randomness; otherwise, our trees will all be in unnaturally perfect rows and columns. The randomness is expressed as a fraction of the grid cell size. We finally USE the tree itself; the definition for the tree will have already been emitted, in the program's mainline.

The Forest Generator

The actual mainline of the program turns out to be pretty simple. We begin by picking off the command-line arguments and processing them using our `procarg()` routine.

```
void main(int argc, char *argv[])
```

```

{
int i, j;

/* start by processing the command-line arguments */

while (argc > 1)
{
if (*argv[1] == '-')
{
procarg(&argv[1][1]);
óargc;
++argv;
}
}

```

Next, we put out our familiar VRML header and outside Separator, along with a PerspectiveCamera for viewing the scene.

```

printf("#VRML V1.0 ascii\n\n# Generated by FOREST\n\nSeparator _{\n");
printf("\tPerspectiveCamera { position %f %f %f }\n",
ncols * horizontal_spacing / 2, 12.0, nrows * _vertical_spacing * 1.5);

```

The camera is positioned roughly halfway along the X axis of the world (remember, the far left corner of the world is at the origin; the world extends outward in X and Z). The vertical position of the camera is half-way up the height of a tree, and we're positioned back from the world (toward us, in a positive Z direction) by one and a half times the Z extent of the world itself.

We've already seen how we USE a tree; it's now time to DEF it. We'll load the tree itself from a file called "tree1.wrl" by using a WWWInline. We'll also enclose it in a Switch node with a whichChild value of 01, which prevents it from being rendered. That way, the only trees in the output file will be those we created using make_tree(). It would be nice if VRML had a way of defining something without instancing it, but as of version 1.0 it doesn't.

```

/* define (but don't instance!) a tree */

printf("\tSwitch {\n\t\twhichChild -1\n\t\tDEF tree WWWInline _{ name \"tree1.wrl\" }\n\t}\n");

```

Finally, we go through the grid and call make_tree() for all those cells where a random number is greater than the placement threshold. We end by emitting the closing curly brace to end the outermost Separator node.

```

/* go through the array, putting out trees when appropriate */

```

```

for (i = 0; i < nrows; ++i)
for (j = 0; j < ncols; ++j)
if ((rand() / (float) RAND_MAX) > _placement_threshold)
make_tree(i, j);
printf("{}\n");
}

```

That's it. We now have a complete virtual forest generator. The unedited source code for this program can be found in [Appendix E](#).

General Ideas Behind VRML Browsers

Writing a VRML browser is a very large project. Several companies have been working on it for awhile now, with complete programming teams, and they still have only partially compliant browsers. Most VRML browsers are still in "beta" testing and will be for some time to come.

An entire book could probably be written about how to create a VRML browser, so we won't be able to get very far in this one small section. However, we can touch on the basics.

Parsing

The first step in creating your own browser is to write a VRML parser. The parser is the set of routines that reads the actual file and builds a corresponding set of data structures in memory.

If you look on the companion CD-ROM that came with this book, you'll find the complete source code for QvLib, a freeware VRML parser that has been placed in the public domain by SGI. Unfortunately, free software is often worth just what you paid for it; QvLib has a number of bugs and idiosyncrasies that make it hard to work with. You might be better off writing your own.

Choosing a Rendering Engine

The next step is to choose a rendering engine. Different browsers have made different choices; the following table summarizes some of them.

Browser	Rendering Engine
WorldView	RealityLab
WebFX	RenderWare
VR Scout	3DR
WebSpace	OpenGL
Fountain	3DR
VRWeb	Mesa

Another rendering engine, which so far hasn't been as widely used, is BRender from Argonaut Software. And no, "BRender" has nothing to do with me; even though my initials are BR, and they do capitalize BR at the start of the product's name, I have nothing whatsoever to do with BRender or Argonaut Software. It is, however, a very fast software-based renderer; it's certainly in the same class as RealityLab or RenderWare.

Putting It All Together

Once you can parse a VRML file and you have a rendering engine in place, you run into the really complicated (and under-specified) part of the project: grafting the VRML data structures onto those used by the renderer you've chosen. There are no easy answers here; it depends a lot on what features your renderer supports and how much dancing you have to do to work around the ones that aren't there. You may have to sacrifice performance for compatibility, or vice versa.

Miscellaneous Nuisance Issues

You'll have to write code to tessellate complex objects like spheres, cones, and cylinders into simpler polygonal shapes. You'll have to be able to support IndexedFaceSets with non-convex faces; that requires you to decompose the faces. You might have to do that anyway, if your rendering engine only supports triangles (although that's an easier kind of decomposition).

Adding Network Support

Once you've got your browser running locally, you'll want to add network support. This requires you to get familiar with the sockets interface on your platform, as well as the HTTP specification. You should also look at how to interface to the user's existing browser (e.g., Netscape).

Supporting Texture Maps

You need to have code to read (at least) JPEG and PNG, and possibly GIF as well. You should be able to find that sort of thing out on the Web.

GZIP Decompression

A growing number of worlds are compressed using GZIP; adding GZIP support to your program, even if it's only in the form of a call to a completely external program, will make your users a lot happier.

User Interface

Speaking of happy users, try to put some time into your user interface design. Among the features you should definitely have are control over rendering quality (wireframe, flat, gouraud, texture mapping) as well as over the navigation mode (walk, fly, examine). Spend some time studying (and using!) existing browsers; decide what you like about them, and what you don't like.

Navigation is a really important issue; there are lots of ways to let the user move around, and you want to choose something that's very natural and intuitive. You might do well to just choose your favorite style from amongst the browsers that are out there. Eventually, some sort of standard will emerge that will let people change from one browser to another without having to learn a whole new way of navigating.

General Advice

Don't do it. It's a lot of work, and it's unlikely you'll come with anything better than what's already out there.

If you're like me, though, you'll ignore this advice and do it anyway.

From Here...

We've written a translator and a generator, and we've talked about how to approach writing a browser. We've also come to the end of the main body of this book; from here on out, it's all appendixes. Have fun, and see you in cyberspace!



Notice: This material is excerpted from *Special Edition Using VRML*, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



Appendix A - Essential Math for Using VRML

Throughout this book, I've made reference to a variety of mathematical techniques. Now it's time to actually discover how it really works.

In this chapter, you'll learn

- More about vertices, vectors, and cartesian coordinates
- How to perform basic operations on vectors
- What a mathematical plane is, and what it's used for
- What a matrix is
- How transformations are represented by matrices

Let's start with the basics....

Vertices and Vectors

These two terms are often used interchangeably, and they are related to each other. As you know from previous chapters, a vertex is just a location in space represented by three numbers: X, Y, and Z. A vector is also represented by X, Y, and Z; the difference is more conceptual than practical. Instead of a location in space, a vector specifies a direction and a magnitude. The difference between a vertex and a vector is the same as the difference between a Coordinate3 node and Normal node.

Imagine two vertices in space: one is at (12 18 72) and the other at (47 39 100). If you were to travel from the first point to the second, you'd be traveling along a vector; that vector can be computed by subtracting your starting point from your destination, component by component. In other words, you'd be traveling along (47-12 39-18 100-72), or (35 21 28). This operation is called "vector subtraction."

Another way of thinking of it is that a vertex is a vector starting from the origin; in other words, (12 18 72) is really (12-0 18-0 72-0). That's why it's called "vector subtraction" instead of "vertex subtraction."

Since you can subtract vectors, it makes sense that you can add them as well. Figure A.1 shows three vectors: A, B, and C:

Fig. A.1

Vector addition and subtraction.

Vector A is just vertex V2 minus vertex V1, and B is just V3 minus V2. Vector C is V3 minus V1, but it's also the vector sum of A and B; in other words, $C = A + B$. Similarly, $B = C - A$.

Notice that a vector is not the same as a line: a line connects two specific points in space, whereas a vector defines a direction. If you take a vector and "pin" it to a vertex, the result is a line.

Vector Cross Products

Not only can you add and subtract vectors, you can also multiply them. However, multiplication is a little different from addition and subtraction. There are two ways of multiplying vectors: the "cross product" and the "dot product."

The cross product $C = A \text{ cross } B$ is computed as follows:

$$C.x = A.y * B.z - A.z * B.y$$

$$C.y = -(A.x * B.z - A.z * B.x)$$

$$C.z = A.x * B.y - A.y * B.x$$

The result is a vector that is perpendicular to both A and B. In other words, if A and B are two vectors, then C will be a vector that sticks up at right angles to both of them. Figure A.2 shows this relationship.

insert figure

Fig. A.2

Two vectors and their cross product.

This only works if neither A nor B are the null vector (0 0 0). A and B also cannot be collinear (that is, they can't point in the same direction).

So if you have two vectors, you can find a third which is perpendicular to both of them. You can then take the cross product of that vector with either of the first two, to find one that's perpendicular to both of them; the result is three perpendicular vectors as shown in Figure A.3:

insert figure

Fig. A.3

Three mutually-perpendicular vectors.

These three vectors define a coordinate system; they're just like a set of coordinate axes, but they're not necessarily oriented parallel to the world's X, Y, and Z axes.

The primary use of the cross product is to generate normal vectors. If you have three points in a polygon (or face), they can be used to define two vectors (just like the V1, V2, and V3 points above were used to define the A and B vectors). The cross product of those two vectors is a vector perpendicular to (in other words, "normal to") the polygon.

One important point to keep in mind is that the order in which we do the cross product determines which direction the normal points. In other words, A cross B points in the opposite direction from B cross A. Similarly, the directions in which A and B point is important; since that depends on the order of the vertices V1, V2, and V3, we need to be consistent. You may remember that in earlier chapters we mentioned that the order in which a polygon's vertices are listed is important; if they're clockwise, the vector points in one direction and if they're counterclockwise it points in the other.

Since the direction of the normal vector is important for backface removal and lighting, consistency matters!

Vector Magnitude and Normalized Vectors

The length of a vector is called it's "magnitude," and it's computed using the formula

$$\text{magnitude}(A) = \sqrt{A.x * A.x + A.y * A.y + A.z * A.z}$$

In other words, the square root of the sum of the squares of the X, Y, and Z components of the vector.

If we take a vector and divide each of its components (X, Y, and Z) by the magnitude, we get something called a "normalized" vector. A normalized vector always has a magnitude of 1.0, which turns out to be quite handy.

NOTE

Note that a "normalized" vector has nothing to do with a "normal" vector; "normalized" means its magnitude is 1.0, whereas "normal" means it's perpendicular to something else. A normal vector can be (but doesn't have to be) normalized.

Vector Dot Products

We've seen how cross products can be used for computing normals. Dot products are very different.

The dot product is just

$$A \text{ dot } B = A.x * B.x + A.y * B.y + A.z * B.z$$

In other words, you multiply the X components of the two vectors, then multiply the Y components, then multiply the Z components, and finally add them all together. The result is not a vector; it's a "scalar" (that is, a single value).

Compare this to the magnitude calculation that we did earlier. Does it look familiar? The magnitude is just the square root of the dot product of a vector with itself:

$$\text{magnitude}(A) = \sqrt{A \cdot A}$$

The other important property of the dot product is that it's related to the angle between the two vectors. More specifically,

$$A \cdot B = \text{magnitude}(A) * \text{magnitude}(B) * \cos(\theta)$$

where θ is the angle between A and B.

Dot Products and Lighting

The main use of the dot product in VR is in lighting calculations. You may remember from Chapter 14 that the intensity of the diffuse light at a particular vertex is related to the angle between the incident light and the normal vector at that vertex. In fact, it's directly proportional to the cosine of the angle between the two. To compute that intensity, we just use the dot product:

$$\cos(\theta) = (L \cdot N) / (\text{magnitude}(L) * \text{magnitude}(N))$$

where L is the vector from the light source to the vertex and N is the normal at that vertex (as specified in a VRML Normal node or computed by the browser).

Now, the $\text{magnitude}()$ calculation is pretty expensive: it involves doing a square root. To do the lighting calculation above, we have to do it twice! Very slow. Not only that, we have to do this for every vertex, for every light source. In other words, if there are 100 vertices and 5 light sources, we have to do it 500 times! That's why adding additional lights to a VRML scene makes it slow down so much. Doing a thousand square roots (two for each vertex, for each light) is not cheap.

However, we can save ourselves one $\sqrt{}$ operation by pre-normalizing the normal vectors; that means that $\text{magnitude}(N)$ is guaranteed to be 1.0, so we can ignore it. Of course, that only works if we transform the light's location into object space, where the precomputed normal vector is stored; otherwise, we'd have to transform the normal (which means re-normalizing, since the transform may involve scaling the vector).

We still have to compute the vector L (which is just the light source location minus the vertex) and find its magnitude. Or do we? We can't really avoid it if it's a `PointLight` or `SpotLight`, but if it's a `DirectionalLight` then we can save ourselves some work. A directional light source shines the same on all vertices; you don't have to compute the vector from the light source to the vertex, since there isn't any one point that the light is coming from. Therefore the direction vector can be pre-normalized; since its magnitude is known to be 1.0, the intensity is just the dot product of the lighting direction vector with the normal—no square roots involved! That's why a `DirectionalLight` is cheaper than a `PointLight` or a `SpotLight`.

Planes

The difference between a plane and a polygon is the same as the difference between an infinite line and a line segment. A polygon is a flat surface, bounded by edges and vertices; a plane is like an infinitely large piece of fabric, stretching out forever in all directions.

Just as a line segment is a segment of an infinite line that's defined by two vertices, a polygon (or face) is a segment of a plane that's defined by three or more vertices. A polygon lies in a plane.

The equation of a plane is

$$ax + by + cz + d = 0$$

What that means is that for any point (x y z) in the plane, the above equation is true. For every point not in the plane, the above equation is false. The equation determines which vertices make up the plane.

Now, here's something interesting: the a, b, and c values are the X, Y, and Z components of a vector normal to the plane. (Note that the a, b, and c here are numbers, not vectors like A, B, and C above).

You'll also notice that the above equation can be written like this:

$$-d = ax + by + cz$$

In other words, d is just (a b c) dot (x y z), with the sign flipped.

Now, if you have three points in the same plane (in fact, any three points define a plane), you can get two vectors; if you take the cross product of those vectors you get the normal to the plane (in other words, the a, b, and c values). Then take the dot product of that vector with any of the three original vertices, and the result is -d. That's how you compute the equation of a plane given three points. You can save a step if you already have the normal vector and any vertex in the plane.

Planes and Backface Removal

Now, here's where the plane equation gets used in a VR system. You may remember from Chapter 11 that one of the steps in the pipeline is backface removal. In order to do backface removal, you have to determine which side of a polygon the user's viewpoint is on.

The equation of a plane is just a special case of a more general point-to-plane distance equation:

$$\text{distance} = ax + by + cz + d$$

The result is the distance of the point (x y z) from the plane whose coefficients are a, b, c, and d. Now, for a point in the plane, the distance in this equation will be zero, which gives us the earlier equation for the plane.

As it turns out, all we need to know for backface removal is the sign of the distance. If it's positive, we're on one side of the plane (and therefore the polygon); if it's negative, we're on the other side. So, we take the dot product of the viewer's (x y z) coordinates with the polygon's (precomputed) normal vector, and compare the result to d to determine whether the front or back of the face is towards us.

Matrices

In VR applications, a matrix is a four-by-four array of numbers. In general, a matrix can have any number of rows and columns; for example, a vector can be thought of as a one-by-three matrix. In fact, vectors are often stored as one-by-four matrices, with the fourth element being 1.0; this makes things

work out nicely, for reasons that will become clear later.

Unlike vectors, you never have any reason to add or subtract matrices; the only thing you ever do with matrices is multiply them.

$$\underline{C = A * B}$$

To multiply two matrices, you compute each element of the result as the dot product of the corresponding row of one matrix and column of the other. In other words, to compute the element in the second row of the third column in matrix C, you would multiply the second row of A by the third column of B. You would do this for each of the sixteen elements of the matrix.

Since a vector is just a special kind of matrix, you can multiply a vector by a matrix to get another vector. This is referred to as "transforming" the vector using the matrix. Sound familiar? That's rightóthese matrices are used to store the kinds of transformations I've talked about many times throughout this book.

If you have a vector $B = (x \ y \ z \ 1)$ and a matrix A that looks like this:

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

Then you would compute vector $C = A * B$ as follows:

$$\underline{C[0] = ax + by + cz + d}$$

$$\underline{C[1] = ex + fy + gz + h}$$

$$\underline{C[2] = ix + jy + kz + l}$$

$$\underline{C[3] = mx + ny + oz + p}$$

This should also look familiar; each row of elements in a matrix corresponds to the equation of a plane. The 0th, 1st, and 2nd elements of C (i.e., the X, Y, and Z components) are just the distance from (x y z) to the corresponding plane. The upper three-by-three sub-matrix is a set of three vectors perpendicular to the three planes; those vectors define a coordinate system, as described earlier.

So, a matrix can transform a vertex (x y z) from one coordinate system to anotheróin other words, a matrix is a transform, and a transform is a matrix.

Identity Matrix

A matrix that doesn't change the vertex at all is called an "identity" matrix, and it looks like this:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

that is, it has 1's on the diagonal and 0's everywhere else. Therefore,

$$\underline{C[0] = 1 * x + 0 * y + 0 * z + 0 = x}$$

$$\underline{C[1] = 0 * x + 1 * y + 0 * z + 0 = y}$$

$$\underline{C[2] = 0 * x + 0 * y + 1 * z + 0 = z}$$

$$\underline{C[3] = 0 * x + 0 * y + 0 * z + 1 = 1}$$

In other words, C is just (x y z 1.0)óthe same as the original vector.

Scaling Matrix

Now, let's say we wanted to scale a vertex. The matrix is just the same as an identity matrix, but instead of 1.0 along the diagonal, we have the corresponding scale values:

S.x	0	0	0
0	S.y	0	0
0	0	S.z	0
0	0	0	1

This means that

$$\underline{C[0] = S.x * x}$$

$$\underline{C[1] = S.y * y}$$

$$\underline{C[2] = S.z * z}$$

$$\underline{C[3] = 1.0}$$

So each component gets scaled separately.

Translation Matrix

Translation is even easier: only the fourth column of the matrix is used, and it stores the translation values. In other words, the matrix is:

1	0	0	T.x
0	1	0	T.y
0	0	1	T.z
0	0	0	1

So

$$\underline{C[0] = x + T.x}$$

$$\underline{C[1] = y + T.y}$$

$$\underline{C[2] = z + T.z}$$

$$\underline{C[3] = 1.0}$$

This just adds the translation, element-by-element, to the vector.

Rotation Matrices

Now, rotation matrices are more complex. The basic idea is to create a two-by-two sub-matrix that looks something like this:

$$\cos(\theta) \quad \sin(\theta)$$

$$-\sin(\theta) \quad \cos(\theta)$$

and position those four elements within the upper three-by-three sub-matrix of the overall four-by-four transformation matrix.

It's not as complicated as it sounds.

For example, let's say we're rotating around the Z axis; the matrix would look like this:

$\cos(\theta)$	$\sin(\theta)$	0	0
$-\sin(\theta)$	$\cos(\theta)$	0	0
0	0	1	0
0	0	0	1

Therefore

$$\underline{C[0] = x * \cos(\theta) + y * \sin(\theta)}$$

$$\underline{C[1] = x * -\sin(\theta) + y * \cos(\theta)}$$

$$\underline{C[2] = z}$$

$$\underline{C[3] = 1.0}$$

which is what you would expect; the Z component doesn't change when you rotate around the Z axis, but X and Y do.

Similarly, a rotation around the X axis would look like this:

1	0	0	0
0	$\cos(\theta)$	$-\sin(\theta)$	0
0	$-\sin(\theta)$	$\cos(\theta)$	0
0	0	0	1

and a rotation around Y would be:

$\cos(\theta)$	0	$\sin(\theta)$	0
0	1	0	0

$-\sin(\theta)$	0	$\cos(\theta)$	0
0	0	0	1

Again, notice that Y will not be altered by this matrix.

Now, what if we have a rotation around an arbitrary axis, not just one of the X, Y, and Z axes? Well, the complete matrix looks like this:

???help-audie

$\cos(\theta) + x*x*(1-\cos(\theta))$	$x*y*(1-\cos(\theta)) + z*\sin(\theta)$	$x*z*(1-\cos(\theta)) - y*\sin(\theta)$	0
$x*y*(1-\cos(\theta)) - z*\sin(\theta)$	$\cos(\theta) + y*y*(1-\cos(\theta))$	$y*z*(1-\cos(\theta)) - x*\sin(\theta)$	0
$x*z*(1-\cos(\theta)) + y*\sin(\theta)$	$y*z*(1-\cos(\theta)) - x*\sin(\theta)$	$\cos(\theta) + z*z*(1-\cos(\theta))$	0
0	0	0	1

Notice that if we plug in an axis of (1 0 0), we get the matrix for a rotation around X that was shown above. The same is true for Y and Z; the rotation-around-axis matrix is the general case, and the others are all just special cases.

Accumulating Transforms

To accumulate transforms, you just multiply the matrices. For example, if we have a vertex V, and matrices for transforming it by scaling (S), rotation (R), and translation (T), we would do the following:

$$V2 = T * R * S * V$$

The transformations are applied such that the ones closest to V are done first; that is, the vertex will be transformed by S to scale it, then by R to rotate it, and then by T to translate it.

However, we can multiply the matrices together first:

$$V2 = (T * R * S) * V$$

and store this accumulated transform. This turns out to be much more efficient, since we'll be transforming a large number of vertices using the same set of transforms. When traversing a VRML object hierarchy, you're essentially multiplying transform matrices as you descend the tree, and using the accumulated matrices to transform the vertices of a shape.

The order of matrix multiplication is important; $T * R * S$ is not the same as $S * R * T$!

Relationship to Transformation Nodes

Obviously, the VRML Scale node corresponds to a scaling matrix, and the Translation node to a translation matrix. The Rotation node gives the axis and angle, which produces the rotation matrix mentioned above.

A Transform node that looks like this:

Transform {

translation T.x T.y T.z

rotation R.x R.y R.z R.a

scaleFactor S.x S.y S.z

scaleOrientation O.x O.y O.z O.a

center C.x C.y C.z

}

will produce a set of matrices that perform the following operations:

Centering

The vertex is first shifted so that the origin is at (C.x C.y C.z); this is done using a translation matrix created from (-C.x -C.y -C.z). In other words, we subtract the center point from every vertex; that way the rotation point (C.x C.y C.z) becomes the origin (0 0 0).

Scaling Rotation

The vertex is next rotated to align its coordinate system to the scaling orientation given by (O.x O.y O.z -O.a). Note that the angle O.a is negated; that means the object is rotated in the opposite direction.

Scaling

The vertex is then scaled using the scaling matrix resulting from (S.x S.y S.z).

Reverse Scaling Rotation

The vertex is then rotated back, undoing the result of the scaling rotation; this is done using a rotation around (O.x O.y O.z O.a).

Rotation

Next, the vertex is rotated by the specified rotation (R.x R.y R.z R.a).

Reverse Centering

The vertex is then shifted back, by adding back the (C.x C.y C.z) vector.

Translation

Finally, the vertex is translated by $(T.x \ T.y \ T.z)$.

All of these operations are obviously represented by matrices, and the matrices for all of the individual steps are multiplied together to produce a single, cumulative transformation matrix. In other words, despite its apparent complexity, the Transform node produces just one matrix.

The Peculiar MatrixTransform Node

The MatrixTransform node stores the matrix directly. However, it stores it in transposed form; that is, the rows and columns are swapped relative to what you would expect. In other words, the first four numbers listed in the node are the first column, not the first row, and so on. This may seem strange; it's a leftover from OpenInventor, the format that originally inspired VRML.

Inverse Matrices

An inverse matrix is one that does the opposite transform from its non-inverted equivalent. This can be very useful. For example, we mentioned earlier (under lighting) that we need to transform the light source into object coordinates; this can easily be done by taking the object's transform matrix (the one that converts its object coordinates into world coordinates) and computing its inverse. Each light source's location (in world coordinates) is then transformed using that inverse matrix to find the light's location in object coordinates.

That's also why the 1.1 version of the VRML specification requires that the matrix specified in a MatrixTransform node be invertible.

From Here...

Well, that's all the basic math you're likely to need. If you need more details, there are several excellent graphics textbooks around that will explain it all in more detail. In particular, "Computer Graphics Principles and Practice" by Foley and van Dam is worth checking out.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



Appendix B - The VRML 1.0 Specification

This Appendix contains the complete specification of the Virtual Reality Modeling Language, version 1.0. The specification is in the public domain, and may be distributed without restriction. Note that items that will change in version 1.1 have been marked with [see change #nn] throughout; the numbers refer to the list of approved changes in Appendix C.

This specification was originally written by:

- Gavin Bell, Silicon Graphics, Inc.
- Anthony Parisi, Intervista Software
- Mark Pesce, VRML List Moderator

Acknowledgments

I want to thank three people who have been absolutely instrumental in the design process: Brian Behlendorf, whose drive (and disk space) made this process happen; and Tony Parisi and Gavin Bell, the final authors of this specification, who have put in a great deal of design work, ensuring that we have a satisfactory product. My hat goes off to all of them, and to all of you who have made this process a success.

- ðB7 Mark Pesce

I would like to add a personal note of thanks to Jan Hardenbergh of Oki Advanced Products for his diligent efforts to keep the specification process on track, and his invaluable editing assistance. I would also like to acknowledge Chris Marrin of Silicon Graphics for his timely contributions to the final design.

- Tony Parisi

Revision History

- First DraftóNovember 2, 1994
- Second DraftóMay 8, 1995

- Third DraftóMay 26, 1995

Table of Contents

Introduction

- VRML Mission Statement
- History
- Version 1.0 Requirements

Language Specification

- Language Basics
- Coordinate System
- Fields
- Nodes
- Instancing
- Extensibility
- An Example

Browser Considerations

- File Extensions
- MIME Types

Introduction

The Virtual Reality Modeling Language (VRML) is a language for describing multi-participant interactive simulationsóvirtual worlds networked via the global Internet and hyperlinked with the World Wide Web. All aspects of virtual world display, interaction, and internetworking can be specified using VRML. It is the intention of its designers that VRML become the standard language for interactive simulation within the World Wide Web.

The first version of VRML allows for the creation of virtual worlds with limited interactive behavior. These worlds can contain objects which have hyperlinks to other worlds, HTML documents, or other valid MIME types. When the user selects an object with a hyperlink, the appropriate MIME viewer is launched. When the user selects a link to a VRML document from within a correctly configured WWW browser, a VRML viewer is launched. Thus VRML viewers are the perfect companion applications to standard WWW browsers for navigating and visualizing the Web. Future versions of VRML will allow for richer behaviors, including animations, motion physics, and real-time multiuser interaction.

This document specifies the features and syntax of Version 1.0 of VRML.

VRML Mission Statement

The history of the development of the Internet has had three distinct phases: first, the development of the TCP/IP infrastructure, which allowed documents and data to be stored in a proximally independent way; that is, Internet provided a layer of abstraction between data sets and the hosts which manipulated them. While this abstraction was useful, it was also confusing; without any clear sense of "what went where," access to Internet was restricted to the class of sysops/net surfers who could maintain internal cognitive maps of the data space.

Next, Tim Berners-Lee's work at CERN, where he developed the hypermedia system known as World Wide Web, added another layer of abstraction to the existing structure. This abstraction provided an "addressing" scheme, a unique identifier (the Universal Resource Locator), which could tell anyone "where to go and how to get there" for any piece of data within the Web. While useful, it lacked dimensionality; there's no there within the web, and the only type of navigation permissible (other than surfing) is by direct reference. In other words, I can only tell you how to get to the VRML Forum home page by saying, "<http://www.wired.com/>", which is not human-centered data. In fact, I need to make an effort to remember it at all. So, while the World Wide Web provides a retrieval mechanism to complement the existing storage mechanism, it leaves a lot to be desired, particularly for human beings.

Finally, we move to "perceptualized" Internetworks, where the data has been sensualized, that is, rendered sensually. If something is represented sensually, it is possible to make sense of it. VRML is an attempt (how successful, only time and effort will tell) to place humans at the center of the Internet, ordering its universe to our whims. In order to do that, the most important single element is a standard that defines the particularities of perception. Virtual Reality Modeling Language is that standard, designed to be a universal description language for multi-participant simulations.

These three phases, storage, retrieval, and perceptualization, are analogous to the human process of consciousness, as expressed in terms of semantics and cognitive science. Events occur and are recorded (memory); inferences are drawn from memory (associations); and from sets of related events, maps of the universe are created (cognitive perception). What is important to remember is that the map is not the territory, and we should avoid becoming trapped in any single representation or worldview. Although we need to design to avoid disorientation, we should always push the envelope in the kinds of experience we can bring into manifestation!

This document is the living proof of the success of a process that was committed to being open and flexible, responsive to the needs of a growing Web community. Rather than reinvent the wheel, we have adapted an existing specification (Open Inventor) as the basis from which our own work can grow, saving years of design work and perhaps many mistakes. Now our real work can beginóthat of rendering our noospheric space.

History

VRML was created in the spring of 1994 at the first annual World Wide Web Conference in Geneva, Switzerland. Tim Berners-Lee and Dave Raggett organized a Birds-of-a-Feather (BOF) session to discuss Virtual Reality interfaces to the World Wide Web. Several BOF attendees described projects already underway to build three-dimensional graphical visualization tools which interoperate with the

Web. Attendees agreed on the need for these tools to have a common language for specifying 3D scene description and WWW hyperlinksóan analog of HTML for virtual reality. The term Virtual Reality Markup Language (VRML) was coined, and the group resolved to begin specification work after the conference. The word "Markup" was later changed to "Modeling" to reflect the graphical nature of VRML.

Shortly after the Geneva BOF session, the www-vrml mailing list was created to discuss the development of a specification for the first version of VRML. The response to the list invitation was overwhelming: within a week, there were over a thousand members. After an initial settling-in period, list moderator Mark Pesce of Labyrinth Group announced his intention to have a draft version of the specification ready by the WWW Fall 1994 conference, a mere five months away. There was general agreement on the list that, while this schedule was aggressive, it was achievable provided that the requirements for the first version were not too ambitious and that VRML could be adapted from an existing solution. The list quickly agreed upon a set of requirements for the first version, and began a search for technologies which could be adapted to fit the needs of VRML.

The search for existing technologies turned up a several worthwhile candidates. After much deliberation the list came to a consensus: the Open Inventor ASCII File Format from Silicon Graphics, Inc. The Inventor File Format supports complete descriptions of 3D scenes with polygonally rendered objects, lighting, materials, ambient properties, and realism effects. A subset of the Inventor File Format, with extensions to support networking, forms the basis of VRML. Gavin Bell of Silicon Graphics has adapted the Inventor File Format for VRML, with design input from the mailing list. SGI has publicly stated that the file format is available for use in the open market, and have contributed a file format parser into the public domain to bootstrap VRML viewer development.

Version 1.0 Requirements

VRML 1.0 is designed to meet the following requirements:

- Platform independence
- Extensibility
- Ability to work well over low-bandwidth connections

As with HTML, the above are absolute requirements for a network language standard; they should need little explanation here.

Early on the designers decided that VRML would not be an extension to HTML. HTML is designed for text, not graphics. Also, VRML requires even more finely tuned network optimizations than HTML; it is expected that a typical VRML scene will be composed of many more "inline" objects and served up by many more servers than a typical HTML document. Moreover, HTML is an accepted standard, with existing implementations that depend on it. To impede the HTML design process with VRML issues and constrain the VRML design process with HTML compatibility concerns would be to do both languages a disservice. As a network language, VRML will succeed or fail independent of HTML.

It was also decided that, except for the hyperlinking feature, the first version of VRML would not support interactive behaviors. This was a practical decision intended to streamline design and implementation. Design of a language for describing interactive behaviors is a big job, especially when

the language needs to express behaviors of objects communicating on a network. Such languages do exist; if we had chosen one of them, we would have risked getting into a "language war." People don't get excited about the syntax of a language for describing polygonal objects; people get very excited about the syntax of real languages for writing programs. Religious wars can extend the design process by months or years. In addition, networked interobject operation requires brokering services such as those provided by CORBA or OLE, services which don't exist yet within WWW; we would have had to invent them. Finally, by keeping behaviors out of Version 1, we have made it a much smaller task to implement a viewer. We acknowledge that support for arbitrary interactive behaviors is critical to the long-term success of VRML; they will be included in Version 2.

Language Specification

The language specification is divided into the following sections:

- Language Basics
- Coordinate System
- Fields
- Nodes
- Instancing
- Extensibility
- An Example

Language Basics

At the highest level of abstraction, VRML is just a way for objects to read and write themselves. Theoretically, the objects can contain anythingó3D geometry, MIDI data, JPEG images, anything. VRML defines a set of objects useful for doing 3D graphics. These objects are called Nodes.

Nodes are arranged in hierarchical structures called scene graphs [see change 10]. Scene graphs are more than just a collection of nodes; the scene graph defines an ordering for the nodes. The scene graph has a notion of stateónodes earlier in the scene can affect nodes that appear later in the scene. For example, a Rotation or Material node will affect the nodes after it in the scene. A mechanism is defined to limit the effects of properties (separator nodes), allowing parts of the scene graph to be functionally isolated from other parts.

A node has the following characteristics:

- What kind of object it is. A node might be a cube, a sphere, a texture map, a transformation, etc.
- The parameters that distinguish this node from other nodes of the same type. For example, each Sphere node might have a different radius, and different texture maps nodes will certainly contain different images to use as the texture maps. These parameters are called Fields. A node can have 0 or more fields.
- A name to identify this node. Being able to name nodes and refer to them elsewhere is very powerful; it allows a scene's author to give hints to applications using the scene about what is in the scene, and creates possibilities for very powerful scripting extensions. Nodes do not have to be

named, but if they are named, they can have only one name. However, names do not have to be unique—several different nodes may be given the same name.

- Child nodes. Object hierarchy is implemented by allowing some types of nodes to contain other nodes. Parent nodes traverse their children in order during rendering. Nodes that may have children are referred to as group nodes. Group nodes can have zero or more children.

The syntax chosen to represent these pieces of information is straightforward:

DEF objectname objecttype { fields children }

Only the object type and curly braces are required; nodes may or may not have a name, fields, and children.

Node names must not begin with a digit, and must not contain spaces or control characters, single or double quote characters, backslashes, curly braces, the plus character, or the period character.

For example, this file contains a simple scene defining a view of a red cone and a blue sphere, lit by a directional light:

#VRML V1.0 ascii

Separator {

DirectionalLight {

direction 0 0 -1 # Light shining from viewer into scene

}

PerspectiveCamera {

position -8.6 2.1 5.6

orientation -0.1352 -0.9831 -0.1233 1.1417

focalDistance 10.84

}

Separator { # The red sphere

Material {

diffuseColor 1 0 0 # Red

}

Translation { translation 3 0 1 }

Sphere { radius 2.3 }

}

Separator { # The blue cube

```
Material {  
diffuseColor 0 0 1 # Blue  
}  
  
Transform {  
translation -2.4 .2 1  
rotation 0 1 1 .9  
}  
  
Cube {}  
}  
}
```

General Syntax

For easy identification of VRML files, every VRML file must begin with the characters:

#VRML V1.0 ascii

[see change #20]. Any characters after these on the same line are ignored. The line is terminated by either the ASCII newline or carriage-return characters.

The "#" character begins a comment; all characters until the next newline or carriage return are ignored. The only exception to this is within string fields, where the "#" character will be part of the string. [see change #15]

Note: Comments and whitespace may not be preserved; in particular, a VRML document server may strip comments and extraneous whitespace from a VRML file before transmitting it. Info nodes should be used for persistent information like copyrights or author information. Info nodes could also be used for object descriptions.

Blanks, tabs, newlines, and carriage returns are whitespace characters wherever they appear outside of string fields. One or more whitespace characters separates the syntactical entities in VRML files, where necessary.

After the required header, a VRML file contains exactly one VRML node. That node may of course be a group node, containing any number of other nodes.

Coordinate System

VRML uses a cartesian, right-handed, 3-dimensional coordinate system. By default, objects are projected onto a 2-dimensional device by projecting them in the direction of the positive Z axis, with the positive X axis to the right and the positive Y axis up. A camera or modeling transformation may be used to alter

this default projection.

The standard unit for lengths and distances specified is meters. The standard unit for angles is radians.

Fields

There are two general classes of fields; fields that contain a single value (where a value may be a single number, a vector, or even an image), and fields that contain multiple values. Single-valued fields all have names that begin with "SF," and multiple-valued fields have names that begin with "MF." Each field type defines the format for the values it writes.

Multiple-valued fields are written as a series of values separated by commas, all enclosed in square brackets. If the field has zero values then only the square brackets ("[]") are written. The last may optionally be followed by a comma. If the field has exactly one value, the brackets may be omitted and just the value written. For example, all of the following are valid for a multiple-valued field containing the single integer value 1:

1

[1,]

[1]

SFBitMask

A single-valued field that contains a mask of bit flags. Nodes that use this field class define mnemonic names for the bit flags. SFBitMasks are written to file as one or more mnemonic enumerated type names, in this format:

(flag1 | flag2 | ...)

If only one flag is used in a mask, the parentheses are optional. These names differ among uses of this field in various node classes.

SFBool

A field containing a single boolean (true or false) value. SFBools may be written as 0 (representing FALSE), 1, TRUE, or FALSE.

SFColor

A single-valued field containing a color. SFColors are written to file as an RGB triple of floating point numbers in standard scientific notation, in the range 0.0 to 1.0. [see change #16]

SFEnum

A single-valued field that contains an enumerated type value. Nodes that use this field class define mnemonic names for the values. SFEnums are written to file as a mnemonic enumerated type name. The name differs among uses of this field in various node classes.

SFFloat

A field that contains one single-precision floating point number. SFFloats are written to file in standard scientific notation.

SFImage

A field that contains an uncompressed 2-dimensional color or grayscale image.

SFImages are written to file as three integers representing the width, height, and number of components in the image, followed by width*height hexadecimal values representing the pixels in the image, separated by whitespace. A one-component image will have one-byte hexadecimal values representing the intensity of the image. For example, 0xFF is full intensity, 0x00 is no intensity. A two-component image puts the intensity in the first (high) byte and the transparency in the second (low) byte.

Pixels in a three-component image have the red component in the first (high) byte, followed by the green and blue components (so 0xFF0000 is red). Four-component images put the transparency byte after red/green/blue (so 0x0000FF80 is semitransparent blue). A value of 1.0 is completely transparent, 0.0 is completely opaque. Note: Each pixel is actually read as a single unsigned number, so a 3-component pixel with value "0x0000FF" can also be written as "0xFF" or "255" (decimal). Pixels are specified from left to right, bottom to top. The first hexadecimal value is the lower left pixel of the image, and the last value is the upper right pixel.

For example,

1 2 1 0xFF 0x00

is a 1-pixel-wide by 2-pixel-high grayscale image, with the bottom pixel white and the top pixel black. And:

2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFFF 0xFFFFF00

is a 2-pixel-wide by 4-pixel-high RGB image, with the bottom left pixel red, the bottom right pixel green, the two middle rows of pixels black, the top left pixel white, and the top right pixel yellow.

SFLong

A field containing a single long (32-bit) integer. SFLongs are written to file as an integer in decimal, hexadecimal (beginning with "0x"), or octal (beginning with "0") format.

SFMatrix

A field containing a transformation matrix. SFMatrices are written to file in row-major order as 16 floating point numbers separated by whitespace. For example, a matrix expressing a translation of 7.3 units along the X axis is written as:

1 0 0 0 0 1 0 0 0 0 1 0 7.3 0 0 1

SFRotation

A field containing an arbitrary rotation. SFRotations are written to file as four floating point values separated by whitespace. The four values represent an axis of rotation followed by the amount of right-handed rotation about that axis, in radians. For example, a 180-degree rotation about the Y axis is:

0 1 0 3.14159265

SFString

A field containing an ASCII string (sequence of characters). SFStrings are written to file as a sequence of ASCII characters in double quotes (optional if the string doesn't contain any whitespace). Any characters (including newlines) may appear within the quotes. To include a double quote character within the string, precede it with a backslash. For example:

Testing

"One, Two, Three"

"He said, \"Immel did it!\""

are all valid strings.

SFVec2f

Field containing a two-dimensional vector. SFVec2fs are written to file as a pair of floating point values separated by whitespace.

SFVec3f

Field containing a three-dimensional vector. SFVec3fs are written to file as three floating point values separated by whitespace.

MFColor

A multiple-value field that contains any number of RGB colors. MFColors are written to file as one or more RGB triples of floating point numbers in standard scientific notation. When more than one value is present, all of the values must be enclosed in square brackets and separated by commas. For example:

[1.0 0.0 0.0, 0 1 0, 0 0 1]

represents the three colors red, green, and blue.

MFLong

A multiple-value field that contains any number of long (32-bit) integers.

MFLongs are written to file as one or more integer values, in decimal, hexadecimal, or octal format. When more than one value is present, all the values are enclosed in square brackets and separated by commas; for example:

[17, -0xE20, -518820]

MFVec2f

A multiple-value field that contains any number of two-dimensional vectors. MFVec2fs are written to file as one or more pairs of floating point values separated by whitespace. When more than one value is present, all of the values are enclosed in square brackets and separated by commas; for example:

[0 0, 1.2 3.4, 98.6 -4e1]

MFVec3f

A multiple-value field that contains any number of three-dimensional vectors. MFVec3fs are written to file as one or more triples of floating point values separated by whitespace. When more than one value is present, all of the values are enclosed in square brackets and separated by commas; for example:

[0 0 0, 1.2 3.4 5.6, 98.6 -4e1 212]

[see change #11] [see change #12]

Nodes

VRML defines several different classes of nodes. Most of the nodes can be classified into one of three categories; shape, property, or group.

Shape nodes define the geometry in the scene. Conceptually, they are the only nodes that draw anything. Property nodes affect the way shapes are drawn.

And grouping nodes gather other nodes together, allowing collections of nodes to be treated as a single object. Some group nodes also control whether or not their children are drawn. [see change #18]

Nodes may contain zero or more fields. Each node type defines the type, name, and default value for each of its fields. The default value for the field is used if a value for the field is not specified in the VRML file.

The order in which the fields of a node are read is not important; for example, "Cube { width 2 height 4 depth 6 }" and "Cube { height 4 depth 6 width 2 }" are equivalent.

Here are the 36 nodes grouped by type.

The first group are the shape nodes.

- AsciiText
- Cone
- Cube
- Cylinder
- IndexedFaceSet
- IndexedLineSet
- PointSet
- Sphere

The second group are the properties. These can be further grouped into properties of the geometry and its appearance, matrix or transform properties, and cameras and lights

Appearance nodes:

- Coordinate3
- FontStyle
- Info
- LOD [not really...]
- Material
- MaterialBinding
- Normal
- NormalBinding
- ShapeHints
- Texture2
- Texture2Transform
- TextureCoordinate2
- Transformation nodes:
 - MatrixTransform
 - Rotation
 - Scale
 - Transform
 - Translation

Camera nodes:

- OrthographicCamera
- PerspectiveCamera

Light nodes:

- DirectionalLight
- PointLight
- SpotLight

Grouping nodes:

- Group [see note #29]
- Separator
- Switch
- TransformSeparator [see note #30]
- WWWAnchor

Finally, the WWWInline node does not fit neatly into any category.

AsciiText

This node represents strings of text characters from the ASCII coded character set. The first string is rendered with its baseline at (0,0,0). All subsequent strings advance y by $\text{ó}(\text{size} * \text{spacing})$. See FontStyle for a description of the size field. The justification field determines the placement of the strings in the x dimension. LEFT (the default) places the left edge of each string at x=0. CENTER places the center of each string at x=0. RIGHT places the right edge of each string at x=0. Text is rendered from left to right, top to bottom in the font set by FontStyle. The width field defines a suggested width constraint for each string. The default is to use the natural width of each string. Setting any value to 0 indicates the natural width should be used for that string.

The text is transformed by the current cumulative transformation and is drawn with the current material and texture.

Textures are applied to 3D text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right. The T origin can occur anywhere along each character, depending on how that character's outline is defined.

JUSTIFICATION

LEFT Align left edge of text to origin

CENTER Align center of text to origin

RIGHT Align right edge of text to origin

FILE FORMAT/DEFAULTS

AsciiText {

string "" # MFString

spacing 1 # SFFloat

justification LEFT # SFEnum

width 0 # MFFloat

}

Cone

This node represents a simple cone whose central axis is aligned with the Y axis. By default, the cone is centered at (0,0,0) and has a size of -1 to +1 in all three directions. The cone has a radius of 1 at the bottom and a height of 2, with its apex at 1 and its bottom at -1. The cone has two parts: the sides and the bottom.

The cone is transformed by the current cumulative transformation and is drawn with the current texture and material.

If the current material binding is PER_PART or PER_PART_INDEXED, the first current material is used for the sides of the cone, and the second is used for the bottom. Otherwise, the first material is used for the entire cone.

When a texture is applied to a cone, it is applied differently to the sides and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cone. The texture has a vertical seam at the back, intersecting the yz-plane. For the bottom, a circle is cut out of the texture square and applied to the cone's base circle. The texture appears right side up when the top of the cone is rotated towards the -Z axis.

PARTS

SIDES The conical part

BOTTOM The bottom circular face

ALL All parts

FILE FORMAT/DEFAULTS

Cone {

parts ALL # SFBitMask

bottomRadius 1 # SFFloat

height 2 # SFFloat

}

Coordinate3

This node defines a set of 3D coordinates to be used by a subsequent IndexedFaceSet, IndexedLineSet, or PointSet node. This node does not produce a visible result during rendering; it simply replaces the current coordinates in the rendering state for subsequent nodes to use.

FILE FORMAT/DEFAULTS

Coordinate3 {

point 0 0 0 # MFVec3f

}

Cube

This node represents a cuboid aligned with the coordinate axes. By default, the cube is centered at (0,0,0) and measures 2 units in each dimension, from -1 to +1. The cube is transformed by the current cumulative transformation and is drawn with the current material and texture.

If the current material binding is PER_PART, PER_PART_INDEXED, PER_FACE, or PER_FACE_INDEXED, materials will be bound to the faces of the cube in this order: front (+Z), back (-Z), left (-X), right (+X), top (+Y), and bottom (-Y). [see change #14]

Textures are applied individually to each face of the cube; the entire texture goes on each face. On the front, back, right, and left sides of the cube, the texture is applied right side up. On the top, the texture appears right side up when the top of the cube is tilted toward the camera. On the bottom, the texture appears right side up when the top of the cube is tilted towards the -Z axis.

FILE FORMAT/DEFAULTS

Cube {

width 2 # SFFloat

height 2 # SFFloat

depth 2 # SFFloat

}

Cylinder

This node represents a simple capped cylinder centered around the Y axis. By default, the cylinder is centered at (0,0,0) and has a default size of -1 to +1 in all three dimensions. The cylinder has three parts: the sides, the top (y = +1), and the bottom (y = -1). You can use the radius and height fields to create a cylinder with a different size.

The cylinder is transformed by the current cumulative transformation and is drawn with the current material and texture.

If the current material binding is PER_PART or PER_PART_INDEXED, the first current material is used for the sides of the cylinder, the second is used for the top, and the third is used for the bottom. Otherwise, the first material is used for the entire cylinder.

When a texture is applied to a cylinder, it is applied differently to the sides, top, and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cylinder. The texture has a vertical seam at the back, intersecting the yz-plane. For the top and bottom, a circle is cut out of the

texture square and applied to the top or bottom circle. The top texture appears right side up when the top of the cylinder is tilted toward the +Z axis, and the bottom texture appears right side up when the top of the cylinder is tilted toward the -Z axis.

PARTS

SIDES The cylindrical part

TOP The top circular face

BOTTOM The bottom circular face

ALL All parts

FILE FORMAT/DEFAULTS

Cylinder {

parts ALL # SFBitMask

radius 1 # SFFloat

height 2 # SFFloat

}

DirectionalLight

This node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. [see change #3]

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator does not affect any objects outside that separator. [see change #31]

FILE FORMAT/DEFAULTS

DirectionalLight {

on TRUE # SFBool

intensity 1 # SFFloat

color 1 1 1 # SFColor

direction 0 0 -1 # SFVec3f

}

FontStyle

This node defines the current font style used for all subsequent AsciiText. Font attributes only are defined. It is up to the browser to assign specific fonts to the various attribute combinations. The size field specifies the height (in object space units) of glyphs rendered and determines the vertical spacing of adjacent lines of text.

FAMILY

SERIF Serif style (such as TimesRoman)

SANS Sans Serif Style (such as Helvetica)

TYPEWRITER Fixed pitch style (such as Courier)

STYLE

NONE No modifications to family

BOLD Embolden family

ITALIC Italicize or Slant family

FILE FORMAT/DEFAULTS

FontStyle {

size 10 # SFFloat

family SERIF # SFEnum

style NONE # SFBitMask

}

Group

[see change #29]. This node defines the base class for all group nodes. Group is a node that contains an ordered list of child nodes. This node is simply a container for the child nodes and does not alter the traversal state in any way. During traversal, state accumulated for a child is passed on to each successive child and then to the parents of the group (Group does not push or pop traversal state as separator does).

FILE FORMAT/DEFAULTS

Group {

}

IndexedFaceSet

This node represents a 3D shape formed by constructing faces (polygons) from vertices located at the current coordinates. IndexedFaceSet uses the indices in its coordIndex field to specify the polygonal faces. An index of -1 indicates that the current face has ended and the next one begins.

The vertices of the faces are transformed by the current transformation matrix.

Treatment of the current material and normal binding is as follows:

PER_PART and PER_FACE specify a material or normal for each face. PER_VERTEX specifies a material or normal for each vertex. The corresponding _INDEXED bindings are the same, but use the materialIndex or normalIndex indices. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX_INDEXED; if insufficient normals exist in the state, vertex normals will be generated automatically.

Explicit texture coordinates (as defined by TextureCoordinate2) may be bound to vertices of an indexed shape by using the indices in the textureCoordIndex field. As with all vertex-based shapes, if there is a current texture but no texture coordinates are specified, a default texture coordinate mapping is calculated using the bounding box of the shape. The longest dimension of the bounding box defines the S coordinates, and the next longest defines the T coordinates. The value of the S coordinate ranges from 0 to 1, from one end of the bounding box to the other. The T coordinate ranges between 0 and the ratio of the second greatest dimension of the bounding box to the greatest dimension.

Be sure that the indices contained in the coordIndex, materialIndex, normalIndex, and textureCoordIndex fields are valid with respect to the current state, or errors will occur.

FILE FORMAT/DEFAULTS

IndexedFaceSet {

coordIndex 0 # MFLong

materialIndex -1 # MFLong

normalIndex -1 # MFLong

textureCoordIndex -1 # MFLong

}

IndexedLineSet

This node represents a 3D shape formed by constructing polylines from vertices located at the current coordinates. IndexedLineSet uses the indices in its coordIndex field to specify the polylines. An index of -1 indicates that the current polyline has ended and the next one begins.

The coordinates of the line set are transformed by the current cumulative transformation.

Treatment of the current material and normal binding is as follows: The PER_PART binding specifies a material or normal for each segment of the line. The PER_FACE binding specifies a material or normal

for each poly-line. PER_VERTEX specifies a material or normal for each vertex. The corresponding _INDEXED bindings are the same, but use the materialIndex or normalIndex indices. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX_INDEXED; if insufficient normals exist in the state, the lines will be drawn unlit. The same rules for texture coordinate generation as IndexedFaceSet are used.

FILE FORMAT/DEFAULTS

IndexedLineSet {

coordIndex 0 # MFLong

materialIndex -1 # MFLong

normalIndex -1 # MFLong

textureCoordIndex -1 # MFLong

}

Info

This class defines an information node in the scene graph. This node has no effect during traversal. It is used to store information in the scene graph, typically for application-specific purposes, copyright messages, or other strings.

Info {

string "<Undefined info>" # SFString

}

LOD

This group node is used to allow applications to switch between various representations of objects automatically. The children of this node typically represent the same object or objects at varying levels of detail, from highest detail to lowest.

The specified center point of the LOD is transformed by the current transformation into world space, and the distance from the transformed center to the world-space eye point is calculated. If the distance is less than the first value in the ranges array, then the first child of the LOD group is drawn. If between the first and second values in the ranges array, the second child is drawn, etc. If there are N values in the ranges array, the LOD group should have N+1 children. Specifying too few children will result in the last child being used repeatedly for the lowest levels of detail; if too many children are specified, the extra children will be ignored. Each value in the ranges array should be less than the previous value, otherwise results are undefined. [see change #6] [see change #29]

FILE FORMAT/DEFAULTS

LOD {

```
range [ ] # MFFloat  
center 0 0 0 # SFVec3f  
}
```

Material

This node defines the current surface material properties for all subsequent shapes. Material sets several components of the current material during traversal. Different shapes interpret materials with multiple values differently. To bind materials to shapes, use a MaterialBinding node. [see change #2] [see change #8] [see change #13]

FILE FORMAT/DEFAULTS

```
Material {  
ambientColor 0.2 0.2 0.2 # MFColor  
diffuseColor 0.8 0.8 0.8 # MFColor  
specularColor 0 0 0 # MFColor  
emissiveColor 0 0 0 # MFColor  
shininess 0.2 # MFFloat  
transparency 0 # MFFloat  
}
```

MaterialBinding

This node specifies how the current materials are bound to shapes that follow in the scene graph. Each shape node may interpret bindings differently. The current material always has a base value, which is defined by the first value of all material fields. Since material fields may have multiple values, the binding determines how these values are distributed over a shape.

The bindings for faces and vertices are meaningful only for shapes that are made from faces and vertices. Similarly, the indexed bindings are only used by the shapes that allow indexing.

When multiple material values are bound, the values are cycled through, based on the period of the material component with the most values [see change #4]. For example, the following table shows the values used when cycling through (or indexing into) a material with 2 ambient colors, 3 diffuse colors, and 1 of all other components in the current material. (The period of this material cycle is 3):

Material	Ambient Color	Diffuse Color	Other
0	0	0	0
1	1	1	0
2	1	2	0

3 (same as 0)	0	0	0

BINDINGS

DEFAULT Use default binding

OVERALL Whole object has same material

PER_PART One material for each part of object

PER_PART_INDEXED One material for each part, indexed

PER_FACE One material for each face of object

PER_FACE_INDEXED One material for each face, indexed

PER_VERTEX One material for each vertex of object [see change #5]

PER_VERTEX_INDEXED One material for each vertex, indexed [see change #5]

FILE FORMAT/DEFAULTS

MaterialBinding {

value DEFAULT # SFEnum

}

MatrixTransform

This node defines a geometric 3D transformation with a 4-by-4 matrix. Note that some matrices (such as singular ones) may result in errors. [see change #7]

FILE FORMAT/DEFAULTS

MatrixTransform {

matrix 1 0 0 0 # SFMatrix

0 1 0 0

0 0 1 0

0 0 0 1

}

Normal

This node defines a set of 3D surface normal vectors to be used by vertex-based shape nodes (IndexedFaceSet, IndexedLineSet, PointSet) that follow it in the scene graph. This node does not produce a visible result during rendering; it simply replaces the current normals in the rendering state for

subsequent nodes to use. This node contains one multiple-valued field that contains the normal vectors.

FILE FORMAT/DEFAULTS

Normal {

vector 0 0 1 # MFVec3f

}

NormalBinding

This node specifies how the current normals are bound to shapes that follow in the scene graph. Each shape node may interpret bindings differently.

The bindings for faces and vertices are meaningful only for shapes that are made from faces and vertices. Similarly, the indexed bindings are only used by the shapes that allow indexing. For bindings that require multiple normals, be sure to have at least as many normals defined as are necessary; otherwise, errors will occur.

BINDINGS

DEFAULT Use default binding

OVERALL Whole object has same normal

PER_PART One normal for each part of object

PER_PART_INDEXED One normal for each part, indexed

PER_FACE One normal for each face of object

PER_FACE_INDEXED One normal for each face, indexed

PER_VERTEX One normal for each vertex of object

PER_VERTEX_INDEXED One normal for each vertex, indexed

FILE FORMAT/DEFAULTS

NormalBinding {

value DEFAULT # SFEnum

}

Orthographic Camera

An orthographic camera defines a parallel projection from a viewpoint. This camera does not diminish objects with distance, as a perspective camera does. The viewing volume for an orthographic camera is a rectangular parallelepiped (a box).

By default, the camera is located at (0,0,1) and looks along the negative Z axis; the position and

orientation fields can be used to change these values. The height field defines the total height of the viewing volume.

A camera can be placed in a VRML world to specify the initial location of the viewer when that world is entered. VRML browsers will typically modify the camera to allow a user to move through the virtual world.

Cameras are affected by the current transformation, so you can position a camera by placing a transformation node before it in the scene graph. The default position and orientation of a camera is at (0,0,1) looking along the negative Z axis.

[see change #9] [see change #26]

FILE FORMAT/DEFAULTS

OrthographicCamera {

position 0 0 1 # SFVec3f

orientation 0 0 1 0 # SFRotation

focalDistance 5 # SFFloat

height 2 # SFFloat

}

Perspective Camera

A perspective camera defines a perspective projection from a viewpoint. The viewing volume for a perspective camera is a truncated right pyramid.

By default, the camera is located at (0,0,1) and looks along the negative Z axis; the position and orientation fields can be used to change these values. The heightAngle field defines the total vertical angle of the viewing volume.

See more on cameras in the OrthographicCamera description.

[see change #9] [see change #26]

FILE FORMAT/DEFAULTS

PerspectiveCamera {

position 0 0 1 # SFVec3f

orientation 0 0 1 0 # SFRotation

focalDistance 5 # SFFloat

heightAngle 0.785398 # SFFloat

}

PointLight

This node defines a point light source at a fixed 3D location. A point source illuminates equally in all directions; that is, it is omni-directional. [see change #3]

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator does not affect any objects outside that separator. [see change #31]

FILE FORMAT/DEFAULTS

```
PointLight {  
on TRUE # SFBool  
intensity 1 # SFFloat  
color 1 1 1 # SFColor  
location 0 0 1 # SFVec3f  
}
```

PointSet

This node represents a set of points located at the current coordinates. PointSet uses the current coordinates in order, starting at the index specified by the startIndex field. The number of points in the set is specified by the numPoints field. A value of -1 for this field indicates that all remaining values in the current coordinates are to be used as points.

The coordinates of the point set are transformed by the current cumulative transformation. The points are drawn with the current material and texture.

Treatment of the current material and normal binding is as follows: PER_PART, PER_FACE, and PER_VERTEX bindings bind one material or normal to each point. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX. The startIndex is also used for materials or normals when the binding indicates that they should be used per vertex.

FILE FORMAT/DEFAULTS

```
PointSet {  
startIndex 0 # SFLong  
numPoints -1 # SFLong  
}
```


Rotation

This node defines a 3D rotation about an arbitrary axis through the origin. The rotation is accumulated into the current transformation, which is applied to subsequent shapes.

FILE FORMAT/DEFAULTS

Rotation {

rotation 0 0 1 0 # SFRotation

}

See rotation field description for more information.

Scale

This node defines a 3D scaling about the origin. If the components of the scaling vector are not all the same, this produces a nonuniform scale.

FILE FORMAT/DEFAULTS

Scale {

scaleFactor 1 1 1 # SFVec3f

}

Separator

This group node performs a push (save) of the traversal state before traversing its children and a pop (restore) after traversing them. This isolates the separator's children from the rest of the scene graph. A separator can include lights, cameras, coordinates, normals, bindings, and all other properties. [see change #32]

Separators can also perform render culling. Render culling skips over traversal of the separator's children if they are not going to be rendered, based on the comparison of the separator's bounding box with the current view volume. Culling is controlled by the renderCulling field. These are set to AUTO by default, allowing the implementation to decide whether or not to cull.

CULLING ENUMS

ON Always try to cull to the view volume

OFF Never try to cull to the view volume

AUTO Implementation-defined culling behavior

FILE FORMAT/DEFAULTS

Separator {

renderCulling AUTO # SFEnum

}

ShapeHints

The ShapeHints node indicates that IndexedFaceSets are solid, contain ordered vertices, or contain convex faces. These hints allow VRML implementations to optimize certain rendering features. Optimizations that may be performed include enabling backface culling and disabling two-sided lighting. For example, if an object is solid and has ordered vertices, an implementation may turn on backface culling and turn off two-sided lighting. If the object is not solid but has ordered vertices, it may turn off backface culling and turn on two-sided lighting.

The ShapeHints node also affects how default normals are generated. When an IndexedFaceSet has to generate default normals, it uses the creaseAngle field to determine which edges should be smoothly shaded and which ones should have a sharp crease. The crease angle is the angle between surface normals on adjacent polygons. For example, a crease angle of .5 radians (the default value) means that an edge between two adjacent polygonal faces will be smooth shaded if the normals to the two faces form an angle that is less than .5 radians (about 30 degrees). Otherwise, it will be faceted.

VERTEX ORDERING ENUMS

UNKNOWN_ORDERING Ordering of vertices is unknown

CLOCKWISE Face vertices are ordered clockwise

(from the outside)

COUNTERCLOCKWISE Face vertices are ordered counterclockwise

(from the outside)

SHAPE TYPE ENUMS

UNKNOWN_SHAPE_TYPE Nothing is known about the shape

SOLID The shape encloses a volume

FACE TYPE ENUMS

UNKNOWN_FACE_TYPE Nothing is known about faces

CONVEX All faces are convex

FILE FORMAT/DEFAULTS

ShapeHints {

vertexOrdering UNKNOWN_ORDERING # SFEnum [see change #34]

shapeType UNKNOWN_SHAPE_TYPE # SFEnum [see change #34]

faceType CONVEX # SFEnum

creaseAngle 0.5 # SFFloat

}

Sphere

This node represents a sphere. By default, the sphere is centered at the origin and has a radius of 1. The sphere is transformed by the current cumulative transformation and is drawn with the current material and texture.

A sphere does not have faces or parts. Therefore, the sphere ignores material and normal bindings, using the first material for the entire sphere and using its own normals. When a texture is applied to a sphere, the texture covers the entire surface, wrapping counterclockwise from the back of the sphere. The texture has a seam at the back on the yz-plane.

FILE FORMAT/DEFAULTS

Sphere {

radius 1 # SFFloat

}

SpotLight

This node defines a spotlight light source. A spotlight is placed at a fixed location in three-space and illuminates in a cone along a particular direction. The intensity of the illumination drops off exponentially as a ray of light diverges from this direction toward the edges of the cone. The rate of drop-off and the angle of the cone are controlled by the dropOffRate and cutOffAngle fields. [see change #3]

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator does not affect any objects outside that separator. [see change #31]

FILE FORMAT/DEFAULTS

SpotLight {

on TRUE # SFBool

intensity 1 # SFFloat

color 1 1 1 # SFVec3f

location 0 0 1 # SFVec3f

direction 0 0 -1 # SFVec3f

dropOffRate 0 # SFFloat

cutOffAngle 0.785398 # SFFloat

```
}
```

Switch

This group node traverses one, none, or all of its children. One can use this node to switch on and off the effects of some properties or to switch between different properties.

The `whichChild` field specifies the index of the child to traverse, where the first child has index 0.

A value of -1 (the default) means do not traverse any children. A value of -3 traverses all children [see change #33], making the switch behave exactly like a regular Group. [see change #28]

FILE FORMAT/DEFAULTS

```
Switch {
```

```
whichChild -1 # SFLong
```

```
}
```

Texture2

This property node defines a texture map and parameters for that map. This map is used to apply texture to subsequent shapes as they are rendered.

The texture can be read from the URL [see change #35] specified by the `filename` field. To turn off texturing, set the `filename` field to an empty string (`""`). [see change #1]

Textures can also be specified inline by setting the `image` field to contain the texture data. Specifying both a URL and data inline will result in undefined behavior.

WRAP ENUM

REPEAT Repeats texture outside 0-1 texture coordinate range

CLAMP Clamps texture coordinates to lie within 0-1 range

FILE FORMAT/DEFAULTS

```
Texture2 {
```

```
filename "" # SFString
```

```
image 0 0 0 # SFImage
```

```
wrapS REPEAT # SFEnum
```

```
wrapT REPEAT # SFEnum
```

```
}
```

Texture2Transform

This node defines a 2D transformation applied to texture coordinates. This affects the way textures are applied to the surfaces of subsequent shapes.

The transformation consists of (in order) a nonuniform scale about an arbitrary center point, a rotation about that same point, and a translation. This allows a user to change the size and position of the textures on shapes.

FILE FORMAT/DEFAULTS

```
Texture2Transform {  
translation 0 0 # SFVec2f  
rotation 0 # SFFloat  
scaleFactor 1 1 # SFVec2f  
center 0 0 # SFVec2f  
}
```

TextureCoordinate2

This node defines a set of 2D coordinates to be used to map textures to the vertices of subsequent PointSet, IndexedLineSet, or IndexedFaceSet objects. It replaces the current texture coordinates in the rendering state for the shapes to use.

Texture coordinates range from 0 to 1 across the texture. The horizontal coordinate, called S, is specified first, followed by the vertical coordinate, T.

FILE FORMAT/DEFAULTS

```
TextureCoordinate2 {  
point 0 0 # MFVec2f  
}
```

Transform

This node defines a geometric 3D transformation consisting of (in order) a (possibly) nonuniform scale about an arbitrary point, a rotation about an arbitrary point and axis, and a translation.

FILE FORMAT/DEFAULTS

```
Transform {  
translation 0 0 0 # SFVec3f  
rotation 0 0 1 0 # SFRotation
```

scaleFactor 1 1 1 # SFVec3f

scaleOrientation 0 0 1 0 # SFRotation

center 0 0 0 # SFVec3f

}

The transform node

Transform {

translation T1

rotation R1

scaleFactor S

scaleOrientation R2

center T2

}

is equivalent to the sequence

Translation { translation T1 }

Translation { translation T2 }

Rotation { rotation R1 }

Rotation { rotation R2 }

Scale { scaleFactor S }

Rotation { rotation -R2 }

Translation { translation -T2 }

TransformSeparator

[see change #30]. This group node is similar to the separator node in that it saves state before traversing its children and restores it afterwards. However, it saves only the current transformation; all other state is left as is. This node can be useful for positioning a camera, since the transformations to the camera will not affect the rest of the scene, even though the camera will view the scene. Similarly, this node can be used to isolate transformations to light sources or other objects.

FILE FORMAT/DEFAULTS

TransformSeparator {

}

Translation

This node defines a translation by a 3D vector.

FILE FORMAT/DEFAULTS

Translation {

translation 0 0 0 # SFVec3f

}

WWWAnchor

The WWWAnchor group node loads a new scene into a VRML browser when one of its children is chosen [see change #22]. Exactly how a user "chooses" a child of the WWWAnchor is up to the VRML browser; typically, clicking on one of its children with the mouse will result in the new scene replacing the current scene. A WWWAnchor with an empty ("") name does nothing when its children are chosen. The name is an arbitrary URL. [see change #35]

WWWAnchor behaves like a Separator, pushing the traversal state before traversing its children and popping it afterwards.

The description field in the WWWAnchor allows for a friendly prompt to be displayed as an alternative to the URL in the name field. Ideally, browsers will allow the user to choose the description, the URL, or both to be displayed for a candidate WWWAnchor.

The WWWAnchor's map field is an enumerated value that can be either NONE (the default) or POINT. If it is POINT, then the object-space coordinates of the point on the object the user chose will be added to the URL in the name field, with the syntax "?x,y,z."

MAP ENUM

NONE Do not add information to the URL

POINT Add object-space coordinates to URL

FILE FORMAT/DEFAULTS

WWWAnchor {

name "" # SFString

description "" # SFString

map NONE # SFEnum

}

WWWInline

The WWWInline node reads its children from anywhere in the World Wide Web. Exactly when its children are read is not defined; reading the children may be delayed until the WWWInline is actually displayed. A WWWInline with an empty name does nothing. The name is an arbitrary URL. [see change #35]

The effect of referring to a non-VRML URL in a WWWInline node is undefined.

If the WWWInline's bboxSize field specifies a nonempty bounding box (a bounding box is nonempty if at least one of its dimensions is greater than zero), then the WWWInline's object-space bounding box is specified by its bboxSize and bboxCenter fields [see change #21]. This allows an implementation to view-volume cull or LOD switch the WWWInline without reading its contents.

FILE FORMAT/DEFAULTS

WWWInline {

name "" # SFString

bboxSize 0 0 0 # SFVec3f

bboxCenter 0 0 0 # SFVec3f

}

Instancing

A node may be the child of more than one group. This is called "instancing" (using the same instance of a node multiple times, called "aliasing" or "multiple references" by other systems), and is accomplished by using the "USE" keyword.

The DEF keyword both defines a named node, and creates a single instance of it. The USE keyword indicates that the most recently defined instance should be used again. If several nodes were given the same name, then the last DEF encountered during parsing "wins." DEF/USE is limited to a single file; there is no mechanism for USEing nodes that are DEFed in other files. [see change #23]

A name goes into scope as soon as the DEF is encountered, and does not go out of scope until another DEF of the same name or end-of-file are encountered. Nodes cannot be shared between files (you cannot USE a node that was DEFed inside the file to which a WWWInline refers).

For example, rendering this scene will result in three spheres being drawn. Both of the spheres are named "Joe"; the second (smaller) sphere is drawn twice:

Separator {

DEF Joe Sphere { }

Translation { translation 2 0 0 }

Separator {

DEF Joe Sphere { radius .2 }

Translation { translation 2 0 0 }

}

USE Joe # radius .2 sphere will be used here

Extensibility

Extensions to VRML are supported by supporting self-describing nodes. Nodes that are not part of standard VRML must write out a description of their fields first, so that all VRML implementations are able to parse and ignore the extensions. [see change #18] [see change #19]

This description is written just after the opening curly brace for the node, and consists of the keyword "fields" followed by a list of the types and names of fields used by that node, all enclosed in square brackets and separated by commas. For example, if Cube was not a standard VRML node, it would be written like this:

Cube {

fields [SFFloat width, SFFloat height, SFFloat depth]

width 10 height 4 depth 3

}

Specifying the fields for nodes that ARE part of standard VRML is not an error; VRML parsers must silently ignore the field specification.

Is-a Relationships

A new node type may also be a superset of an existing node that is part of the standard. In this case, if an implementation for the new node type cannot be found the new node type can be safely treated as the existing node it is based on (with some loss of functionality, of course). To support this, new node types can define an MFString field called "isA" containing the names of the types of which it is a superset. For example, a new type of Material called "ExtendedMaterial" that adds index of refraction as a material property can be written as:

ExtendedMaterial {

fields [MFString isA, MFFloat indexOfRefraction,

MFCOLOR ambientColor, MFCOLOR diffuseColor,

MFCOLOR specularColor, MFCOLOR emissiveColor,

MFFloat shininess, MFFloat transparency]

```
isA [ "Material" ]  
  
indexOfRefraction .34  
  
diffuseColor .8 .54 1  
  
}
```

Multiple is-a relationships may be specified in order of preference; implementations are expected to use the first for which there is an implementation. [see change #24]

An Example

This is a longer example of a VRML scene. It contains a simple model of a track light consisting of primitive shapes, plus three walls (built out of polygons) and a reference to a shape defined elsewhere, both of which are illuminated by a spotlight. The shape acts as a hyperlink to some HTML text.

```
#VRML V1.0 ascii  
  
Separator {  
  
Separator { # Simple track-light geometry:  
  
Translation { translation 0 4 0 }  
  
Separator {  
  
Material { emissiveColor 0.1 0.3 0.3 }  
  
Cube {  
  
width 0.1  
  
height 0.1  
  
depth 4  
  
}  
  
}  
  
Rotation { rotation 0 1 0 1.57079 }  
  
Separator {  
  
Material { emissiveColor 0.3 0.1 0.3 }  
  
Cylinder {  
  
radius 0.1  
  
height .2
```

}

}

Rotation { rotation -1 0 0 1.57079 }

Separator {

Material { emissiveColor 0.3 0.3 0.1 }

Rotation { rotation 1 0 0 1.57079 }

Translation { translation 0 -.2 0 }

Cone {

height .4

bottomRadius .2

}

Translation { translation 0 .4 0 }

Cylinder {

radius 0.02

height .4

}

}

}

SpotLight { # Light from above

location 0 4 0

direction 0 -1 0

intensity 0.9

cutOffAngle 0.7

}

Separator { # Wall geometry; just three flat polygons

Coordinate3 {

point [

-2 0 -2, -2 0 2, 2 0 2, 2 0 -2,

-2 4 -2, -2 4 2, 2 4 2, 2 4 -2]

}

IndexedFaceSet {

coordIndex [0, 1, 2, 3, -1,

0, 4, 5, 1, -1,

0, 3, 7, 4, -1

]

}

}

WWWAnchor { # A hyperlinked cow:

name "http://www.foo.edu/CowProject/AboutCows.html"

Separator {

Translation { translation 0 1 0 }

WWWInline { # Reference another object

name "http://www.foo.edu/3DObjects/cow.wrl"

}

}

}

}

Browser Considerations

This section describes the file-naming and MIME conventions to be used in building VRML browsers and configuring WWW browsers to work with them.

File Extensions

The file extension for VRML files is .wrl (for world).

MIME

The MIME type for VRML files is defined as follows:

x-world/x-vrml

The MIME major type for 3D world descriptions is x-world. The MIME minor type for VRML documents is x-vrml. Other 3D world descriptions, such as oogl for The Geometry Center's Object-Oriented Geometry Language, or iv, for SGI's Open Inventor ASCII format, can be supported by using different MIME minor types.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



Appendix C - Proposed Changes for VRML 1.1

The following are proposed changes to the VRML 1.0 specification. These have all been approved by the VAG (VRML Architecture Group) as listed in the minutes of the August 21-23, 1995 meeting. Please send any comments on these proposed changes to the [www-vrml mailing list \(www-vrml@wired.com\)](mailto:www-vrml@wired.com).

Throughout this list, the term "deprecated" means that the use of that feature is not recommended. It need not be supported by browsers, and should not be used in new VRML files.

1. The Texture2 node will require support for JPEG, with support for PNG recommended (in part because it has an alpha channel). GIF is not recommended, due to licensing restrictions.
2. A standard way of mapping a Material to a color and set of coefficients (ka,kd,ks) will be defined. This is for renderers that don't compute the color components separately.
3. The exact lighting equation to use will be specified. It will be noted that the ambient color should be ignored, since in practice it's the same as diffuse.
4. The idea of "cycling" in MaterialBinding will be deprecated and then dropped.
5. PER_VERTEX and PER_VERTEX_INDEXED material binding will only support diffuse color and transparency, and many browsers may not even support those except in unlit shading.
6. The LOD description will be clarified. The center point for the range calculations is [0,0,0] in object coordinates. Renderers may adjust the ranges based on performance (e.g., to maintain a constant frame rate), so LOD should not be used to simulate the effect of behavior. LOD can read a WWWInline at any time. The range values should be transformed, effectively transforming the camera position into object coordinates.
7. The MatrixTransform node can only be used to specify a combination of rotation, translation, and scaling. Use of the final column of the matrix is not supported. All matrices must be invertible.
8. In transparency, an alpha value of 0 is opaque, and 1 is transparent. Many browsers will not support partial transparency, and will treat alpha values greater than or equal 0.5 as 1.0 and values below 0.5 as 0.

9. The first camera specified in the file should be the default one; the others are available as other entry points into the scene. The camera's DEF'd name can be used with the URL of a WWWAnchor, separated from it by a #. The focalDistance can be used as a navigational hint as to how fast to move.
10. It will be explicitly stated that although the file format uses a scene graph metaphor, browsers are not required to maintain the scene graph in memory.
11. MFFloat and MFString will be added to the list of types (they were overlooked initially).
12. The SFNode type will be added.
13. Browsers should recognize materials with multiple emissive colors, and no specified diffuse or specular colors, as being "unlit." This removes the need for a LightModel node. Browsers are unlikely to support a mixture of lit and unlit materials in a single object.
14. In the Cube node, it will be made clear that width=x, height=y, and depth=z.
15. The # character is allowed in SFString, but not in node names.
16. In SFColor, the values will be in a Kernighan and Ritchie style scanf format.
17. The scope of Translate, Rotate, Scale, and Transform will be defined.
18. A clear convention on case for field names and node names is adopted, to allow parsers to tell when they've finished with fields and are now parsing children. Field names are fieldName, node names are NodeName.
19. Custom nodes and fields should have a name ending in EXT if used by more than one vendor, or FOOX (where FOO is a vendor name) for vendor-specific extensions.
20. The header will (of course) say 1.1 instead of 1.0; it will also say "utf8" instead of "ascii."
21. In the description of WWWInline, it will be made clear that the bboxCenter is just a hint as to where the center of the box will wind up; it is not to be used for translation.
22. In the description of WWWAnchor, it will be made clear that the "deepest" anchor wins.
23. It is strongly suggested that DEF'd names be kept unique.
24. Explicitly say that definitions for new nodes (done with "isA" and "fields") are file scoped; they only need to be specified the first time.
25. A new NavigationHints node, with translationSpeed in meters per second and a navigationMode of WALK, FLY, EXAMINE or DEFAULT:
26. Two new fields have been added to the Camera nodes: nearDistance and farDistance. The default values are 0.01, 5.00.
27. LOD nodes will be treated as Separators.
28. Switch nodes will be treated as Separators.

- 29. Group nodes will be deprecated, then dropped.
- 30. TransformSeparator nodes will be deprecated, then dropped.
- 31. Many browsers will not cause lights to stop at Separators.
- 32. Separator syntax will change; the basic idea is to have properties followed by geometry followed by children.
- 33. Switch nodes will be able to select any one of their children, or none of them, but not all of them.
- 34. The default settings in a ShapeHints node will be vertexOrdering COUNTERCLOCKWISE and shapeType SOLID.
- 35. Anywhere that an URL is specified, an URN should be valid as well; this includes the name fields of WWWInline and WWWAnchor and the filename fields of Texture2 and SoundEmitter.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



Appendix D - Proposed Changes for VRML 2.0

These are some of the changes that have been proposed for future versions of VRML. Some may make it into 1.1, others into 2.0, and some into an intermediate release that's currently being referred to as "1.x". This list is ever-changing; the current version can be found at <<http://sune.uwaterloo.ca/~broehl/vrml/changes.html>>.

Please send any comments on these proposed changes to the [www-vrml mailing list \(www-vrml@wired.com\)](mailto:www-vrml@wired.com).

These are grouped into a number of different categories.

Clarifications

1. It should be clearly stated that the matrix in a MatrixTransform node is column-oriented; that is, the translations are in the last row, not the last column. This is mentioned in the SFMatrix description, but is easily overlooked; it should be spelled out in the MatrixTransform node description.
2. The fact that transformation nodes are applied "backwards" (with the last listed transformation being applied first) should be explicitly stated.
3. For ascii files, the specific handling of non-printable ascii characters and characters with the top bit set should be spelled out. The characters 0x09 (tab), 0x0D (carriage return), 0x0A (newline) and 0x20 through 0x7E are all acceptable. The EOF character (0x1A, control Z) should be treated as an end-of-file; all characters following it should be ignored. Support for internationalization changes that, of course.

New Nodes

1. An "Environment" node:

```
Environment {ambientIntensity 0.0 # SFFloat
```

```

ambientColor 1.0 1.0 1.0 # SFCOLOR
attenuation 0.0 0.0 1.0 # SFVec3f
fogType NONE # enum (NONE, HAZE, FOG)
fogColor 0.0 0.0 0.0 # SFCOLOR
fogVisibility 0.0 # SFFloat
soundAttenuation 0.0 0.0 1.0 # SFVec3f}

```

The fogVisibility is the distance at which fog completely hides everything beyond it; if it's 0.0, the camera's farDistance value is used. Implementation of fog should be strictly optional. The attenuation coefficient is $1 / (A + B*d + C*d*d)$, where A, B and C are attenuation values for the Environment node and d is the distance from the light source. The soundAttenuation field works the same way.

2. A SoundEmitter node. SoundEmitters should behave the same ways shapes and lights do (i.e., their location should be affected by the current transform). A SoundEmitter node looks like this:

```

SoundEmitter {name "" # SFString
description "" # SFString
intensity 1.0 # SFFloat
location 0 0 0 # SFVec3f
direction 0 0 0 # SFVec3f
maxRangeOfEffect INF # SFFloat
minRangeOfEffect INF # SFFloat
angleOfEffect 2PI # SFFloat}

```

The description field provides a textual description of the sound; that way browsers that don't support sound have the option of displaying the description instead, probably in a separate window ("you hear a foghorn"). Even browsers that do support sound should have the text display option, for hearing-impaired users. As with texture maps, we should specify a minimum set of sound file formats that browsers must support; the proposed types are audio/basic, audio/x-aiff, audio/x-mpeg and MIDI.

Sound would be expected to be single-channel (in other words, monophonic), since any stereo placement would be handled by the browser based on the listener's location and orientation.

"Spot" sounds are specified by having an angleOfEffect, whereas "Point" sounds don't (they have the default angleOfEffect of 2PI, which means they radiate equally in all directions). Ambient sounds are specified in a peculiar way: they're located at the origin.

3. A MediaAttributes node:

```

MediaAttributes {speed 1.0 # SFFloat

```

```
looping NONE # SFEnum
startOffset 0 # SFFloat
endOffset NONE # SFFloat}
```

It's worth noting that the startOffset and endOffset are only really meaningful for sound samples. If our sample format is WAV (which has that information built-in) then the startOffset and endOffset fields aren't needed.

15. An ElevationGrid node, to define terrain:

```
ElevationGrid {nx 0 # SFLong
ny 0 # SFLong
step 1,1 # SFVec2f
heights [ ] # MFFloat}
```

The grid starts at 0,0,0 in object coordinates, goes positive X then negative Z.

A Sky node:

```
Sky {groundColor 0,0,0 # SFCOLOR
bottomSkyColor 0,0,0 # SFCOLOR
topSkyColor 0,0,0 # SFCOLOR
skyTexture "" # SFString}
```

6. A "Generalized cylinder" node.

7. A TextureCoordinateBinding node; it should only allow PER_VERTEX and PER_VERTEX_INDEXED binding.

8. A "Hint" node: This has been re-proposed, with a new format:

```
BrowserHint {hint "" # MFString}
```

This would contain a series of hints of the form "hintname value." This is an alternative to named Info nodes, and addresses the concerns about polluting the namespace.

9. A "ResetProperties" node that would set all the (non-transform) properties back to their default value:

```
ResetProperties {}
```

This would be more concise than having people specify ShapeHints, MaterialBindings, NormalBindings and Texture2Transform at the start of any file they think might get WWWInline'd.

New or Changed Fields

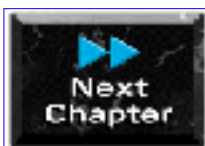
1. A transparencyColor field on the Texture2 node, for those file formats (such as JPEG) that do not provide an alpha channel. Alternatively, require the use of PNG instead of JPEG when an alpha channel is needed.
2. An apexRadius field for the Cylinder node. If not specified, it would default to being the same as the radius value. May not be worth implementing if a GeneralizedCylinder node is available.
3. A shapeConvex field in the ShapeHint node. It would be of type SFBool, with a default value of FALSE. If TRUE, it means the shapes which follow are known to be convex polyhedrons. Browsers can ignore this, of course. [This might be the same as shapeType SOLID.]
4. Change the map field of the WWWAnchor node to be an SFBitmask, with values of POINT, ORIENTATION and TEXTURE. POINT causes the camera's (object-space) location to be sent, ORIENTATION causes its (object-space) orientation to be sent, and TEXTURE causes its texture map coordinates to be sent. Regardless of the order in which the values are specified, they're always sent with the POINT first (if specified), the ORIENTATION next (if specified) and the TEXTURE coordinates last (if specified). The value NONE would also be accepted, in which case none of the extra information is sent.
5. A "tessellation" field in the ShapeHints node, giving the number of faces to use for tessellating a Sphere, Cone or Cylinder.
6. A "textureQuality" field in the ShapeHints node. It would be an SFEnum, with values of PERSPECTIVE and LINEAR, to indicate whether true, perspective-correct texture mapping is required or whether faster bi-linear can be used.

General

1. Allow WWWInline's "name" field to be an MFString, so it can have alternatives if the first one is unavailable.
2. A way of defining geometry without instancing it.
3. Support for animated texture maps. This could be done by allowing the file to be of type MPEG, or by allowing the filename field in the Texture2 node to be an MFString (to specify multiple frames for a very short animation).
4. Support for internationalization. I'll spare everyone the details.
5. Drop the requirement that lights only illuminate some objects in the scene, but not others (i.e., the requirement that light stop at Separators). This has been re-proposed. Note that Separators are currently being used for two unrelated things: movement hierarchies and restriction of light sources. This produces problems later, when we want to have things like car headlights or flashlights that people can use.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



img src="buy.gif" height=52 width=75>

Appendix E - C Source Code

This chapter contains the source code listings for the nff2vrml.c and forest.c programs from back in [Chapter 32](#).

NFF2VRML.C

```
/* Convert a Sense8 NFF file into VRML */
```

```
/* Written by Bernie Roehl, October 1995 */
```

```
/*
```

NFF Features not supported:

Textures (VRML doesn't yet support PER_FACE texture maps)

24-bit color specifiers (12 bit is assumed)

Also, normal handling is not working right (NFF and VRML have very different ideas about how it should work)

```
*/
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <ctype.h>
```

```
typedef struct { float x, y, z; } Vertex;
```

```
typedef struct
```

```

{
int npts; /* number of points in this polygon */
int *points; /* indices into the vertex array */
int color; /* index into the array of color values */
} Poly;

typedef unsigned long Color;

Color *colors = NULL; /* one entry for each *unique* color */

int ncolors = 0; /* number of unique colors defined */

Vertex *vertices = NULL; /* the array of vertices */

int nvertices = 0; /* number of vertices defined */

Vertex *normals = NULL; /* the array of normals */

int nnormals = 0; /* number of normals defined */

Poly *polys = NULL; /* polygons */

int npolys = 0; /* number of polygons defined */

float scale = 1.0; /* scale factor */

char objname[256]; /* the name of the object */

int shading = 0; /* shading flag (not currently used) */

int solid = 1; /* set if backface removal okay */

Vertex camloc = { 0, 0, 0 }; /* camera location (currently not used) */

Vertex camdir = { 0, 0, 0 }; /* camera direction (currently not used) */

char *progname = "NFF2VRML"; /* name of this program */

/* read a line, stripping comments and skipping blank lines */

/* returns 0 at end of file */

int getline(char *buff)
{
do
{
char *p;

```

```

if (gets(buff) == NULL) /* end of file */
return 0;

p = strstr(buff, "//"); /* look for start of comment */

if (p)
*p = '\0'; /* truncate the line at that point */
} while (buff[0] == '\0'); /* skip blank lines */

return 1; /* not end of file */

}

void *ecalloc(int nitems, int size)
{
void *p = calloc(nitems, size);

if (p == NULL)
{
fprintf(stderr, "%s: calloc() failed ó not enough _memory\n", progname);
exit(1);
}

return p;
}

void make_camera(void)
{
int i;

float minx, maxx, miny, maxy, minz, maxz;

/* compute bounding box */

minx = maxx = vertices[0].x;
miny = maxy = vertices[0].y;
minz = maxz = vertices[0].z;

for (i = 1; i < nvertices; ++i)
{

```



```

if (vertices[i].x < minx)
    minx = vertices[i].x;
if (vertices[i].x > maxx)
    maxx = vertices[i].x;
if (vertices[i].y < miny)
    miny = vertices[i].y;
if (vertices[i].y > maxy)
    maxy = vertices[i].y;
if (vertices[i].z < minz)
    minz = vertices[i].z;
if (vertices[i].z > maxz)
    maxz = vertices[i].z;
}
printf("\tPerspectiveCamera { position %f %f %f }\n",
(minx + maxx) / 2, -(miny + maxy) / 2, maxz + (maxz - minz));
}
void main(int argc, char *argv[])
{
    char buff[256];
    int i, j;
    if (argc > 1)
        sscanf(argv[1], "%f", &scale);
    getline(buff);
    if (strnicmp(buff, "nff", 3))
    {
        fprintf(stderr, "No header line ó not a valid NFF file\n");
        exit(1);
    }
}

```

```

while (getline(buff))
{
if (isdigit(buff[0]))
break;

if (!strnicmp(buff, "viewpos", 7))
sscanf(buff, "viewpos %f %f %f", &camloc.x, &camloc.y, &camloc.z);

else if (!strnicmp(buff, "viewdir", 7))
sscanf(buff, "viewdir %f %f %f", &camdir.x, &camdir.y, &camdir.z);

else if (!strnicmp(buff, "version", 7))
{
float version;

sscanf(buff, "version %f", &version);

if (version != 2.0)
fprintf(stderr, "Warning: NFF version is %f, not
2.0\n", version);

}

else
{
char tempstr[100];

/* pick up object name */

if (sscanf(buff, "%s %s", objname, tempstr) == 2)
shading = toupper(tempstr[9]) == 'n';

}

}

/* first line is number of vertices */

nvertices = atoi(buff);

vertices = calloc(nvertices, sizeof(Vertex));

normals = calloc(nvertices, sizeof(Vertex));

```

```

/* read the vertices themselves */
for (i = 0; i < nvertices; ++i)
{
    float x, y, z;

    int m;

    getline(buff);

    m = sscanf(buff, "%f %f %f norm %f %f %f",
        &vertices[i].x, &vertices[i].y, &vertices[i].z,
        &normals[nnormals].x, &normals[nnormals].y, &normals[nnormals].z);

    if (m >= 6)
        ++nnormals;
}

/* next line is the number of polygons */
getline(buff);

npolys = atoi(buff);

/* allocate the polygon array */
polys = calloc(npolys, sizeof(Poly));

/* allocate the color array; at worst, we'll need one for each polygon */
colors = calloc(npolys, sizeof(Color));

/* read the polygons */
for (i = 0; i < npolys; ++i)
{
    Color col;

    char *p;

    /* read the next line of input */
    getline(buff);

    /* if any of the polys are double-sided, the object can't be solid
    if (strstr(buff, "both"))

```

```

solid = 0;

/* read the number of points */
polys[i].npts = strtoul(buff, &p, 0);

/* allocate the array of points for this polygon */
polys[i].points = calloc(polys[i].npts, sizeof(int));

/* read the points themselves */
for (j = 0; j < polys[i].npts; ++j)
    polys[i].points[j] = strtoul(p, &p, 0);

/* read the color (assume 12-bit) */
col = strtoul(p, &p, 0);

/* see if we've already got this color in the array */
for (j = 0; j < ncolors; ++j)
    if (colors[j] == col)
        break;

/* if it's a new color, add it to the array of colors */
if (j >= ncolors)
    colors[ncolors++] = col;

polys[i].color = j;
}

/* put out header */
printf("#VRML V1.0 ascii\n# Converted by NFF2VRML\n\n");
printf("Separator {\n");

make_camera();

/* put the object name out inside an Info node */
if (objname[0])
    printf("\tDEF Title Info { string \"%s\" }\n", objname);

/* faces are counterclockwise; if none of the faces were double-sided,
then it's a SOLID shape */

```

```

printf("\tShapeHints { vertexOrdering COUNTERCLOCKWISE %s }\n",
solid ? "shapeType SOLID" : "shapeType UNKNOWN_SHAPE_TYPE");

printf("\tMaterialBinding { value PER_FACE_INDEXED }\n");

/* emit the coordinates */

printf("\tCoordinate3 {\n\t\tpoint [\n");

for (i = 0; i < nvertices; ++i)

printf("\t\t\t%f %f %f,\n",
vertices[i].x * scale,
-vertices[i].y * scale, /* coordinate system is _flipped */
vertices[i].z * scale);

printf("\t\t]\n\t}\n");

#ifdef NORMALS_WORKING

/* if any normals were specified, emit them */

if (nnormals == nvertices)
{
printf("\tNormal {\n\t\tvector [\n");

for (i = 0; i < nnormals; ++i)

printf("\t\t\t%f %f %f,\n", normals[i].x, _normals[i].y, normals[i].z);

printf("\t\t]\n");
}

#endif

/* put out the unique colors as materials */

printf("\tMaterial {\n\t\t\tdiffuseColor [\n");

for (i = 0; i < ncolors; ++i)

printf("\t\t\t%f %f %f,\n", ((colors[i] >> 8) & 0x0F)/15.0, _((colors[i] >> 4) & 0x0F)/15.0, (colors[i] &
0x0F)/15.0);

printf("\t\t]\n\t}\n");

printf("\tIndexedFaceSet {\n");

/* put out the actual polygons */

```

```
printf("\t\tcoordIndex [\n");  
for (i = 0; i < npolys; ++i)  
{  
printf("\t\t\t");  
for (j = 0; j < polys[i].npts; ++j)  
printf("%d,", polys[i].points[j]);  
printf("-1,\n");  
}  
printf("\t\t]\n");  
/* and their material indices */  
printf("\t\tmaterialIndex [\n");  
for (i = 0; i < npolys; ++i)  
{  
if (i % 10 == 0)  
printf("\t\t\t");  
printf("%d, ", polys[i].color);  
if ((i+1) % 10 == 0) /* see if we've filled a line */  
printf("\n");  
}  
if (i % 10 != 0)  
printf("\n");  
printf("\t\t]\n");  
printf("\t}\n");  
printf("}\n");  
}
```

FOREST.C

/* Generate a random VRML forest */

/* Written by Bernie Roehl, October 1995 */

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

float placement_threshold = 0.75; /* if rand is greater than this, _place a tree */

float autumn_threshold = 0.75; /* if rand is greater than this, _use fall colors */

int nrows = 32, ncols = 32; /* number of rows and columns in the grid */

float horizontal_spacing = 5, vertical_spacing = 5; /* sizes of the cells */

/* process command-line arguments */

void procarg(char *arg)

{

switch (*arg++)

{

case 'r': nrows = atoi(arg); break;

case 'c': ncols = atoi(arg); break;

case 'a': autumn_threshold = atof(arg); break;

case 'p': placement_threshold = atof(arg); break;

case 'h': horizontal_spacing = atof(arg); break;

case 'v': vertical_spacing = atof(arg); break;

case '?':

fprintf(stderr, "Usage: forest [options]\noptions _include:\n\t-rrows -ccols -aautumn_threshold -
_placement_threshold\n\t-hhorizontal_spacing -vvertical_spacing - _?\n");

exit(0);

break;

default: fprintf(stderr, "Warning: unrecognized option\n"); break;

}

```

}

/* create a tree */

void make_tree(int row, int col)
{
float size_factor = (rand() / (float) RAND_MAX) * (2.0 - 0.25) + 0.25;

printf("\tSeparator {\n");

printf("\t\tTransform {\n");

printf("\t\t\tscaleFactor %f %f %f\n", size_factor, size_factor, size_factor);

printf("\t\t\trotation 0 1 0 %f\n", (rand() * 2 * 3.1415926) / RAND_MAX);

printf("\t\t\ttranslation %f 0 %f\n",
horizontal_spacing * (col + rand() / (float) RAND_MAX - 0.5),
vertical_spacing * (row + rand() / (float) RAND_MAX - 0.5));

printf("\t\t}\n");

/* determine which set of colors to use */

if ((rand() / (float) RAND_MAX) > autumn_threshold)
printf("\t\tUSE spring_material\n");

else
printf("\t\tUSE autumn_material\n");

printf("\t\tUSE tree\n");

printf("\t}\n");

}

void main(int argc, char *argv[])
{

int i, j;

/* start by processing the command-line arguments */

while (argc > 1)
{

if (*argv[1] == '-')

```



```

{
procarg(&argv[1][1]);
óargc;
++argv;
}
}

/* put out header */

printf("#VRML V1.0 ascii\n\n# Generated by FOREST\n\nSeparator _{\n");
printf("\tPerspectiveCamera { position %f %f %f }\n",
ncols * horizontal_spacing / 2, 12.0, nrows * _vertical_spacing * 1.5);

/* define (but don't instance!) a tree */

printf("\tSwitch {\n\t\twhichChild -1\n\t\tDEF tree WWWInline { _name \"tree1.wrl\" }\n\t}\n");

/* define the two sets of colors */

printf("\tDEF spring_material Material { diffuseColor [ 0.0 _0.666667 0.0, 0.466667 0.466667 0.0 ]
}\n");

printf("\tDEF autumn_material Material { diffuseColor [ 0.25 _0.25 0.0, 0.466667 0.466667 0.0 ] }\n");

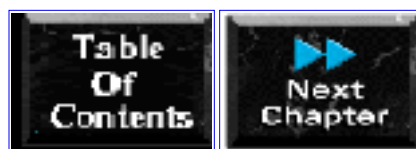
/* go through the array, putting out trees when appropriate */

for (i = 0; i < nrows; ++i)
for (j = 0; j < ncols; ++j)
if ((rand() / (float) RAND_MAX) > placement_threshold)
make_tree(i, j);

printf("}\n");
}

```

The source code for both of these programs, and several others as well, can be found on the companion CD-ROM.



Notice: This material is excerpted from Special Edition Using VRML, ISBN: 0-7897-0494-3. The electronic version of this material has not been through the final proof reading stage that the book goes through before being published in printed form. Some errors may exist here that are corrected before the book is published. This material is provided "as is" without any warranty of any kind.



Appendix F - The VRML CD

To enjoy the world of VRML is to start exploring those worlds and visions created by those who are shaping its future. You'll need a browser or a plug-in for your browser to get yourself started, and creating your own worlds will be quicker, easier, and more fun with VRML toolkits and utilities.

As the excitement of VRML escalates, you won't want to wait to find the right softwareóso we've assembled some of the best and latest software available for whatever platform you use. No searching, no hunting, no long downloads. The software on the VRML CD is ready for downloading to your computer and ready to install.

Browsers

To navigate through a VRML world you'll need a browser and we have quite a selection here. The majority of these programs are beta versionsóworks in progress, in other words. They're all functional and the latest versions that could be obtained. By the time you purchase this book there will be even newer versions, even new VRML applications themselves. Many of the browsers will work stand-alone and as helper applications; two of them (VRScout and WebFX) are plug-ins specifically for Netscape, while the VRML Add-in for Internet Explorer works only with Microsoft Internet Explorer.

Toolkits and Utilities

If you'd like to create a world or populate the one you've got, you'll need the various tools and utilities found here. You'll find format conversion programs, modeling programs, and worldbuilding toolkits.

Objects and Worlds

You'd be hard-pressed not to wander through some existing worlds before you create your own. You'll find a number of these objects and worlds discussed in detail in the book; use them to better understand key VRML concepts and as sample environments to get you started in your own world-building.

Websites

Explore, discover, and participate in the VRML community. We've given you many links to software developers, VRML worlds, mailing lists, and cool VRML sites on the Web.

Docs

The original documents outlining the VRML specifications are here, as well as the proposed specifications for VRML 1.1 and suggested changes as VRML 2.0 evolves.

Source

If you're a programmer interested in writing your own VRML browser or other applications, we've included the source code to VRWebóone of the best known VRML browsers (also available for more platforms than any other browser).

Using the CD

We've constructed the CD so it functions as a Web site. Using your favorite browser, you can load the file startme.htm and be up and running. Click on the hotlinks to find the tools and programs; in most cases, the end of each link path will bring you to the subdirectory or folder that contains the compressed version of the softwareóready to be downloaded and installed on your machine. When possible, you'll also find the contents of that file already decompressed, if you choose to install directly from the CD-ROM. Programs that need to be decompressed will most often need WinZip, which we've included and set a link for on the first page of the CD. Of course, on every Web page you'll find links to the software and companies who've developed the software, making it simple for you to track down the latest version.

Shareware

Much of the software included on this disk is shareware, a distribution scheme whereby the user gets to try the software out before paying for it. Many of these programs, once registered and paid for (usually a very small sum) will give you additional features, longer working periods, additional documentation, and possibly future upgrades (or any combination of these things). If you appreciate the efforts of the shareware authors, please support them and register!

