An Investigation of Visibility Techniques to Improve Rendering Speed in 3D Browsers

Diplomarbeit in Telematik

Georg N. Mészáros

Technische Universität Graz
Institut für Informationsverarbeitung
und Computergestützte neue Medien (IICM)

Fertigstellung: 26. 8. 1996
Prüfungsfach: Informationssysteme
Betreuer: Dipl.-Ing. Keith Andrews
Begutachter: O. Univ.-Prof. Dr. Dr. h.c. Hermann Maurer

Ich versichere, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Die Diplomarbeit ist in englischer Sprache verfaßt.

Acknowledgements

This work was inspired by an excellent lecture on Interactive Computer Graphics by Mel Slater during my studies at Queen Mary & Westfield College, University of London.
I would like to thank Michael Pichler for his helpful technical advice regarding VRweb and the GE3D library, and Keith Andrews and Prof. Hermann Maurer for their support during my work at the IICM, University of Technology, Graz.

Abstract

As VRML becomes *the* standard for describing 3D scenes on the Internet, many VRML viewers are being developed with proprietary interests or for specific target systems or protocols. VRweb is a VRML viewer available as both binary and source code for multiple platforms. VRweb source code is copyrighted, but is freely available for non-commercial use and therefore providing a good platform for research and experiment.

Almost all currently available VRML 3D browsers are built on graphics libraries using depth values in a Z-buffer to achieve correct visibility in the scene. In this thesis I investigate the possibility of improving rendering speed in VRweb through visibility calculation to determine and render only the visible parts.

Especially when navigating through virtual environments like buildings or houses it seems convincing that, *being* in a room, the part of the scene that lies outside the room, does not need to be rendered.

In the course of this thesis, VRML and some 3D browsers are introduced before giving a short introduction to 3D graphics, viewing and rendering. The explanation of different visibility approaches is followed by an overview of some 3D graphics libraries. After a closer look at VRweb, the implemented visibility algorithms are discussed. Finally the chosen approach is evaluated and compared with the standard rendering method using a Z-buffer before giving a framework of how to accelerate interactive walkthroughs in buildings using hierarchical viewing volume culling in VRweb.

Contents

Introduction

There is so much information online today, it's hard to imagine the amount and size of things that will exist in digital storage tomorrow. We must be able to create new tools to visualize, search, and manipulate information. Virtual environments provide a way to combine the best features of real-world information navigation – memory of places and visual clues – with the best features of online navigation – fast searches, sorting and quick cross-referencing.

Experience is by far the best teacher. Because virtual environments can mirror the effects of reality, it is possible to make systems and objects that we can learn how to use as effectively as we learn things in real life. By creating realities, it is possible to do away with many aspects of interfaces as we know them today.

Therefore VRML, a way of describing 3D objects and scenes, is one of the most exciting developments on the Internet today and offers tantalizing previews of what is to come in cyberspace. The VRML format merely specifies what the client machine should render. The speed of the machine's graphics capabilities has very much influence on the achieved frame rates and therefore on the acceptance of 3D as a medium for browsing, retrieving information, or examinating objects.

The current implementation of VRweb takes advantage of graphics hardware acceleration to provide real time rendering. Up to now no provision has been made to speed up the rendering process either by removing hidden surfaces or by implementing special algorithms for workstations without graphics hardware support.

In the course of this thesis, I investigate how to speed up rendering (especially on slower machines) at least for the static parts of an environment, where only the viewing position changes, through new algorithms supporting the calculation of non-visible parts of a 3D model.

I would like not only to improve the rendering speed in VRweb, but also to prompt developers and researchers to not only be concerned with building new interfaces on top of old rendering engines, but also with new visibility techniques. Since VRweb provides a good platform for research and experiment, future research work might be built on the current implementation in VRweb.

Chapter 1

Virtual Reality and the Internet

1.1 Introduction

1993 was the year of the great Internet explosion, when charts following net usage on the North American backbones went crazy. The numbers are still increasing. The internet is a pretty nice thing - but what do I do will all that information? How can I use it in an appropriate way? I do not want to get lost in Cyberspace!

Kevin Hughes [Hug95] cites that *cyberspace is a an interconnected computer-mediated environment in which all prior media are represented* - that is correct. I would only like to extend this definition that there are not only the same media represented but that we have the possibility to find new ways of offering and retrieving information.

New opportunities arise through computer supported information technology. The most fascinating ones seem to be covered with expressions like *hypermedia*, *interaction*, *virtual reality*, and *3D graphics*.

In this chapter I would like to give a short introduction of the history of the Internet and standards like *HTML*, introduce *VRML*, *Java*, the currently most important 3D browsers and modelers, and mention some collaborative 3D environments.

1.2 Internetworking and the Internet

The Internet, a worldwide network of networks, is becoming so pervasive that it is often referred to as *The Net*. The recent popularity and visibility of the World Wide Web (WWW or W3) on the Internet has introduced the term *The Web*, embracing the sum of Internet services and becoming more or less synonymous

with "The Net" and the Internet.

The World Wide Web is defined by three key specifications: HTML, HTTP and URLs. HTML (the HyperText Markup Language) defines how text is marked up with tags to define certain constructs (such as enumerated lists or a hypertext link). The final presentation of a text document depends on how individual WWW clients render particular HTML constructs.

HTTP (the HyperText Transfer Protocol), the WWW's client-server protocol is used by clients to send a request message and servers reply with a response message, in a single stateless transaction.

Through its URL (Uniform Resource Locator) mechanism, WWW encapsulates other Internet protocols by enabling links to any document on any WWW, Gopher or FTP server worldwide, as well as newsgroups or Telnet sessions. "The Web" has embraced and become almost synonymous with "The Net".

In October 1994, the World Wide Web Consortium (W3C) was formed to develop WWW standards and reference code. The Virtual Reality Modeling Language (VRML), a 3D file format for modeling objects and worlds on the Web, was specified in version 1.0 in June 1995 and has been rapidly extended until the announcement of Moving Worlds VRML 2.0 in June 1996. The introduction of Java, a secure language which executes on a "virtual machine", by Sun Microsystems promises to bring interactivity to the Web in a form of downloadable mini-applications, or "applets".

The future of the Web will offer a lot more possibilities than "old" media like television or the telephone ever could. Feedback from a reader to the author of a document or communication between more than two participants from all over the world will be essential for the next milestone in information technology.

But why do we need "Virtual Environments" for things we can already do with email, newsgroups or applications like Internet Relay Chat (IRC) – where everybody can talk to whoever willing to have a chat? Why are 3D-environments important to use in terms of attracting users, for navigation on the Web or for meetings and conferences?

People need to be with and communicate with others. At least it is the most natural thing for people to do. Collaborative virtual environments provide gathering grounds for new communities and types of interaction, and they give people a voice like they have never had before.

We can share experiences and visions and learn to understand the other person's point of view. We can attend concerts, act in plays, and attend classes with an international audience. As long as people have something to say to somebody else, they can say it online.

Due to the importance of virtual environments and interactivity for the fu-

ture of the Web I would like to give some more details on VRML, Java, and collaborative 3D environments in the following sections.

1.3 VRML - Virtual Reality Modeling Language

The Virtual Reality Modelling Language [Pes95] is a language for describing interactive simulations – virtual worlds networked via the global Internet and hyperlinked with the World Wide Web [BLCL+94]. All aspects of virtual world display and interaction can be specified using VRML. It is the intention of its designers that VRML becomes the standard language for interactive simulation within the World Wide Web.

VRML is to 3D what HTML is to 2D text. While HTML specifies how textual documents are represented, VRML is a format that describes how three-dimensional environments can be explored and created on the World Wide Web. Since 2D is really just a subset of 3D, any two-dimensional object can be easily represented in a three-dimensional environment. In Mark Pesce's book "VRML: Browsing and Building in Cyberspace" Tim Berners-Lee, the "father" of the Web, reasons that VRML is the future of the Web because it is more natural for us to be immersed in a three-dimensional space than to click our way through hyperlinked pages.

Unlike programming languages such as C++, VRML does not have to be compiled and run. Rather, VRML files get parsed and then displayed by a viewer. It also allows for more interactivity and facilitates incremental improvements.

As you navigate through the scene, you will notice that some objects are linked. If you click on them, you will jump either to another VRML world or to another media type such as HTML. While this gets you to other places quickly, it leads to the same problem as "jumping" from link to link in HTML: you jump from a document on one topic on a server in one place to another document on a different topic in a completely different place.

Traditional commercial Web sites can draw in new users by adding three-dimensional environments that are fun to explore and by providing a natural way to navigate through the information available on the site. Ultimately, you will be able to collaborate with other users with live audio links in context-rich 3D environments rather than typing away at the command prompt in a text-based chat room.

The performance of rendering external VRML scenes, fetched over the Net, highly depends on the power of your CPU and display and only to a lesser extent

on the bandwidth of your Internet connection. Luckily, VRML includes a feature that has been used successfully by game developers for years: levels of detail. Through levels of detail, your browser can take some shortcuts and display an approximation of an object before calculating it in detail. Since you have a first-person perspective in VRML, the browsers "knows" what you are looking at. For example, if you are still quite far away from an object, it can just display the approximation, which is the same object with a lower level of detail. As you move closer to this object, the VRML browser will dynamically display higher resolution versions of the same object. By the time you get close enough to the object to pick it up, you will see the object in all its glorious detail. If the creator of the world chose the right switching points, you won't even notice the transitions since far-away objects are so small that you can't perceive the more detailed features anyway.

In the long run, the answer is simple. Every site that is set up to engage visitors and keep them coming back will eventually be three-dimensional. Except for cases where the content is highly symbolic and where the potential visitors are well-trained to access information through command-line interfaces, there is a role for 3D in virtually every Web site. Even if the content of your site is primarily symbolic, you can use 3D to help less symbolically-inclined visitors to visualize your data.

While text editors such as vi were sufficient to convey the most complex ideas via HTML, you will need to invest in good 3D creation and VRML authoring tools to design worlds that go beyond simple spheres and cubes. Although it is theoretically possible to type a VRML file that describes the geometry and all the surface attributes of a complex construction, this really doesn't work in practice. Text is adequate for symbolic reasoning, but it is a poor visualizer. We can easily conjure up the image of a complex object, but to convey all of its details is simply impossible.

Future versions of VRML will allow rich behaviours, including animation (changing scenes) and real-time multi-user interaction.

1.3.1 VRML related Software Development Libraries

The following software libraries are directly related to VRML. They have been written specifically for the Virtual Reality Modeling Language, therefore software developers who are writing VRML applications should find them useful.

- QvLib, a VRML 1.0 parser library. The VRML Library, QvLib, is a set of C++ routines which can parse VRML files. Its output is a "parse tree",

which may be traversed by a program to generate a "view" (or a translation) of a VRML environment. Paul Strauss and Gavin Bell, Silicon Graphics. `http://vrml.wired.com/vrml.tech/qv.html`

- QvLib Reverse Parser, a plug-in replacement for the qvtraverse.cpp source module will output VRML from a parse tree built by QvLib. Tenet Networks. `http://www.tenet.net/html/qvregen.html`

- QvLib with OpenGL rendering support, a version of QvLib with OpenGL rendering support for simple nodes: cube, sphere, cylinder, perspectiveCamera, etc. Torgeir Viemo, Institutt for Informatikk. `http://www.ii.uib.no/t̃orgeir/`

- QvTraverse C++ source code that adds OpenGL calls to QvLib to form a sample traverser for VRML. Jan Hardenbergh. `http://vrml.wired.com/arch/1107.html`

- A Yacc/lex VRML parser. C, Yacc and lex source for parsing VRML files. The sample program simply checks for syntactical correctness. Semantic actions may be added to build a parse tree. `http://www.sics.se/dive/docs/dive_vrml.html`

1.3.2 The Moving Worlds VRML 2.0 Specification

The key concepts of VRML 2.0 include how nodes are combined into scene graphs, how nodes receive and generate events, how to create node types using prototypes, how to add node types to VRML and export them for use by others, how to incorporate programmatic scripts into a VRML file, and various topics on nodes.

I would like to mention some new features compared to older versions of VRML:

- Behaviour – direct modification of a node's attribute like geometry, position or appearance, or the change of attributes over time.

- Nodes can receive a number of *incoming events*, which can change the node. Nodes can also send out a number of *changed events*, which indicate that something in the node has changed (for example, *position changed or colour changed*).

- The connection between the node generating the event and the node receiving the event is called a *route*. A route is a syntactic construct for establishing event paths between nodes.

- The world creator may place any number of *viewpoints* in the world – interesting places from which the user might wish to view the world.
- TimeSensors can generate events as time passes. Typically the times generated by TimeSensors will roughly correspond to "real" time. Time (0.0) starts at 12 midnight GMT January 1, 1970.
- Sensor nodes generate events. Geometric sensor nodes generate events based on user action, such as mouse click or navigating close to a particular object.
- A Script node is activated when it receives an event. At that point the browser executes the program in the Script node's *url* field (passing the program to an external interpreter if necessary). The program can perform a wide variety of actions: sending out events (and thereby changing the scene), performing calculations, communicating with servers elsewhere on the Internet, and so on. Scripts can be written in any language supported by the browser. VRMLScript and Java are the most common scripting language implementations.

1.4 Java

The Java programming language and environment [Sun] is designed to solve a number of problems in modern programming practice. It started as a part of a larger project to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. One way to characterize a system is with a set of buzzwords, as the developers of Java do.

> "Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language."

Treating these characteristics in turn:

Simple

Most programmers working these days use C and, for object oriented programming, C++. Java was as closely designed to C++ in order to ease the adaptation to Java, but Java omits rarely used, confusing features of C++. These omitted

features consist of operator overloading, multiple inheritance, and extensive type coercions.

Another aspect of being simple is being small. The size of the basic interpreter and class support is about 40k bytes; adding the standard libraries and thread support adds an additional 175k bytes.

Object-Oriented

Object-oriented design is very powerful because of the clean definition of interfaces which makes it possible to write reusable software. The design defines how different modules work together. The object-oriented facilities of Java are essentially those of C++, with extensions from Onjective C for more dynamic method resolution – interfaces – a concept which is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementations.

Distributed

Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease that programmers are used to when accessing a local file system.

Robust

C++ inherits a number of loopholes in compile-time checking from C, which is relatively lax concerning method and procedure declaration. In Java, declarations are required, implicit declarations are not supported.

The biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Java has true arrays which know their size. In addition, it is not possible to cast an arbitrary integer to a pointer.

Secure

Java is intended to be used in networked/distributed environments. Java enables the construction of virus-free systems through applications running on virtual machines.

There is a strong interplay between "robust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Architecture Neutral

In general, networks are composed of a variety of systems with a variety of CPU and operating system. To enable a Java application to execute anywhere on the network, the compiler generates an architecture neutral object file format – the compiled "bytecode" is executable on all processors, the Java runtime system (the virtual machine) was ported to.

The generated bytecode instructions have nothing to do with a particular computer architecture. The design goal was to make the bytecode both easy to interpret on any machine and easy to translate into native machine code on the fly.

Portable

Unlike C and C++, there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point number.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows, and the Macintosh.

The Java system itself is quite portable. Sun's compiler is written in Java and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Multithreaded

Writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style.

Java has a sophisticated set of synchronization primitives standard across all platforms that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare. Much of the style of this integration comes from Xerox's Cedar/Mesa system.

Dynamic

Problems with programming languages where the code is always implemented, like C or C++, arise when a library is being updated. In such a case the whole application would need to be recompiled.

By making the interconnections between modules late, Java completely avoids these problems and makes the use of the object-oriented paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

Classes have a runtime representation: there is a class named Class, instances of which contain runtime class definitions. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. However, in Java, finding out based on the runtime type information is straightforward.

It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and have it easily dynamically-linked into the running system.

1.4.1 Summary

The Java language provides a powerful addition to the tools that programmers have at their disposal. Java makes programming easier because it is object-oriented and has automatic garbage collection. In addition, because compiled Java byte-code is architecture-neutral, Java applications are ideal for a diverse environment like the Internet.

It seems possible that all Internet applications might be implemented in Java in the coming years. Concerning 3D, a library called Java3D is under development.

1.4.2 Java3D

Java3D is one of several Media APIs being jointly developed by Sun and multiple other vendors. The preliminary specification for Java3D is expected out sometime in the second half of 1996, with implementations to follow several quarters later.

Java3D is intended to serve as a high-level, platform-independent 3D rendering API, primarily at the scene graph level, but Java3D also includes immediate mode support.

Support for VRML 2.0 and other industry standard 3D file formats was an important design goal for Java3D. But Java3D is only a run-time API; thus a browser manufacturer will need to include some file format specific support; for

example a VRML 2.0 loader that will read VRML 2.0 files and convert them into appropriate Java3D constructs.

As part of the agreement that formed the various JavaMedia consortia, the ground rules restrict partner companies from commenting on details of the various APIs before concurrence. This is the reason for the lack of detailed public information on Java3D.

1.4.3 Liquid Reality

Liquid Reality [X] is a VRML toolkit from Dimension X, licenced by Microsoft in July 1996. The toolkit is a set of Java class libraries that gives you VRML functionality. With the toolkit, you can create viewers, tools, and solutions that are VRML 2.0 compliant. In addition, the tool kit is extensible using Java. This extensibility allows you to create custom nodes that meet your needs. Liquid Reality can be used in conjunction with ICE (Dimension X's graphics library) or can use Direct3D (see Section 4.6) as the graphics library on the Windows platform. The toolkit has the following features:

- Platform independence
- Support for platform level 3D interfaces – Direct3D (fast rendering)
- Can build viewers and applications that work inside of Netscape and as Java applets
- Create a VRML 2.0 compliant viewer

A beta version of the Liquid Reality toolkit is available for a free download. This toolkit is for the Windows 95/NT4.0, Solaris, Linux, and Irix platforms.

1.4.4 Habanero

The Habanero project [NCS] at NCSA is investigating the enhancements in distributed interpersonal communication made possible when single-user computer software tools are recast as multi-user, collaborative work environments. A fundamental restructuring of the nature of distributed work is necessary, and desirable, in the interest of improving efficiency and productivity of the teams involved.

Habanero is a framework for sharing Java objects with colleagues distributed around the Internet. Included, or planned, are all the networking facilities, routing, arbitration and synchronization mechanisms necessary to accomplish the sharing of state data and key events between collaborator's copies of a software tool. Authentication and privacy features are also planned. There is no inherent limit in the number of tools per session, nor is there a limit on the type of tools

that may be shared. As the project progresses, additional capabilities will enable routing of Habanero session information to a very wide number of participants. A limited, but representative set of applications is provided. No security model is presently in place.

Due to the nature of Habanero's birthplace (NCSA at the University of Illinois), there may be detectable biases toward the sciences, particularly computational science and computer science, and the education community. However, the Habanero framework is sufficiently generalized to be immediately usable in a very large number of communities and problem domains.

There will be at least one multi-platform Web Browser that's Habanero-sharable. It's release is anticipated for the third quarter of 1996. The availability of a browser will provide access to the legacy of information that's reachable via the Web, and via gateways operating on Web Servers.

1.5 3D Browsers

Since the release of the first VRML viewers/browsers in April 1995, there have been a flood of releases and announcements. Here I present an overview of the field as of August 1996 (which is necessarily neither complete nor up-to-date). The following list is based on information gathered by the VRML Repository [EMNM] and the home pages of the corresponding companies.

1.5.1 WebSpace

WebSpace [Silb] from Silicon Graphics in collaboration with Template Graphics was the first VRML browser to be released. WebSpace is based on the Inventor library and has two navigational metaphors (the examiner viewer and walk viewer). WebSpace binaries are freely available (unsupported) with a supported commercial version; no source code is available. WebSpace runs on SGI, IBM AIX, Sun Solaris (with graphics board), Windows 95 and Windows NT; versions for Macintosh, DEC, HP, and Windows 3.1 are planned.

1.5.2 VRweb

VRweb from the IICM, NCSA, and the Gopher team is implemented using OpenGL on SGI, Dec Alpha, and Microsoft Windows NT. For all platforms an implementation using the Mesa library (an OpenGL workalike) is available. It neither needs nor benefits from special graphics hardware, thus allowing browsing VRML scenes on ordinary X-terminals and PCs too. Source code is available for

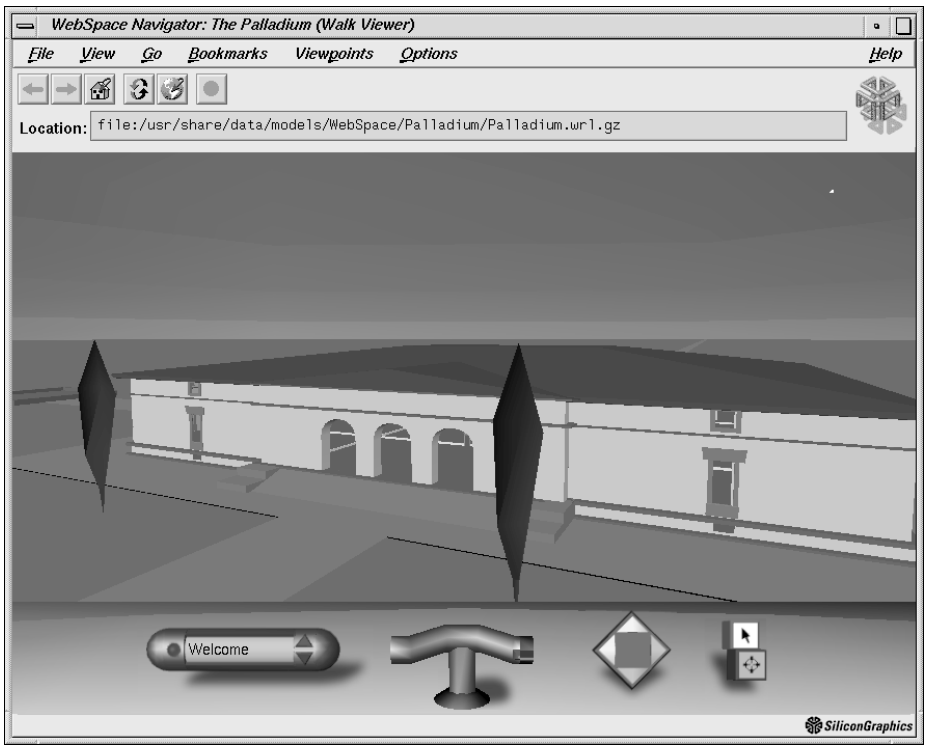

Figure 1.1: WebSpace in walk mode showing the Palladium, Silicon Graphics.

UNIX/X11 and Windows. Details in Chapter 5. Currently Supported Platforms: Windows 95, Windows NT, Windows 3.x (with Win32S), HP-UX, SUN OS, SUN Solaris, SGI IRIX, DEC Alpha, DEC ULTRIX, LINUX, FreeBSD, IBM AIX.

1.5.3 WebView

WebView [Lin] from the San Diego Supercomputer Center (SDSC) is a publically available VRML browser. It runs on any SGI platform with OpenGL and Open-Inventor installed, including IRIX 5.2 and 5.3. WebView is available as source code; it has four viewing styles (examiner, fly, plane, walk) and an integrated editing facility. It can either be run as a standalone browser or as an external viewer with a Web browser. It is intended as a public development and test platform, but is limited to SGI platforms under UNIX.

1.5.4 Live3D

Live3D [Net] from Netscape Inc is a VRML browser for the Windows environment. The binaries are free for non-commercial use and during an evaluation period. It incorporates IRC 3D chatting and physically-based navigation metaphors (including collision detection) as well as VRML authoring facilities. Live3D requires Netscape Navigator 2.0 or later and is currently available for Windows platforms (NT, 3.1, 95) and for PowerMac (beta version). A Unix version is planned.

Live3D is built on top of the proposed Moving Worlds VRML 2.0 specification; it extends the Netscape plug-in API to support Navigator plug-ins in 3D space and defines Java and JavaScript objects.

1.5.5 WorldView

WorldView [Int] from InterVista Software is targeted to empower standard PCs for real-time application. All network communication is built into this standalone browser, which does not rely on a cooperating Web browser. WorldView is available for Windows 95 and Windows NT. Versions for PowerMac and UNIX (SGI, Sun OS) are planned. WorldView was the first stand-alone VRML browser for the Windows platform; it incorporates Microsoft's Reality Lab 3D software rendering technology.

1.5.6 i3D

i3D [Cen]. Using a Spaceball or a mouse, the user can intuitively navigate in-

side 3D worlds, while selecting 3D objects with the mouse triggers requests for access to remote documents of any media type, from text to other 3D models. i3D also supports stereo rendering for CristalEyes LCD shutter glasses and the Multi-Channel Option. i3D was developed initially at CRS4, Cagliari, Italy. Its development is now pursued at CERN by the VENUS group. Currently Supported Platforms: SGI with IRIX 5.2 or later and DEC Alpha with Digital UNIX V3.2C with Open3D V3.2 (with OpenGL support).

1.5.7 WebOOGL

WebOOGL [MMBL] from the Geometry Center of the University of Minnesota is available in source code based on Geomview. Geomview is a program for viewing and manipulating 3D objects and is designed to act as display unit for external modules creating geometry. Multiple control windows exist for motion control, properties and editing. The WebOOGL browser is available for SGI and Sun OS platforms, soon for Linux and other X-Window System Platforms.

WebOOGL is quasi-compliant to VRML 1.0, that means that parts of the VRML specification are silently ignored. The following nodes are currently ignored by WebOOGL: AsciiText, Level Of Detail, SpotLight (treated as a DirectionalLight), Texture, Texture2Transform, TextureCoordinate2.

The WebOOGL project was started in 1994 in response to the call for proposals for VRML file format candidates. The WebOOGL file format is a straightforward adaptation of the OOGL (Object Oriented Graphics Language) file format used by Geomview to include hyperlinks. The proposal took second place in the vote on the VRML mailing list, with first place going to the Open Inventor-based format. The WebOOGL 1.0 software package was put together as a proof of concept and demonstrated at the Second International Conference on the World Wide Web in October 1994 in Chicago.

1.5.8 VRML 2.0 Browsers

These VRML Browsers support the VRML 2.0 specification.

- *Cosmo Player* [Sila] from Silicon Graphics is compliant with the VRML 2.0 specification including scripts, sensors and sound. A specified subset of JavaScript is used as the scripting language. It runs as a plug-in to the Netscape Navigator. Currently Supported Platforms: SGI IRIX, Windows 95, Windows NT. Silicon Graphics, Inc.

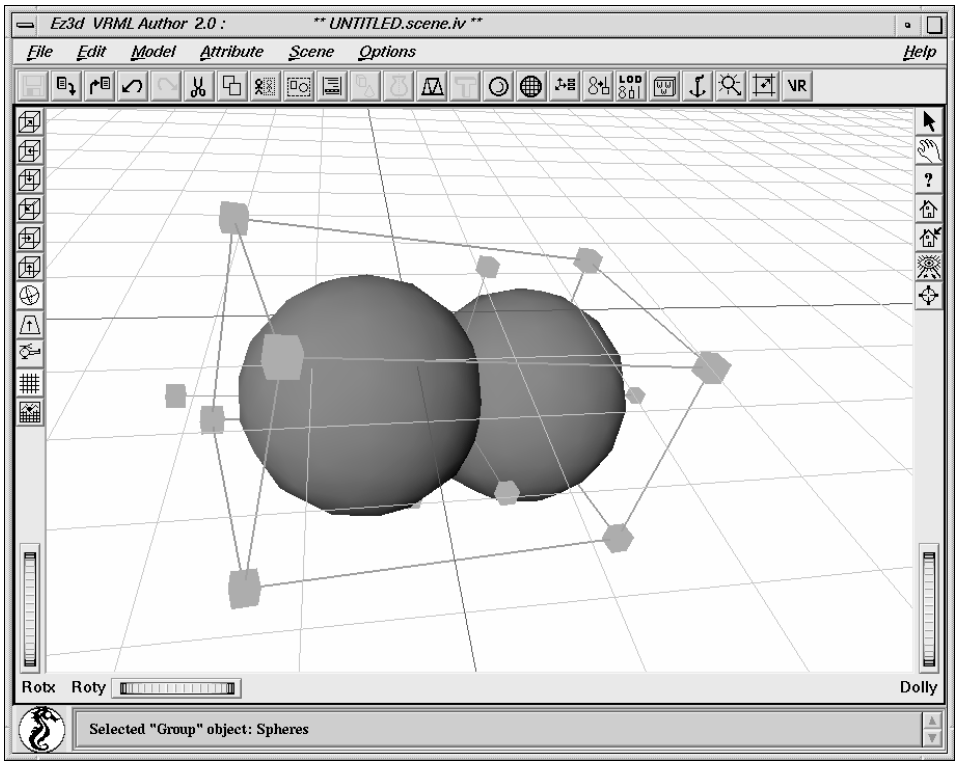

Figure 1.2: Ez3D VRML Author from Radiance Software International.

- *Sony CyberPassage / Community Place* [Son] is a VRML 2.0 browser that also supports multi-user shared worlds, including text chat, and shared behaviours written in Java. It is upwardly compatible with VRML1.0 files. The latest versions also supports the older extensions to VRML (E-VRML) proposed by Sony. Currently Supported Platforms: Windows 95 (with DirectX) and Windows NT 3.51.

1.6 Modelers

A modeler is used to create a three-dimensional scene or object. Most modelers do not require any programming knowledge. The following ones support the VRML 1.0 format. The list is based on information from the VRML Repository [EMNM].

- *Ez3d VRML Author* [Rad] from Radiance Software International is a VRML authoring system with advanced 3D modeling, texture and material mapping, VR scene optimization, WWW inlines/anchors, and raytracing. A "hierarchical object list" with editing facilities might help to create intelligently structured scenes to support visibility algorithms in object space.

- *Medit* from Medit Virtual Reality SL is a low cost modelling tool, designed for generating real time 3D databases for use with software like IRIS Performer. Medit works on any Silicon Graphics Workstation running IRIX version 4.0.5 or above. The ftp copy is a complete working version of Medit, but limited to saving models with less than 1001 polygons.

- *Pioneer* from Caligari Corporation is a complete Microsoft Windows based VRML authoring solution that includes rapid modeling in perspective space, manipulation of texture-mapped VRML objects in real time and interactive lighting.

- *Sony Cyber Passage Conductor* is a tool that allows you to add behaviours, sound and images to VRML 1.0 files. Support for behaviours is via the Sony extensions to VRML. Behaviours are written in TCL and the authoring tool allows you to drag and drop scripts onto existing VRML files to build interactive, animated worlds. You will need the associated Cyber Passage Browser to view these enhanced worlds. Supported platforms are Windows 95 and Windows NT.

- *TriSpectives 1.0 / Professional* from 3D/Eye is a 3D drawing program for Windows 95. With over 1,000 3D clipart models, TriSpectives can be used to add 3D animations to presentations, create 3D illustrations for brochures, flyers and posters, and produce 3D drawings for diagrams and reports with quite a lot import/export features.

- *Virtual Home Space Builder* from ParaGraph International is available for Windows 3.1, Windows 95 and Windows NT. Home Space Builder is mainly concerned with building *homes*, their decorations and multimedia attachments. The developers claim to export VRML 1.0 and VRML 2.0 format.

- *Virtus WalkThrough Pro 2.5* runs on Windows 95 and Power Macintosh. This latest version of Virtus' modeling package includes the ability to export VRML in 2-D and 3-D DXF, Illustrator 1.1, VRML, TIFF, PICT, and BMP. It further supports stereo head-mount displays. A library of 3D-items is included with this package.

- *WebSpace Author 1.0* from Silicon Graphics is a part of the WebFORCE Software Environment. Some features implemented are a Level of Detail (LOD) editor, a Polygonal Reduction Editor and Inline creation to optimize *hand made* scenes. WebSpace runs on any SGI workstation with IRIX 5.3 or 6.1. Supported file formats are Open Inventor and VRML. Alias, Wavefront, Softimage, 3D Studio, IGES, and DXF files can be imported via integrated translators.

1.7 Collaborative 3D Environments

1.7.1 Worlds Chat

Worlds Chat 1.0 Gold [Wor] is a 3D multi-user environment from Worlds Inc. connected to the Internet. You can gather with users from around the world as you explore corridors, chat rooms, and hidden surprises. You can experience online chat within a 3D world. Music and sound effects enhance your journey as you encounter dozens of chat rooms each with a different topic and mood. The Worlds Chat client runs on Windows 3.1 and Windows 95 with a SLIP, PPP or direct connection to the Internet.

1.7.2 Active Worlds / Alpha World

Alpha World [Wor] from Worlds Inc. was the first virtual world created with Active Worlds technology which is Windows 95 and Windows NT compliant and is aligned with Microsoft's Internet product family.

The concept behind Alpha World is to build a virtual reality where people can go and talk with each other. Every user is represented with an "Avatar" (3D representation) so that everyone can see each other walking around. Allowing ownership of land and the ability to build almost anything on your land (with textures and sound support) makes Alpha World coming close to real world behaviour.

Alpha World is not a VRML browser in the same sense as those listed above. This program is a virtual world in its own right. It connects to a specific site (the Alpha World- / Active Worlds server) where information on who's there and what's been built is stored. Only the user can change anything on his or her property. The Alpha World client runs under Windows 3.1 (with Win32S) and Windows 95.

The new technology platform of Active Worlds consists of a set of servers communicating with a Windows '95 browser program called the Active Worlds

Browser. The Active Worlds Browser supports Active X and lets the user visit virtual worlds much like the Microsoft Explorer Web browser the World Wide Web.

1.7.3 Gamma

Gamma [Wor] is the codename for the latest 3D technology from Worlds Inc, it is an integrated client/server technology suite which is enhanced VRML 2.0 compliant that provides viewers, tools, and servers that are required to create, maintain, and distribute multi-user virtual reality environments over the Internet.

The Gamma Shaper is the virtual authoring component, written in Java with special features such as shared state, portals, sound and voice. Gamma Shaper's visual interface supports construction of room-based spaces intuitively while still providing advanced features like behaviours.

Worlds Inc. will release the Gamma Viewer, especially designed to support Worlds' extension to VRML 2.0, such as multi-user and shared state support.

The Gamma Server, written in Java and C++, has unlimited scalability, different servers can be connected to form a larger virtual world. End users can transparently move from world to world, even *walking* across server boundaries on different physical machines. By storing object states on the server, the Gamma Server allows features like transferring objects between users (e.g. files), moving objects from one location to another and changing an object's property. The World Server directory services is developed on a SQL-compliant database to store the user's identity and location since these are critical for a virtual environment. The Gamma Server software runs on SunOS 4.1.3, Solaris 2.4, AIX 4.0, IRIX 5.1, Linux 5.1, SCO, Ultrix, BSD 4.1.3 and Windows NT 3.5+.

1.7.4 CyberHub

CyberHub [Bla] builds on VRML standards to enable 3D multi-user interaction. It includes an integrated user-identification server, a motion tracking database, and an information exchange database that can be run on a single Web server or distributed across multiple hardware platforms. CyberHub runs on any Sun Sparc workstation or under Linux and Windows NT. The server provides Web site traffic monitoring to gain statistics about the number of users or the peak hours of a Web site usage.

The CyberHub browser *CyberLife* is available as a Netscape Navigator plug-in that enhances the built-in Live 3D VRML browsing with multi-user services like interacting with other avatars. The VRML worlds are automatically displayed by

Live3D in the top left frame while CyberLife's user interaction controls appear in frames to the right and bottom of the Live3D window. The client connection requires a Windows 95 or Windows NT workstation.

Chapter 2

3D Graphics and Viewing

2.1 Introduction

Realising the importance and the potential of 3D graphics for the future of information technology and communication in networks, I will introduce the basics of 3D graphics in this chapter. This should help to make it easier to understand the concepts in Chapters 3 and 6 of how to improve rendering speed of virtual worlds. The human brain's demand for at least 10 frames per second (ideal are 25 frames or more) is not yet reachable with an IBM PC or a standard workstation. Nowadays only machines with graphics hardware support are capable of providing sufficient rendering speed. If the frame rate drops below 10 frames per second the feeling of interactivity is lost and users become nauseous and confused.

2.2 Viewing in 3D Graphics

The problem is: given a 3D object, how can it be represented on a 2D plane? The answer arrived by artists and mathematicians is in the notion of a *projection*. A projection is a way of associating each point in 3D space with a unique point in the 2D space in which the representation is to appear. Note that it is always a one-to-many mapping, that is one point in the 2D space will typically represent an infinite number of possible points in the 3D space. In order to be able to do a projection, two things are necessary:

- A 2D surface onto which the projection is to be made. We will call this the *view plane*. It is assumed that the view plane is a plane, that is, it is flat.
- We also need the equivalent of a camera, that is some way of determining how the 3D scene is to be viewed - from which position, looking in which

direction, and so on.

There are two fundamentally different ways of doing a projection. The first is called *parallel* and the second *perspective*. Within these there are many subtypes.

A parallel projection give unrealistic views, in the sense that they don't agree with everyday observation. We know that if we look down a very long road the edges of the road will converge to a point (at infinity). A parallel projection does not result in this – in a parallel projection, parallel lines in a scene remain parallel in the projected image.

In order to get the projection of the road to look correct we need perspective projection. This works rather more like natural vision. In a parallel projection, it is the direction which specifies the view. In a simple perspective projection, it is the viewpoint, or *centre of projection, COP*, and its distance from the view plane which determines the view.

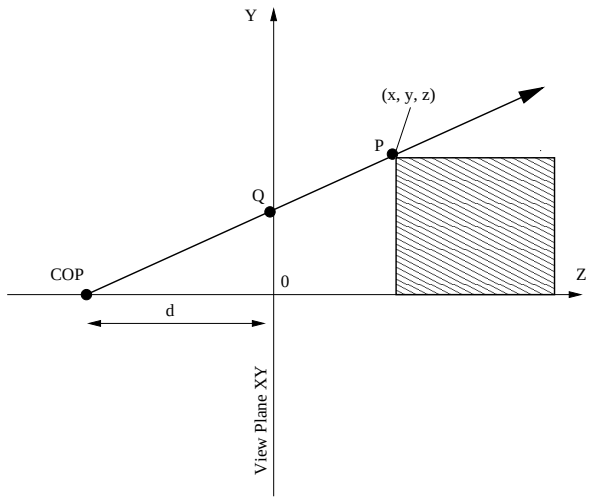

Figure 2.1: Perspective Projection.

2.2.1 The Viewing Coordinate System

The scene in 3D World Coordinates is typically described using a right-handed coordinate system. Here we introduce a set of parameters allowing an arbitrary orientation of the camera [Sla95]. The strategy will then be to apply transformations to this camera, and correspondingly to the scene to get the situation back to the simplicity of Figure 2.1.

There are a number of parameters defining the viewing coordinate system:

- The View Reference Point (VRP). This is the point in World Coordinates (WC) which specifies the origin of the new Viewing Coordinate system. Intuitively, it may be thought of as the point of interest in the scene, or as a point with respect to which the camera (i.e., the view plane and the centre of projection) are to be defined.

- The View Plane Normal (VPN). This is a vector in WC whose direction specifies the positive Z axis of the viewing coordinate system. This axis is therefore obtained as the line through the VRP parallel to the VPN. Intuitively, it may be thought of as the direction in which the abstract camera is pointing. The Z axis is called the N axis in the new system.

- The View Up Vector (VUV). This is a vector in WC whose direction is used to define the positive Y axis of the new coordinate system. The Y axis is formed by projecting the VUV onto a plane which is perpendicular to the VPN and which passes through the VRP. This projection is the Y axis of the new system. The Y axis is referred to as the V axis.

Finally, the X axis of the new system is constructed so as to make a left-handed system given the Y and Z axes. The X axis is often referred to as the U axis.

The names of the three principle axes of the viewing coordinate system give rise to the alternative name – UVN system. These ideas are illustrated in Figure 2.2. It should be clear that the new viewing space is a translation and rotation of the original world space – these two spaces have the same metric.

2.2.2 Specifying the View

Having determined the viewing coordinate system, the remaining parameters can be specified and all remaining parameters are in VC:

- View Plane Distance. The View Plane is a plane orthogonal to the VPN, onto which the scene is projected. The View Plane Distance is the distance of the View Plane, along the VPN, measured from the VRP (i.e., from the origin of the viewing coordinate system, along that system's N axes).

- Centre of Projection. The COP is specified as an offset from the VRP. Rays from every point of the scene converge to this specific point. The intersection of these rays with the View Plane forms the projection. The parallel projection case can be thought of as the situation where the COP is "at minus infinity".

- View Plane Window. This is a window on the View Plane – specified with sides parallel to the U and V axes. Rays from the COP through the four corners of this window determine the *View Volume* as a (possibly irregular) double infinite pyramid. The View Volume is analogous to a cone of vision, it includes only what can be seen from this particular view in the scene. Clearly, the View Volume acts as a 3D clipping region.

- Front and Back Clipping Planes. The construction so far does not actually exclude objects in the scene which are behind the COP. In addition, the observer may wish to exclude parts of the scene which are "too close" or "too far away" from the COP. This can be accomplished by specifying Front and Back Clipping Planes. These are planes which are parallel to the View Plane, and placed at specified distances from the VP.

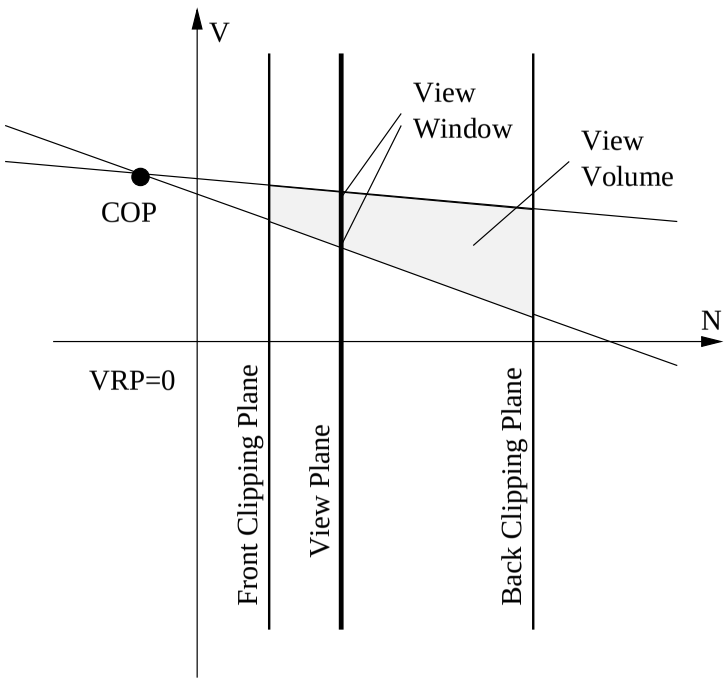

Figure 2.2: Perspective Projection in VC.

Consider transforming the viewing area (in viewing coordinates) to a canonical space. A number of steps are involved in constructing a matrix to transform the system to the *Canonical Coordinate System*.

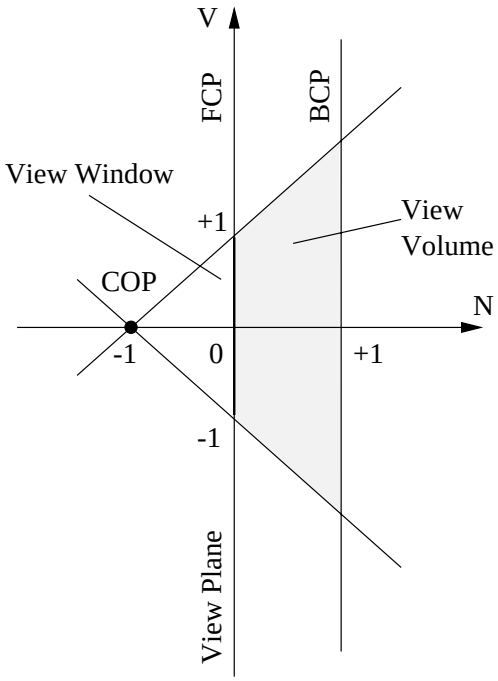

Figure 2.3: Canonical View for Perspective Projection.

2.2.3 The Viewing Pipeline

The Viewing Pipeline is the transformation of WC points to the final projection space. See Figure 2.4.

2.3 Graphics Data Structures Representing Polyhedra

In 3D space we may have objects that are of 0, 1, 2, or 3 dimensions: that is a point, lines and curves, surfaces (occupying area), and solids (occupying volume). The only kind of surface we will consider here are so-called planar surfaces – that is surfaces that are inscribed on a plane. A plane can be thought of as an infinite flat sheet. The particular type of planar surfaces that we shall use are polygons.

Any three distinct points always lie on the same plane. However, a fourth point may not lie on the same plane as that defined by the first three. This is a very strong restriction and the reason why in practice, in computer graphics there is a preference for three sided polygons. There is also a strong preference

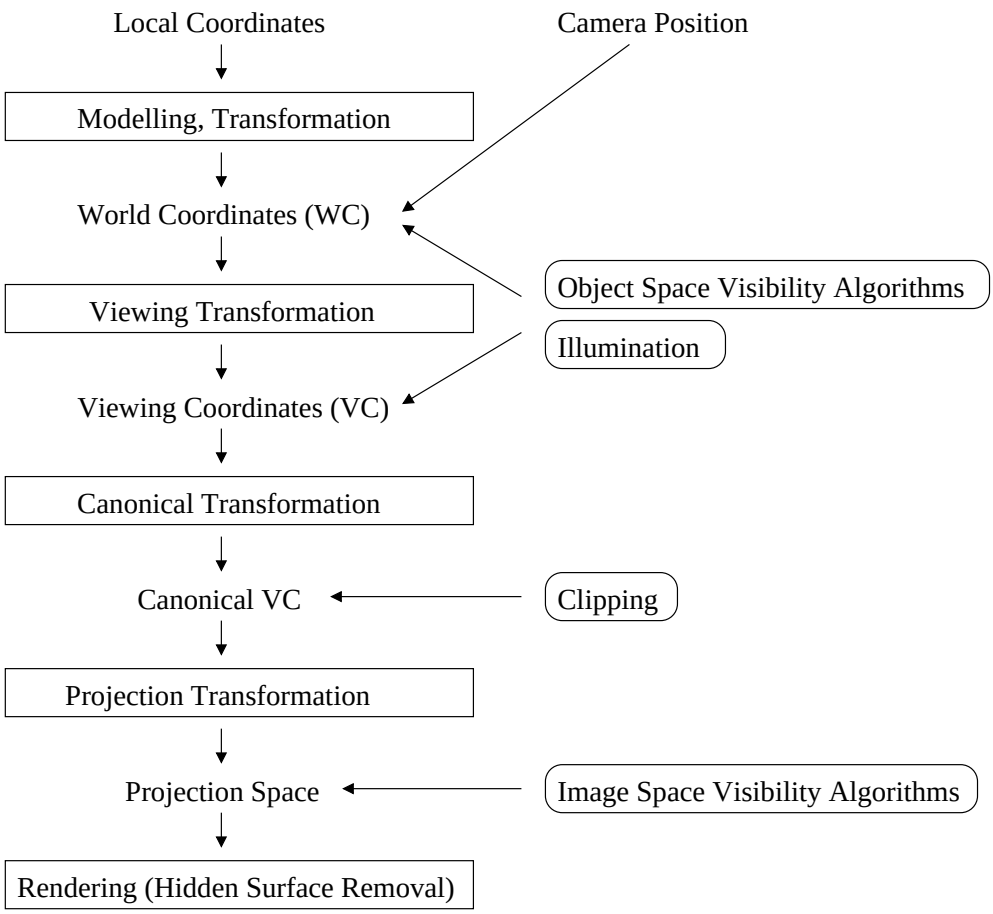

Figure 2.4: The Viewing Pipeline.

for convex polygons. In computer graphics polygons are often called *faces* – this is we think of a complex object being broken up into lots of faces.

The simplest data structure for representing polyhedra would be a collection of separate polygons. This is not very efficient in terms of space, nor useful in terms of modelling. For example, it is clear that each vertex would be stored at least three times, and each edge twice. The following is an example of what can be called the *Indexed Faceset data structure*.

The data structure involves two sequences, one a list of all vertices in the object, and the other a list of all faces, constructed as a sequence of entries referencing the vertices in the vertex sequence. This structure can be made richer by having an edge list. It can be made even more complex, with each

vertex maintaining pointers to the edges and faces in which it participates, and so on.

This is only a simple data structure and it must be considered if it supports all the necessary features for future use. For example, if it does not support the capability of finding all of those faces which share a vertex, then this operation could only be implemented by a very inefficient searching strategy if needed.

Chapter 3

Visibility Techniques

3.1 Introduction

After knowing about the basics of 3D graphics (see Chapter 2) the next problem, the visibility, needs to be addressed. We know how a point or an object is projected to the view plane (the screen) but we don't know yet how the determine which one of the possible candidates (more than one point can lie on the same ray and therefore on the same position on the view plane) is the one that should be visible to the user. Intuitively we would say the closest point to the camera is the one that should be rendered – but how can we determine this "closest point" and objects that are hidden by closer ones. Different approaches are explained in the following sections.

3.2 Basics of Visibility and Rendering

The visibility problem (also called hidden surface removal) is concerned with computing and rendering those parts of a scene which are visible to the camera. This simulates the situation with real vision: an observer of a scene can only see those parts from which light can be traced unobstructed to his or her eyes.

It is usual to characterize such algorithms according to whether they process in *image space* or *object space*. Object space algorithms solve the hidden surface algorithm in the 3D space of the scene. Image space algorithms defer solution until the last instance – i.e., visibility calculations, performed on a pixel by pixel basis, are performed at the moment just before setting the intensity of the pixel on the display. Some algorithms provide a mixture of both approaches.

3.2.1 Techniques for Efficient Visible-Surface Algorithms

To minimize the time that it takes to create a picture, we must organize visible-surface algorithms so that costly operations are performed as efficiently and as infrequently as possible. The following sections describe some general ways to do this, based on Foley et al [FvDFH90].

Coherence

Visible-surface algorithms can take advantage of *coherence* – the degree to which parts of an environment or its projection exhibit local similarities. We exploit coherence when we reuse calculations made for one part of the environment or picture for other nearby parts, either without changes or with incremental changes that are more efficient to make than recalculating the information from scratch. Some of the known kinds of coherence:

- Object coherence. If one object is entirely separate from another, comparisons may need to be done only between the two objects, and not between their component faces or edges.
- Scan-line coherence. The set of visible object spans determined for one scan line of an image typically differs little from the set on the previous line.
- Area coherence. A group of adjacent pixels is often covered by the same visible face.
- Depth coherence. Adjacent parts of the same surface are typically close in depth, whereas different surfaces at the same screen location are typically separated farther in depth.
- Frame coherence. Pictures of the same environment at two successive points in time are likely to be quite similar, despite small changes in objects and viewpoint. Calculations made for one picture can be reused for the next in a sequence.

Extents and Bounding Volumes

Screen extents, the minimum bounding rectangle of a planar object, and bounding volumes, the minimum bounding box of a 3D-object, can be used for trivial intersection testing between two objects, but also to determine whether or not a projector or a ray intersects an object.

Back-Face Culling

If an object is approximated by a solid polyhedron, then its polygonal faces completely enclose its volume. Assume that all the polygons have been defined such that their surface normals point out of their polyhedron. Those polygons whose surface normals point away from the observer lie on a part of the polyhedron whose visibility is completely blocked by other closer polygons of its own. Such invisible back-facing polygons can be eliminated from further processing, a technique known as *back-face culling*. By analogy, those polygons that are not back-facing are often called *front facing*.

A back-facing polygon may be identified by the positive dot product that its surface normal forms with the vector from the center of projection (the eye of the camera) to any point on the polygon.

Spatial Partitioning

Spatial partitioning (also known as spatial subdivision) allows us to break down a large problem into a number of smaller ones. The basic approach is to assign objects or their projections to spatially coherent groups as a preprocessing step. For example, we can divide the view plane with a coarse, regular 2D rectangular grid and determine in which grid spaces each object's projection lies. Projections need to be compared for overlap with only those other projections that fall within their grid boxes. Spatial partitioning can be used to impose a regular 3D grid on the objects in the environment. The process of determining which objects intersect with a projector or a ray can then be sped up by first determining which partitions the projector intersects, and then testing only the objects lying within those partitions.

Hierarchy

Hierarchies can be useful for relating the structure of different objects. A nested hierarchical model, in which each child is considered part of its parent, can also be used to restrict the number of object comparisons needed by a visible-surface algorithm. An object on one level of the hierarchy can serve as an extent for its children if they are entirely contained within it. In this case, if two objects fail to intersect, the lower-level objects of one do not need to be tested for intersection with those of the other.

3.2.2 Algorithms to Solve the Visibility Problem

After having discussed a number of general techniques, we introduce some visibility algorithms, focusing on *Recursive Subdivision*, the *Z-Buffer Algorithm* and the *Binary Space Partitioning Tree* and only mention earlier visibility algorithms like *Roberts' Algorithm* [Rob63], *Appel's Algorithm* [App67], or the *Scan-Line Algorithm* [WREE67] which is an extension of the polygon scan-conversion algorithm.

Recursive Subdivision

This is a divide and conquer algorithm due to Warnock [War69]. This kind of approach is often used in computer graphics. The clipping rectangle on the X-Y plane represents the view window of the camera model (as transformed into Projection Space, PS). If this rectangular window contains a scene in which the hidden surface problem is easily solved, then solve it and render the scene; otherwise split the rectangle into four quadrants, and recursively apply the same principle to each.

Painters' Algorithm

This algorithm [NNS72] makes use of the overpainting properties of raster displays to eliminate hidden surfaces. It renders the polygons in the scene in back-to-front order, so that polygons rendered later overpaint those rendered earlier. Those rendered earlier are those furthest from the camera. Hence the algorithm involves sorting the polygons in an order so that if Q is before P in this ordering, that P does not obscure Q.

Painters' algorithm is of interest for historical reasons only now. In particular, it suffers from the problem that if the viewpoint is changed, then the entire hidden surface computation has to be repeated. An alternative method which performs the computations in object space, but relies on the overpainting properties of raster displays, and which does not require an entirely new computation when the viewport is changed, is based on the idea of *Binary Space Partition Trees*. This is considered in the following sections.

Z-Buffer Algorithm

The *Z-buffer* or *Depth-Buffer* image-precision algorithm, developed by Catmull [Cut74], is one of the simplest visible-surface algorithms to implement in either software or hardware. It requires that we have available not only a frame buffer

F in which colour values are stored, but also a z-buffer Z, with the same number of entries, in which a z-value is stored for each pixel. The z-buffer is initialized to a value representing the z-value at the back clipping plane, and the frame buffer is initialized to the background colour. The values that can be stored in the z-buffer represent the range from the back clipping plane to the front clipping plane. Polygons are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the polygon point being scan-converted at (x,y) is no farther from the viewer than is the point whose colour and depth are currently in the buffers, then the new point's colour and depth replace the old values.

No presorting is necessary and no object-object comparisons are required. The entire process is no more than a search over each set of pairs $\{Z_i(x,y), F_i(x,y)\}$ for fixed x and y, to find the closest Z_i. The z-buffer and the frame buffer record the information associated with the closest z encountered thus far for each (x,y).

Even after the image has been rendered, the z-buffer can still be used to advantage. Since it is the only data structure used by the visible-surface algorithm, it can be saved along with the image and used later to merge in other objects whose z can be computed.

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

+

.5	.5	.5	.5	.5	.5	.5
.5	.5	.5	.5	.5	.5	
.5	.5	.5	.5	.5		
.5	.5	.5	.5			
.5	.5	.5				
.5	.5					
.5						

=

.5	.5	.5	.5	.5	.5	.5	1
.5	.5	.5	.5	.5	.5	1	1
.5	.5	.5	.5	.5	1	1	1
.5	.5	.5	.5	1	1	1	1
.5	.5	.5	1	1	1	1	1
.5	.5	1	1	1	1	1	1
.5	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

.5	.5	.5	.5	.5	.5	.5	1
.5	.5	.5	.5	.5	.5	1	1
.5	.5	.5	.5	.5	1	1	1
.5	.5	.5	.5	1	1	1	1
.5	.5	.5	1	1	1	1	1
.5	.5	1	1	1	1	1	1
.5	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

+

.3					
.4	.3				
.5	.4	.3			
.6	.5	.4	.3		
.7	.6	.5	.4	.3	
.8	.7	.6	.5	.4	.3

=

.5	.5	.5	.5	.5	.5	.5	1
.3	.5	.5	.5	.5	.5	1	1
.4	.3	.5	.5	.5	1	1	1
.5	.4	.3	.5	1	1	1	1
.5	.5	.4	.3	1	1	1	1
.5	.5	.5	.4	.3	1	1	1
.5	.7	.6	.5	.4	.3	1	1
1	1	1	1	1	1	1	1

Figure 3.1: The Z-Buffer. A pixel's shade is shown as its colour, its z value is shown as a number (1 represents the BCP, 0 the FCP). First adding a polygon to the empty z-buffer, then adding another polygon that intersects the first.

BSP-Tree – Binary Space-Partitioning Tree

The binary space-partitioning (BSP) tree algorithm, developed by Fuchs, Kedam, and Naylor [FKN80], is an extremely efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the viewpoint changes, but the objects do not.

The BSP-tree algorithm is based on the work of Schumacker [SBGS69], who noted that *given a scene of polygons, and a COP, anything behind a plane cannot obscure anything in front.*

In order to determine the visible surface at each pixel, traditionally the distance from the viewing position to each polygon which maps onto that pixel is calculated. The BSP-tree approach eliminates these distance calculation entirely. Rather, it transforms the polygonal data base (splitting polygons when necessary) into a binary tree which can be traversed at image generation time to yield a *visibility priority* value for each polygon. These visibility priorities are assigned in such a way that at each pixel the closest polygon to the viewing position will be the one with the highest priority.

An obvious alternative, which was used by Fuchs, does not assign explicit visibility priority values to polygons but uses the traversal to drive a *painter's algorithm* which paints onto the screen's image buffer each polygon as it is encountered in the traversal. Since higher priority polygons are visited later, they will overwrite any overlapping polygons of lower priority.

Preprocessing Phase. Let us now consider the set of polygons $P = \{P_1, P_2, \ldots, P_n\}$ which define the 3D environment. Choose an arbitrary polygon P_k from this set. The plane in which this polygon lies partitions the rest of the 3-space into two half-spaces. Call these S_k *and* $S_{\overline{k}}$. The two half-spaces are identified with the positive and negative sides of the polygon P_k. If P_k was defined with a *front* side, then that side is considered as the positive one.

What can be said about visibility priorities of these polygons? We know that if the viewing position is in one half-space, say in S_k, that no polygon within $S_{\overline{k}}$ can obstruct either polygon P_k or any polygon in S_k.

Therefore, we split each of the polygons in $P - \{P_k\}$ along the plane of P_k, putting the polygons (or parts of them) which lie in S_k into one set and the polygons which lie in $S_{\overline{k}}$ into another set. Polygons coplanar with P_k can be put in either set. We can represent the result of this splitting process by a binary

tree – the binary space partitioning tree – in which the root contains P_k and each branch's subtree contains the set of polygons associated with one of the half-spaces.

We next consider one of the two new sets of polygons, say S_k. We remove a polygon, say P_j and split the remaining polygons in S_k along the plane of P_j, putting those polygons (or parts thereof) lying on the positive side in one set $S_{k,j}$ and those lying on the negative side in another set $S_{k,\bar{j}}$.

To complete the construction of the BSP-tree we continue splitting sets until no non-null sets remain.

This recursive process is only performed once for all possible images from all viewing positions; the tree remains valid as long as the scene doesn't change.

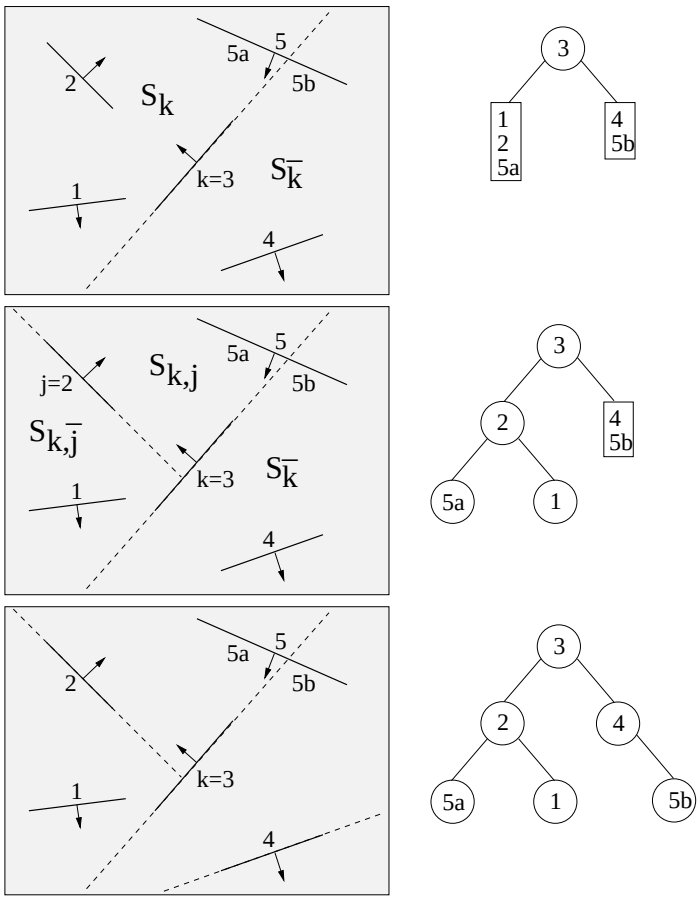

Figure 3.2: Building a BSP-tree with polygon 3 as root in 2D.

Image Generation Phase. Applying a *painter's algorithm*, once the viewing position is known, is the variant of an in-order-traversal of the environment's BSP-tree we want to use (traverse one subtree, visit the root, traverse the other subtree). We wish, for example, to have an order of traversal that visits the polygons from those farthest away to those closest to the current viewing position.

At any given node, there are two possibilities: positive side subtree, node, negative side subtree or the other way round. We choose one of these two orderings based on the relationship of the current viewing position to the node's polygon. Specifically, we are interested in the side (positive or negative) of the node's polygon where the current viewing position is located. Let's call the two sides the *containing* side and the *other* side. The traversal for a back-to-front ordering is the *other* side, the *node*, and the *containing* side. This side-of-node-polygon determination is, of course, just a check of the sign of the Z component of the node polygon's normal vector after the usual transformation to projection space.

Set Operations on BSP-Trees. Thibault and Naylor [TN87] introduced the concept of using a BSP-tree to represent polyhedral solids. They associate an *in* or *out* value with each empty region at the leaves. Assuming that a polyhedron's normals point outward, then an *in* region corresponds to the half-space on the polygon's back, and an *out* region corresponds to the half-space on the polygon's front side. Each internal node defines a plane and has a list of polygons embedded in the plane. The *in* and *out* regions from a convex polyhedral tessalation of space. Thus, a BSP-tree can represent an arbitrary (possible concave) solid with holes as a union of convex *in* regions. Thibault and Naylor show how to produce a BSP-tree from a polygonal boundary representation of a solid and how to perform Boolean set operations on two boundary representations to yield a new BSP-tree.

Clipping with the BSP-tree. It is another use of the BSP-tree to clip a polygon through passing it down the tree representing the clipping area with *in* and *out* leaves. The parts of the polygon in the clipping area end up in *in* leaves and the parts outside the clipping area (the clipped parts) end up in *out* leaves.

BSP-tree compared with Z-buffer

The Z-buffer algorithm is very easy to implement in hardware and in this case a very fast way of rendering, especially if the scene is manipulated, because it is incremental if a polygon is added to the scene. The software implementation

of the Z-buffer algorithm can not be as fast as the hardware-implementation but still has the advantage of being incremental.

The BSP-tree algorithm is very easy to implement in software (simple recursive calls). It takes full advantage of static scenes with only the viewing point changing because of having an extensive preprocessing that is done only once. The image generation is very fast and takes only $O(n)$ time.

3.3 Efficient Rendering Approaches

3.3.1 Introduction

Much work has been done to create visibility algorithms for specific situations like scenes containing axis-aligned cells [TS91]. But there are some very effective general algorithms considering *object-space coherence*, *image-space coherence*, and *temporal coherence* for static scenes. The overhead of each algorithm depends on the size and the depth complexity of the scene.

3.3.2 Binary Space Partitioning Tree

This already above mentioned method by Fuchs [FKN80] is well suited for small static scenes without any visibility calculation in addition to the hidden surface elimination through drawing all polygons in the scene in back-to-front order.

3.3.3 Good Partitioning Tree

If the only operation on the BSP-tree entails visiting the entire tree, as with visibility priority (back-to-front traversal), then the tree size is the only measure needed for determination of the goodness of a tree. Once spatial search operations are introduced, such as ray tracing or set operations (which include clipping and collision detection), in which only a subset of the tree need be visited, the shape of the tree becomes at least as important as size, and probably more important [Nay93].

By using decision probabilities rather than leaf probabilities, we see that locally the best situation at an internal region is one in which we have a partitioning in which most of the information is in a small sub-region(a low probability region), and as little as possible is in a large region. Thus, balanced is not optimal in general, where balanced means equal sized trees with equal probability of being selected. A rule of thumb could say: Construct short paths to large homogeneous regions.

Considering the cost of building a tree, Torres [Tor90] inserts logical planes in the scene to subdivide the 3D space and to separate objects, which are represented by local BSP-trees, so called *Single Trees* that are inserted in the global tree when needed. The resulting structure is a tree of auxiliary planes whose leaves are the single trees of the objects. These new structured BSP-trees are called *dynamic BSP-trees*. The new structure and the significant reduction in the generation time of the tree allow the management of dynamic modifications – only those subtrees of the BSP-tree affected by the modification will be recomputed.

3.3.4 Object-Space Visible-Surface Determination

The SVBSP Algorithm. This approach by Chin and Feiner [CF89] describes an object-space shadow generation algorithm for static polygonal environments illuminated by movable point light sources. Alternatively, the algorithm may also be used to perform object-space visible-surface determination by placing the light source at the eyepoint and returning the list of the non-overlapping lit (visible) polygon fragments that are computed. The algorithm clips the scene polygons against the shadow volume in object space, creating the shadow volume as it proceeds.

The Shadow Volume BSP (SVBSP) tree is a modified version of the general BSP-tree. Each internal node is associated with a *shadow plane* defined by the COP and an edge of a polygon facing the COP. This is one of the *shadow polygons* that Crow refers to as bounding the shadow volume [Cro77]. The direction of the plane's normal is used to determine the half-space in which an object is located. At the leaves are the *in* and *out* cells indicating whether or not a region is interior to the shadow volume.

There are two basic steps to the SVBSP algorithm whose execution is interleaved for each polygon facing the COP:

- Determining shadows. The polygon is filtered down the SVBSP-tree to determine those parts that are shadowed and those that are not.

- Enlarging the SVBSP-tree. The shadow volume for each of the polygon's lit parts is created and added to the SVBSP-tree.

Polygons that are further from the COP than the polygon being tested cannot obstruct it. Therefore, if we process the polygons in a front-to-back order relative to the COP, then each polygon would only have to be compared with the shadow volumes of those polygons that have already been processed and which are closer to the COP than it. The font-to-back ordering can be determined by building

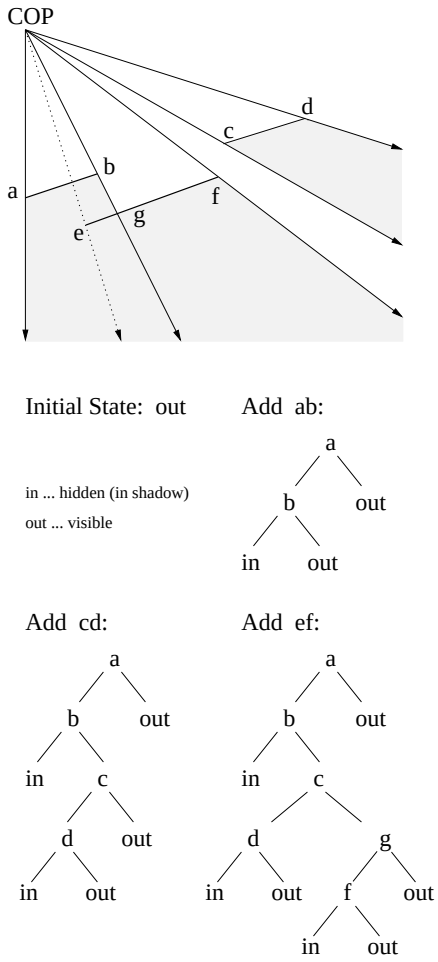

Figure 3.3: Building an SVBSP volume in 2D.

a regular BSP-tree from the original scene polygons and traversing it from the COP. Note, that this BSP-tree needs to be created only once at the outset if no polygons change in the scene.

This algorithm returns the list of the non-overlapping lit (visible) polygon fragments, which can be displayed in arbitrary order.

This algorithm can be broken up earlier to reduce the overhead but producing a non-definitive visibility for the following polygons filtered down the *incomplete* SVBSP-tree to reject completely hidden surfaces. Therefore the outcome of the algorithm would be a list of visible, non overlapping polygon fragments in the SVBSP-tree (being in front) and (partly) visible, probably overlapping polygons

in the remaining BSP-tree, representing the whole scene (marked as not yet processed), which would have to be rendered in the usual back-to-front order first.

Note that it is not necessary to filter any polygon that doesn't face the COP, since it is already eliminated through back face culling.

3.3.5 Hierarchical Z-Buffer Visibility

This algorithm by Green, Kass, and Miller [GKM93] exploits object-space, image-space, and temporal coherence. In object-space, an octree spatial subdivision of the type commonly used to accelerate ray tracing is used. To exploit image-space coherence, traditional Z-buffer scan conversion is augmented with an image-space Z pyramid that allows us to reject hidden geometry very quickly. Finally, to exploit temporal coherence, the geometry that was visible in the previous frame is used to construct a starting point for the algorithm. The result is an algorithm which is orders of magnitude faster than traditional ray-casting or Z buffering for complex models. The algorithm is not difficult to implement and works for arbitrary polygonal databases.

Object-Space Octree. In order to be precise about the octree algorithm, let us begin with some simple definitions. We will say that a polygon is hidden with respect to a Z buffer if no pixel of the polygon is closer to the observer than the Z value already in the Z buffer. Similarly, we will say that a cube is hidden with respect to a Z buffer if all of its faces are hidden polygons. Finally, we will call a node of an octree hidden if its associated cube is hidden. Note that these definitions depend on the sampling of the Z buffer. A polygon which is hidden at one Z-buffer resolution may not be hidden at another.

If a cube is hidden with respect to a Z buffer, then all polygons fully contained in the cube are also hidden. From this observation, the basic algorithm is easy to construct. We begin by placing the geometry into an octree, associating each primitive with the smallest enclosing octree cube. Then we start at the root node of the octree and render it using the following recursive steps: First, we check to see if the octree cube intersects the viewing frustum. If not, we are done. If the cube does intersect the viewing volume, we scan convert the faces of the cube to determine whether or not the whole cube is hidden, If the cube is hidden, we are done. Otherwise, we scan convert any geometry associated with the cube and then recursively render its children in front-to-back order.

Image-Space Z Pyramid. The basic idea of the Z pyramid is to use the original Z buffer as the finest level in the pyramid and then combine four Z

values at each level into one Z value at the next coarser level by choosing the farthest Z from the observer. every entry in the pyramid therefore represents the farthest Z for a square area of the Z buffer. At the coarsest level of the pyramid there is a single Z value which is the farthest Z from the observer in the whole image.

In order to use the Z pyramid to test the visibility of a polygon, we find the finest-level sample of the pyramid whose corresponding image region covers the screen-space bounding box of the polygon. If the nearest Z value of the polygon is farther away than this sample in the Z pyramid, we know immediately that the polygon is hidden. We use this basic test to determine the visibility of octree cubes by testing their polygonal faces, and also to test the visibility of model polygons.

A definitive visibility test can be constructed by applying the basic test recursively through the pyramid. When the basic test fails to show that a polygon is hidden, we go to the next finer level in the pyramid where the previous pyramid region is divided into four quadrants. Here we attempt to prove that the polygon is hidden in each of the quadrants it intersects. Ultimately, we either prove that the entire polygon is hidden, or we recurse down to the finest level of the pyramid and find a visible pixel. If we find all visible pixels this way, we are performing scan conversion hierarchically.

Temporal Coherence List. We maintain a list of the visible cubes from the previous frame, the *temporal coherence list*, and simply render all of the geometry on the list, marking the listed cubes as rendered, before commencing the usual algorithm. We then take the resulting Z buffer and use it to form the initial Z pyramid. If there is sufficient frame-to-frame coherence, most of the visible geometry will already be rendered, so the Z pyramid test will be much more effective than when we start from scratch. The Z pyramid test will be able to prove with less recursions that octree cubes and model polygons are hidden.

3.4 Visibility in VRweb

As described in Section 5, the graphics output in VRweb is done via an abstract interface (GE3D) based on OpenGL, to one of several underlying graphics libraries. GE3D, because of OpenGL, does not provide an interface between the data input (objects in object-coordinates, light sources, transformation on the object to transform to WC, and the camera coordinates) and the actual display through a Z buffer. The only way to change the rendering strategy is to dis-

able the Z buffer functionality and implement another visibility algorithm. To go round the object coordinates it is possible to set the transformation matrix for GE3D to I (not doing anything) and implement another transformation before passing the object coordinates to the graphics interface. Therefore it is possible to feed GE3D with world coordinates with or without Z buffering. GE3D transforms the coordinates to the camera- and viewing coordinate system depending on the current camera position.

With Hardware Graphics Support

The *Hierarchical Z Buffer* algorithm [GKM93] seems to be ideal for machines with hardware Z buffer support, especially for large scenes with a high depth complexity. The object space octree can be replaced by a *good partitioned tree* [Nay93] to speed up this algorithm for scenes where it's polygons are not regularly distributed throughout the scene, which would provide the possibility to introduce a good search and replace algorithm if done like proposed by Torres [Tor90].

Without Hardware Graphics Support

The *SVBSP-tree* algorithm [CF89] seems to be a good alternative to the hierarchical Z buffer algorithm for machines without hardware Z buffer support, completely based on BSP-trees. An object space partitioning like with the hierarchical Z buffer algorithm, but still based on BSP-trees (order!), would speed up the trivial visibility reject before filtering all polygons down the SVBSP-tree to determine their visibility status.

Visibility Approaches

The two main groups of implementing visibility algorithms are either based on the Z buffer (like the *Hierarchical Z Buffer*) or on back-to-front traversal using BSP-trees (like the *Shadow Volume BSP-tree algorithm*). The differences, advantages and disadvantages were mentioned earlier.

Variety of Platforms

VRweb is implemented for a variety of platforms. There are 2 different groups of machines, the one with hardware graphics support and the one with standard workstations or PCs without any hardware support for graphics. Therefore it might be necessary to implement 2 different algorithms to take full advantage

in case of the hardware acceleration, and on the other hand to avoid extensive rendering calculations on standard machines, through using visibility algorithms to reject hidden parts from being rendered.

What is the right thing for VRweb?

My aim is to speed up rendering for standard workstations, therefore the SVBSP-tree algorithm seems to be a step into the right direction. An evaluation of this strategy will show the overhead compared with the speeding up through saving display time for hidden faces, depending on the size and the depth of complexity of the scene.

Chapter 4

3D Graphics Libraries and APIs

Here is a short overview of some of the more important 3D-libraries and their application programming interfaces (APIs):

4.1 OpenGL

OpenGL is a software interface for applications to generate interactive 2D and 3D computer graphics. OpenGL is designed to be independent of operating system, window system, and hardware operations, and it is supported by many vendors. OpenGL is available on PCs and workstations. OpenGL provides a wide range of graphics functions: from rendering a simple geometric point, line, or filled polygon, to texture mapping NURBS curved surfaces.

- Geometric primitives (points, lines, and polygons)
- Raster primitives (bitmaps and pixel rectangles)
- RGBA or colour index mode
- Display list or immediate mode
- Viewing and modeling transformations
- Hidden Surface Removal (depth buffer)
- Alpha Blending (transparency)
- Antialiasing
- Texture Mapping
- Atmospheric Effects (fog, smoke, and haze)
- Polynomial Evaluators (to support Non-uniform rational B-splines)
- Pixel Operations (storing, transforming, mapping, zooming)
- Accumulation Buffer
- Stencil Planes
- Feedback, Selection, and Picking

The OpenGL functions described are provided on every OpenGL implementation to make applications written with OpenGL easily portable between platforms. All licensed OpenGL implementations are required to pass the Conformance Tests, and come from a single specification and language binding document.

4.1.1 OpenGL as an Application Programming Interface (API)

While the X Window System has become the de facto display management standard in the UNIX workstation market, OpenGL was designed to be window system neutral and vendor neutral. This means that other windowing environments, like Windows 95 or Windows NT, are also well suited to support OpenGL. This is a very important benefit, and is possible since the OpenGL specification is independent of

1. window system
2. operating system
3. network

In addition, all implementations of OpenGL, regardless of vendor, must be fully conforming to the basic standard, so the user/programmer is assured that all basic features and functions will be available on all different platforms. Each implementor is required to run and pass a suite of conformance tests to ensure source code compatibility across all OpenGL implementations.

OpenGL itself is controlled by an industry consortium, the OpenGL Architectural Review Board. Members from major hardware and software vendors jointly guide the growth and development of OpenGL, thus assuring that no one vendor dictate the direction on OpenGL.

4.1.2 What OpenGL is Not

OpenGL is not:

- A Tool Kit or high-level Application Programming Interface
- A Windowing System
- A Descriptive Graphics System
- Object Oriented

OpenGL knows nothing of tool kits, although there are tool kits for OpenGL. It is also not a windowing system, and is dependent on a window system to do all the window related tasks (like creating the canvas, dealing with input from the user, etc.). OpenGL is not descriptive, that is the programmer does not set up a model of the scene to be rendered, and then let the graphics system handle the task of doing the drawing. And finally, OpenGL is not object oriented (but there is a object-oriented 3D graphics toolkit, Open Inventor, available from Silicon Graphics, for example).

4.1.3 What OpenGL is

OpenGL is:

- An Immediate Mode System
- Application Format Flexible
- A Procedural System
- Display List Functional

OpenGL is an immediate mode system, that is commands are executed essentially immediately. OpenGL accepts various data types, freeing the programmer from needless conversions. It is procedural in that the programmer issues specific commands to determine what is actually drawn. And it supports non-editable display lists for better network extensibility, resulting in improved performance across server/client applications, as well as improved direct rendering.

4.1.4 How OpenGL Solves the Interface Dilemma

Since OpenGL is window system independent, it does not have to be concerned (too much) with the window system details itself. For each window system which OpenGL is adapted to, a very limited set of "glue" routines are specified, and the remaining core of OpenGL remains identical across all window systems.

This core API is dedicated to the basic rendering functionality, but there exist a number of other aspects of graphics which are not strictly rendering. To address these needs, the OpenGL standard also provides the OpenGL Utility Library (GLU), with routines for the following tasks:

- Transforming coordinates
- Tessellating polygons
- Manipulating images for texture applications

- Rendering canonical shapes, like spheres, cylinders and disks
- Non-uniform rational B-spline (NURBS) curves and surfaces
- Error reporting

When considering the X Window System, OpenGL provides a formal X Extension - GLX. GLX provides the means of "gluing" together OpenGL and X. It also defines the wire protocol for supporting OpenGL as an X server extension, thus allowing an OpenGL application running on one workstation to be rendered on another host, much the same way the standard X protocol supports 2D graphics interoperability.

4.1.5 Further Reading

A 2 volume set, The OpenGL Technical Library, is published by Addison-Wesley. The OpenGL Reference Manual is ISBN 0-201-63276-4. The OpenGL Programming Guide is ISBN 0-201-63274-8.

Also, the USENET news group comp.graphics.opengl is a wealth of information regarding OpenGL aspects. Some additional references are:

1. Davis, Tom. "Moving to OpenGL," IRIS Universe, Number 25, Summer, 1993.

2. Glazier, Bill. "The 'Best Principle': Why OpenGL is emerging as the 3D graphics standard," Computer Graphics World, April, 1992.

3. Karlton, Phil. "Integrating the GL into the X environment: a high performance rendering extension working with and not against X," The X Resource: Proceeding of the 6th Annual X Technical Conference, O'Reilly Associates, Issue 1, Winter, 1992.

4. Kilgard, Mark J. "OpenGL & X: An Introduction," The X Journal. November-December, 1993, page 36-51.

5. Kilgard, Mark J. "Using OpenGL with Xlib," The X Journal. January-February, 1994, page 46-65.

6. Kilgard, Mark J. "OpenGL and Motif Integration," The X Journal. to appear.

7. Neider, Jackie, Tom Davis, and Mason Woo, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1, Addison-Wesley, Reading, Massachusetts, 1993.

8. OpenGL Architecture Review Board, OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1, Addison-Wesley, Reading, Massachusetts, 1992.

9. "OpenGL Programs a New Horizon for Sun," SunWorld, January, 1994, page 15-17.

4.2 Mesa

Mesa [Pau] is a 3-D graphics library with an API which is very similar to that of OpenGL. To the extent that Mesa utilizes the OpenGL command syntax or state machine, it is being used with authorization from Silicon Graphics, Inc. However, the author makes no claim that Mesa is in any way a compatible replacement for OpenGL or associated with Silicon Graphics, Inc. Those who want a licensed implementation of OpenGL should contact a licenced vendor. This software is distributed under the terms of the GNU Library General Public Licence.

Mesa's performance is directly related to CPU and X server performance. Several thousand lit, smooth-shaded, depth-buffered triangles per second is attainable on most systems. Recent releases of Mesa have featured speed optimizations.

There is no support for 3-D graphics hardware at this time. Implementing such support would require detailed information about the interface to the hardware.

The following features are fully implemented:

- all point, line and polygon primitives
- all model and view transformations
- lighting
- smooth shading
- depth buffering
- clipping (against user clip planes and view volume)
- accumulation buffer
- alpha testing/blending
- stencil buffer
- dithering
- logic operations
- evaluators (curves and surfaces)

- feedback/selection
- fog/depth cuing
- polygon/line stippling
- read/write/copy pixels
- tk and aux libraries for X11
- context switching (multiple windows)
- RGB mode simulated in colour mapped windows (8-bit and monochrome)
- Support for Mark Kilgard's GLUT
- blending extensions
- a set of pseudo-GLX functions
- glPixelZoom
- display lists

The following features are mostly/partially implemented:

- texture mapping (50% complete)
- the GLU utility library
- NURBS

4.2.1 How to get Mesa

Mesa is available by anonymous ftp from:

```
Host: iris.ssec.wisc.edu, Directory: pub/Mesa/
Filename: Mesa-1.2.8.tar.Z or Mesa-1.2.8.tar.gz
```

For WWW users, see:

```
iris.ssec.wisc.edu/pub/Mesa
```

Mesa is also mirrored on:

```
sunsite.unc.edu in
pub/packages/development/graphics/mesa or in pub/Linux/libs/X.
```

The distribution includes complete source code, X11 and Windows drivers and many demonstration programs.

4.2.2 Documentation

Since the OpenGL API is used, OpenGL documentation can serve as the documentation for Mesa's core functions. Specifically for Mesa are:

- README file
  `ftp://iris.ssec.wisc.edu/pub/Mesa/README`
- Mesa FAQ
  `http://www.ssec.wisc.edu/ brianp/MesaFAQ.html`
- Mesa User's Guide
  `http://www.ssec.wisc.edu/ brianp/MesaUsersGuide.html`
- Mesa Implementation Notes
  `http://www.ssec.wisc.edu/ brianp/Implementation.html`
- Mac/Mesa WWW page
  `http://www.elte.hu/ boga/Mesa.html`
- Amiga Mesa WWW page
  `http://www.efd.lth.se/ d94sz/amesa`

Mailing list:
Pedro Vasquez has setup the Mesa mailing list. To subscribe, send the following message body to listproc@iqm.unicamp.br
subs mesa (your name)

4.3 GE3D

The GE3D library – Graphics Engine 3D – was designed as a machine independent, immediate mode, 3D graphics interface by Michael Pichler [Pic93] for use by the Harmony 3D Scene Viewer, the precursor of VRweb. GE3D was further extended by Martin Eyl [Eyl95]. The first version of GE3D was developed together with Michael Hofer. The functionality of GE3D includes:

- manipulation of vectors and matrices, and a stack of transformation matrices
- camera definition, both perspective and orthographic
- definition of light sources

- double buffering (two screen pages)
- drawings in wire frame, hidden line, flat shaded, smooth shaded and textured
- drawing of 3D faces (polygons), polyhedra, cubes, spheres, cones, and cylinders
- drawing some 2D primitives: lines, rectangles, arcs, circles, output of text

GE3D's functionality is on a higher level than typical machine dependent libraries like GL or OpenGL. The drawing modes do not bother the user of the library with correct settings of flags for hidden surface elimination, filling, usage of a Z-Buffer and so on. Other functions like manipulation of the matrix stack have corresponding counterparts in low level graphic libraries.

Beside the mentioned enlarged functionality the machine independency of the interface increases the portability of the programs. As the header file is completely independent of any other header file of graphics interfaces, another implementation of the GE3D library can be linked without changes or recompilation of existing programs.

As the structure of VRML is very much tailored to the capabilities of OpenGL [NDW93], the mapping of VRML primitives to the drawing commands of GE3D in VRweb is straightforward.

4.4 RenderWare

RenderWare is a portable 3D API for DOS, Windows 3.1, Windows 95 and PowerMac. It was created for game developers to free them from concerns about hardware acceleration on different platforms. The claimed performance for software rendering on a Pentium 90 is a flat shading pixel fill rate of 27 million pixels/second and a perspective correct texture pixel fill rate of 6.2 million pixels/second.

- Z buffer and scanline renderer
- Multiple coloured lightsources (distant, point and spot lights)
- Gouraud lighted, dithered, transparent, perspective texturemapping.
- Animated textures and environment mapping.
- Object picking.
- Materials support.
- 3D sprites.

- Depth cueing.
- Support for graphics accelerator cards like the GLINT chip.
- Support for stereo glasses and headsets.
- DFX, 3DS and VRML converters.
- 900 pages documentation

More information is available on the Criterion Software home page at `http://www.canon.co.uk/csl/rw.htm` .
RenderWare FTP: `ftp://ftp.canon.co.uk`
Some RenderWare Demos: `ftp://ftp.cs.tu-berlin.de`
A demo of Meme (Multitasking Extensible Messaging Environment) that uses the RenderWare API is available via the Immersive Systems home page at `http://www.immersive.com/`.

4.5 QuickDraw3D

QuickDraw 3D [App] is a cross-platform (Windows, Macintosh) API from Apple for creating and rendering real-time, workstation-class 3D graphics. It consists of human interface guidelines and toolkit, a high-level modeling toolkit, a shading and rendering architecture, a cross-platform file format (3DMF meta file format, with a provided parser) and a device and acceleration manager for plug and play hardware acceleration which allows developers access to third-party 3D acceleration boards.

Two examples for applications using QuickDraw3D are Virtus' *Walkthrough Pro* and the Microsoft Internet Explorer for Macintosh, supporting VRML format.

4.6 Direct3D

Direct3D [Mic] is a complete set of real-time 3-D graphics services that delivers fast software-based rendering of the full 3-D rendering pipeline (transformations, lighting, and rasterization), transparent access to hardware acceleration, and a comprehensive 3-D solution for mainstream PCs under Windows 95. A Windows NT version is expected to be out in the second half of 1996. Direct3D will be included in future versions of the Windows 95 and Windows NT Workstation operating systems. API services include integrated high-level retained-mode and low-level immediate-mode APIs and support for other APIs to sit on top of Direct3D to access 3-D hardware acceleration. Direct3D is fully scalable, enabling

all or part of the 3-D rendering pipeline including geometry transformations, lighting and rasterization to be accelerated by hardware. Direct3D provides access to advanced graphics capabilities of 3-D hardware accelerators including z-buffering, anti-aliasing, alpha blending, mip mapping, atmospheric effects, and perspective correct texture mapping. Tight integration with DirectX and ActiveX technologies allows Direct3D to deliver next-generation 3-D graphics capabilities including video mapping, hardware 3-D rendering in 2-D overlay planes and even sprites providing seamless use of 2-D, 3-D graphics, and digital video, in interactive titles.

4.7 Cosmo GL

Cosmo GL [Sila] brings the full range of OpenGL capabilities and fast graphics performance for applications ranging from games to complex CAD modeling systems on PCs without dedicated 3D graphics hardware.

Cosmo GL inherits the same programming structure developed for OpenGL. Its documented features are all fully implemented and the resulting applications are designed to run exactly the same on all platforms.

4.8 Cosmo 3D

Cosmo 3D [Sila] is a platform-independent graphics toolkit that brings high-performance, real-time, 3D applications to the Internet and the desktop. Implemented in C++, this toolkit supports over 30 leading graphics file formats, including VRML 2.0 and is accessible to Java applications through efficient Java bindings.

Cosmo 3D has been designed to be independent of its underlying rendering architecture, thus enabling maximum portability across all hardware platforms. Developed concurrently on both mainstream personal computers (PCs), as well as high-end professional workstations, Cosmo 3D provides extensive scalability. To achieve maximum performance on mainstream PCs, Cosmo 3D leverages the advanced features of Cosmo GL. Cosmo GL is a highly optimized, fully compliant version of OpenGL for Pentium-based Windows 95 and Windows NT PCs.

Cosmo 3D also provides a native implementation for **Java3D**, the 3D graphics component of the Java virtual machine, by providing an efficient Java binding between a Java application and the natively-compiled Cosmo 3D libraries. Java3D is a graphics application programming interface (API) specification currently being defined through a joint effort by Silicon Graphics, Sun Microsystems, Intel,

and Apple Computer (for more details see Section 1.4.2). Cosmo 3D audio and graphics features:

- Rendering of 3D polygons, lines, points, text, and sound
- Texturing, lighting, transformations, and transparency
- Retained graphics objects that can be rendered individually or as part of a scene graph
- A scene graph that maintains geometry, graphics state, sound, and a transformation hierarchy
- Immediate mode management of graphics state
- Rendering optimizations including level-of-detail (LOD) and culling
- Synchronization of animations, video textures, and audio
- Linear algebra and geometric math primitives
- Morphing and transformation engines for sophisticated character animation
- Intersection and pick testing
- Engines that can drive and be driven by object attributes ("fields")
- Reads and writes VRML 2.0 files

Chapter 5

VRweb

This chapter about VRweb is excerpted from the VRweb home page [And] and a publication by Michael Pichler et al. [POA+95], that appeared in Proceedings of the First Annual Symposium on the Virtual Reality Modelling Language (VRML 95), San Diego, California, December 1995. VRweb, the VRML viewer I have worked with, has been developed at the IICM, Graz University of Technology[1], together with NCSA[2] and the University of Minnesota[3] under the terms of the GNU General Public Licence. It draws on several years of experience with the Harmony 3D Scene Viewer [AP94] for Hyper-G [AKM95, Mau96], which became the base of VRweb. VRweb is designed to work with multiple information systems, namely WWW, Hyper-G and Gopher, as well as on a variety of platforms. It is currently available for most common UNIX platforms (SGI IRIX, Sun Solaris, Sun OS, DEC Alpha, DEC ULTRIX, HP-UX, IBM AIX, FreeBSD and LINUX) as well as Windows (NT, 3.1, 95). Unlike other VRML viewers available in source code, VRweb does not require additional commercial libraries like OpenInventor or Motif, it is based entirely on freely available software components.

VRweb supports five rendering modes: wire frame, hidden line, flat shading, smooth shading, and texturing; the rendering mode specified in the VRML file can be overridden.

VRweb can either be used as a standalone viewer to view local VRML files or in conjunction with a WWW or Hyper-G client to view VRML files on the Net. Depending on the particular configuration, VRweb either functions as a passive

[1] Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology, A-8010 Graz, Austria.

[2] National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana-Champaign.

[3] Distributed Computer Services, University of Minnesota.

helper application unable to resolve external URLs[4] or as a partner application connecting back to the particular browser to resolve URLs as the need arises (for example, when a link anchor is clicked).

The basic software architecture of VRweb is based on multiple layers. The WWW, Gopher, or Hyper-G client communicates via an interface layer with VRweb. The VRML data stream is parsed and an internal data structure constructed. Commands for inline scenes and anchor activations are initiated by the link management module. The rendering component manages user input and visualises the data structure (visibility algorithms). Graphics output is done via an abstract interface (GE3D) to one of several underlying graphics libraries. For parsing the VRML input stream, VRweb utilises a parser derived from the freely available QvLib parser by Paul Strauss of Silicon Graphics. The parser has object-oriented structure and is implemented in C++.

5.1 The VRweb User Interface

The VRweb window is divided vertically into four areas: menu bar, tool bar, display area, and status bar. The menu bar provides access to the full functionality of VRweb, the tool bar and additional accelerator keys provide quick access to commonly used functions.

VRweb supports five rendering modes: wire frame, hidden line, flat shading, smooth shading, and texturing; the rendering mode specified in the VRML file can be overridden. A separate rendering mode may be specified for use during interactive navigation. For example, on a display without hardware graphics acceleration where frame rates are slow for textured models, it might be advisable to navigate in wire frame mode and see a textured version only in the pauses between interaction.

Navigation in 3D space is complicated by the need to control (at least) six degrees of freedom at the same time. VRweb does not assume the availability of any special 3D input device, such as a spaceball or position/orientation sensors, and provides a number of natural mappings for a standard 2D mouse. There are basically two classes of viewpoint control metaphor, depending on whether users wish to move the model itself or to move themselves through the model. Models of *objects* are typically moved and examined, models of *scenes* are typically navigated through (walking, flying, etc.). To cope with these different needs, VRweb provides five navigation metaphors.

Flip mode is used to examine an object, whilst the viewer remains stationary.

[4]Uniform Resource Locator, includes the protocol and the server address for retrieving a file

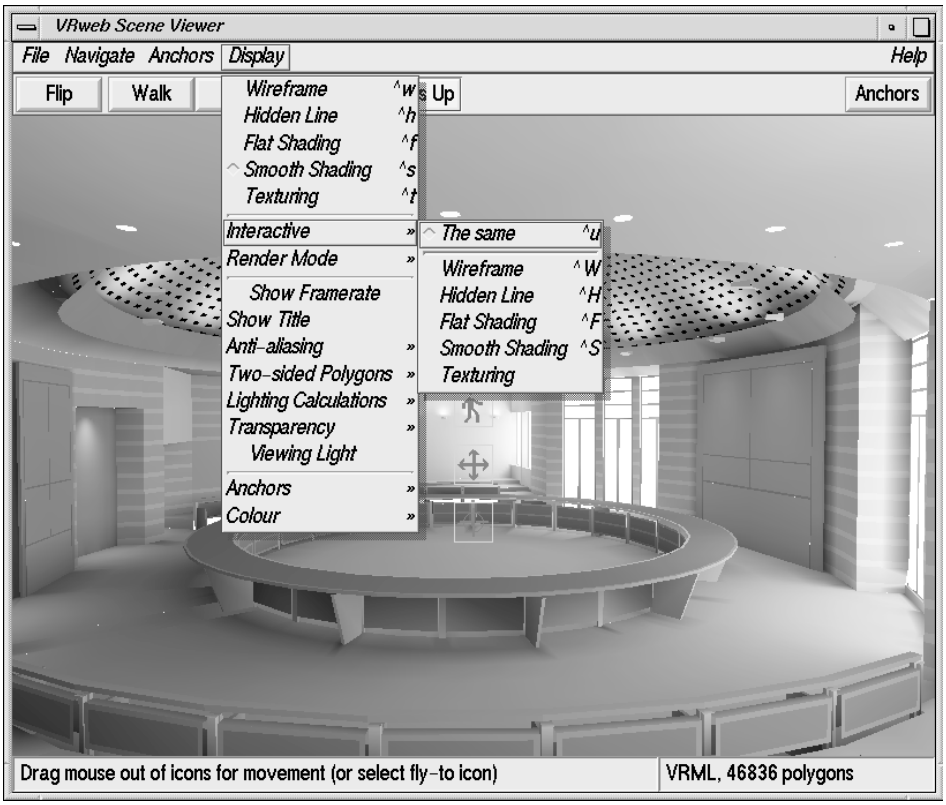

Figure 5.1: The VRweb VRML viewer displaying Lightscape's model of Jerusalem City Hall. Icons are overlaid atop the viewing area for looking, walking, vertical/sideways motion, and point-of-interest navigation. The rendering mode may be set separately for faster interactive navigation.

The mouse buttons have assignments for translation, rotation, and zooming (the object).

The *Walk* metaphor is used to stroll through a 3D environment. Natural walking motion (forwards and backwards, possibly veering slightly to left or right) is assigned to the left mouse button. Complementary controls for vertical motion and side-stepping, and for turning the head are assigned to the middle and right mouse buttons respectively.

Fly is like piloting an aircraft. Flight direction is controlled by the position of the mouse cursor relative to the mid-point of the viewing window (denoted by a cross-hair). The left mouse button activates flight, the other two mouse buttons control acceleration and deceleration. The current direction and speed are indicated in an overlay atop the display area.

Fly To mode implements point-of-interest (POI) style navigation [MCR90]. Here, the user first selects a point of interest somewhere in the model and is then able to perform controlled, logarithmic motion towards (and away from) the POI, approaching by the same fractional distance in each time step. Optionally, a rotational component can be activated (with the Shift key), which results in a final approach path to the POI head-on along the surface normal. This mode is very useful for examining details of a scene and complements other navigation metaphors like Walk or Flip, but is not sufficient as a navigation technique per se.

Heads Up is perhaps the easiest navigation mode for beginners, since its controls are clearly visible. Icons overlaid across the centre of the viewing window (like a pilot's heads-up display) symbolise the individual navigation tools: eyes to look around, a walking person for walking, crossed arrows for vertical and sideways motion, and a crosshair symbol to activate point-of-interest motion.

Should users ever get lost, a "Reset View" function is available to restore the initial view of the scene. A "Level View" function makes the current view horizontal. Other functions include opening local files, saving files, setting preferences, changing colours, etc. The current frame rate can be displayed in the status area. The Hyper-G version of VRweb also supports interactive, point-and-click link creation.

5.2 VRweb as a Multi-System VRML Viewer

VRweb can be used both as a standalone viewer to view local VRML files (e.g. before putting them onto a server) or in conjunction with a WWW or Hyper-G client to view VRML files on the Net. Depending on the particular configuration,

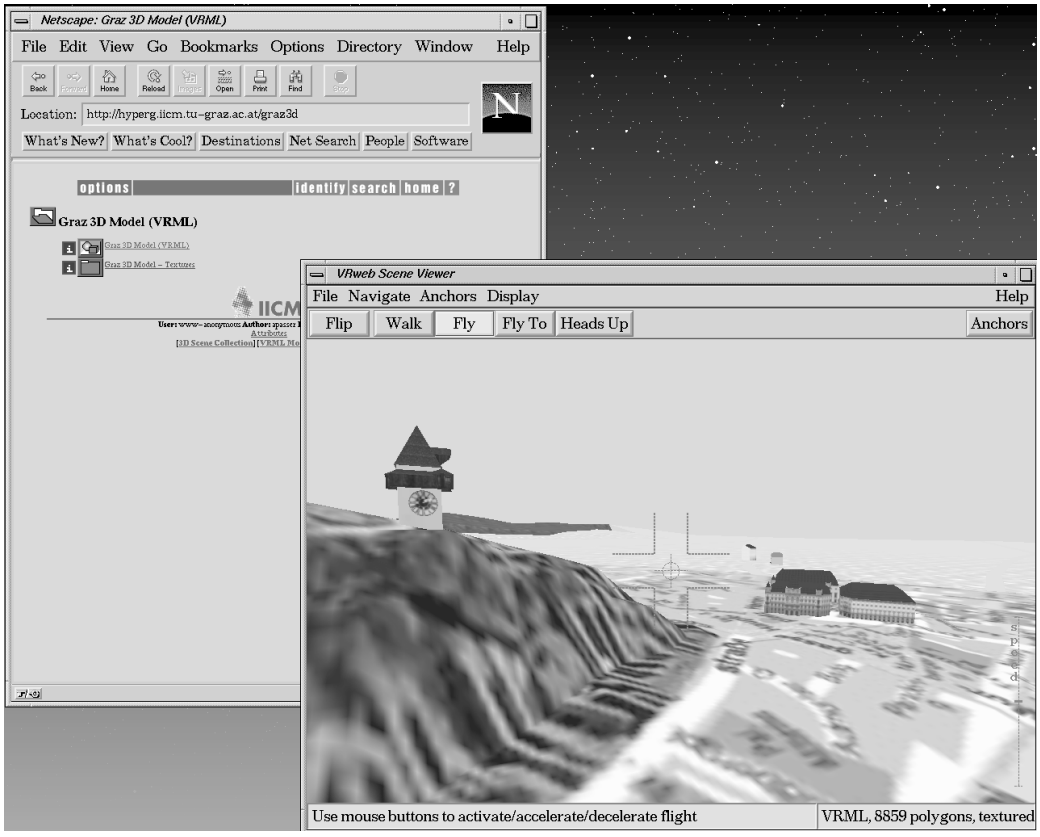

Figure 5.2: VRweb as a helper application with Netscape. The user is navigating through a virtual model of the city of Graz.

VRweb either functions as a passive helper application unable to resolve external URLs or as a partner application connecting back to the particular browser to resolve URLs as the need arises (for example, when a link anchor is clicked). A bidirectional connection to the client is necessary to retrieve anchor destinations that are not VRML models (WWWAnchor nodes). Since VRweb itself speaks HTTP, inlined scenes (WWWInline nodes) and textures are retrieved directly or via a proxy host. A bidirectional connection is useful for general browsing functionality, such as opening a document of arbitrary type, going back and forward in the history, retrieving help files, and setting hotlist entries.

VRweb and WWW

Cooperating with Netscape, inlined scenes and textures are fetched directly by VRweb, since it talks HTTP. But anchor requests are still handled over the remote controlling interface in Netscape, which was implemented in VRweb for a bidirectional communication.

VRweb can also use NCSA Mosaic [And93, Nat] to resolve WWWInline and WWWAnchor requests. The Common Client Interface (CCI) allows external applications such as VRweb to communicate with running sessions of NCSA Mosaic via TCP/IP (or OLE). VRweb uses Mosaic to resolve requests for VRML scenes from a HTTP server, and to manage the display of non-VRML documents which might be referenced in a WWWAnchor node.

VRweb and Hyper-G

Harmony is the Hyper-G authoring tool for Unix/X11 [AK94, MA95]. A number of navigational tools are provided, including a hierarchical overview (collection browser), a bidirectional link overview (local map), and a 3D information landscape. Harmony supports the whole range of Hyper-G features, including link creation in any document type. VRweb's communication with Harmony goes through the Harmony Session Manager, which holds a connection to a Hyper-G server. The Session Manager retrieves metadata and hyperlinks directly, documents themselves are retrieved along a direct connection between the document viewer (VRweb in the case of VRML scenes) and the Hyper-G server. All documents, whether they come from a Hyper-G server or through another protocol like WWW or Gopher, go via the Hyper-G server, which allows for a single document-viewer protocol and organisation-wide caching of documents. Figure 5.3 shows Harmony and VRweb with a virtual tour through the Austrian National Library.

Amadeus is the Windows authoring tool for Hyper-G. Besides viewing all

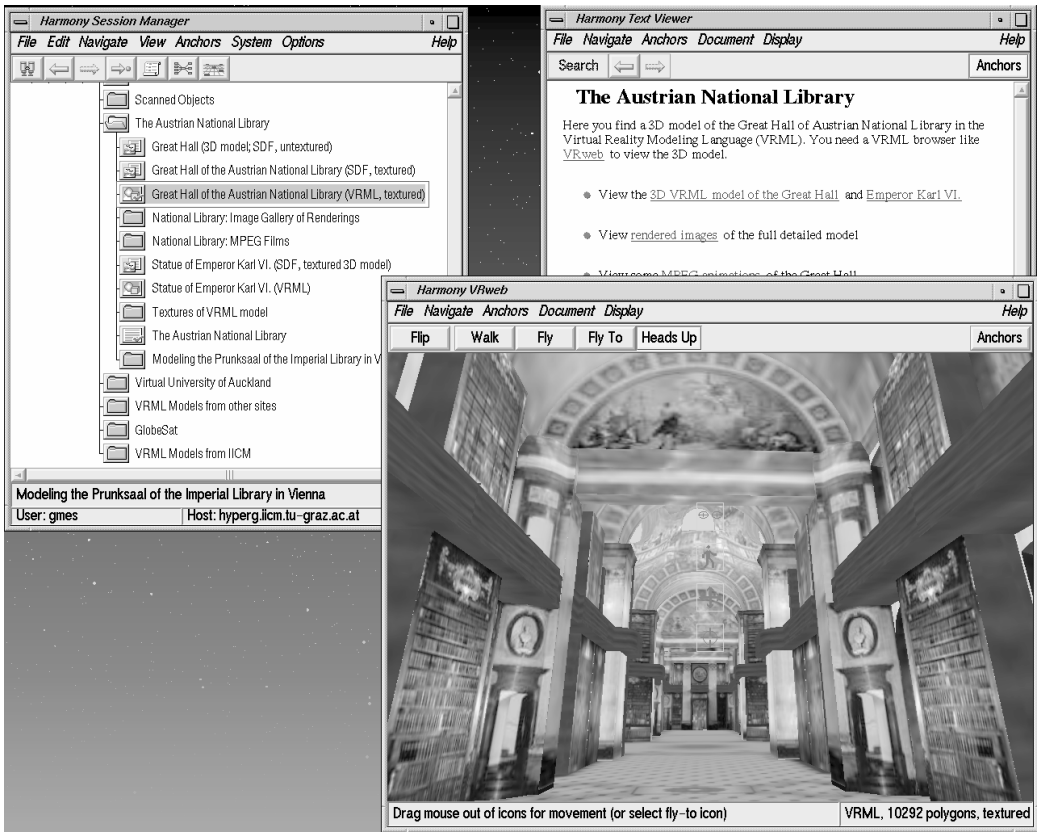

Figure 5.3: VRweb and the Harmony authoring tool for Hyper-G. The user is navigating through a virtual model of the Austrian National Library. In the background are the Harmony Session Manager and Text Viewer.

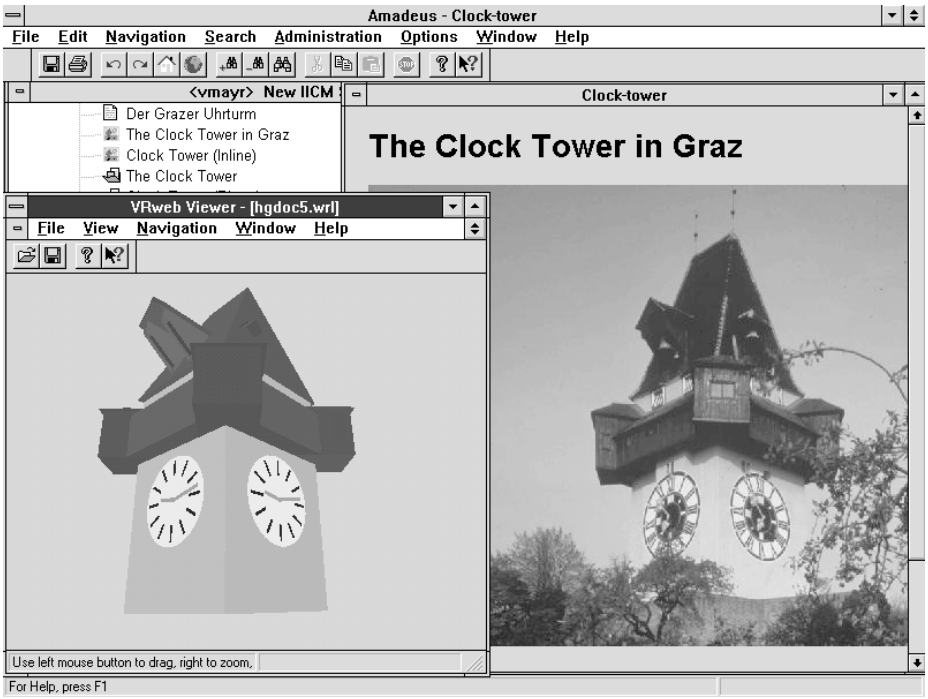

Figure 5.4: VRweb, the Image Viewer, and the Amadeus authoring tool for Hyper-G showing the clocktower of Graz.

types of Hyper-G media it provides easy to use authoring facilities using standard Windows applications and file formats. In the Windows environment all communication with the server is done by Amadeus, the Hyper-G authoring tools for Windows. It provides a link list, holding all anchors for this document, which is generated at document selection time. Inline scenes are handled by Amadeus using asynchronous requests. Link creation is done the same way as for other media types, e.g. text, images, PostScript. Figure 5.4 shows Amadeus and the VRweb for Windows.

Even though links in documents on a Hyper-G server are stored in a link database separate from the documents themselves, link anchors contained within VRML files residing on other servers are respected by VRweb, so as to retain compatibility with the rest of the Web. In the other direction, VRML files on a Hyper-G server have their links merged in on-the-fly, when being accessed from a WWW client.

5.3 VRweb's Software Architecture

Figure 5.5 shows the basic software architecture of VRweb. The WWW, Gopher, or Hyper-G client communicates via an interface layer with VRweb. The arrows indicate the flow of data and control signals.

To keep VRweb source code independent from any particular graphics library, an abstract interface layer called GE3D (Graphics Engine for 3D) is placed between the rendering code and the graphics library. GE3D provides functionality at a slightly higher level of abstraction than, say, OpenGL [NDW93] (a single GE3D call typically resulting in several lower-level OpenGL calls). For example, GE3D has single functions for geometric transformations and drawing polyhedra (a set of faces with related properties). For more details see Section 4.3.

VRweb for Unix/X11

The Unix version of VRweb evolved from the Harmony 3D Scene Viewer for Hyper-G. The design follows object-oriented principles, and is implemented in C++. The base class is the Scene which provides abstract methods for loading, drawing, picking, link management, and selection. Data management, including parsing and traversal to implement the functions, is handled by a class dependent on the scene file format.

For VRML scenes, the data structure built by the browser is kept and has to be traversed for drawing. As the structure of VRML is very much tailored

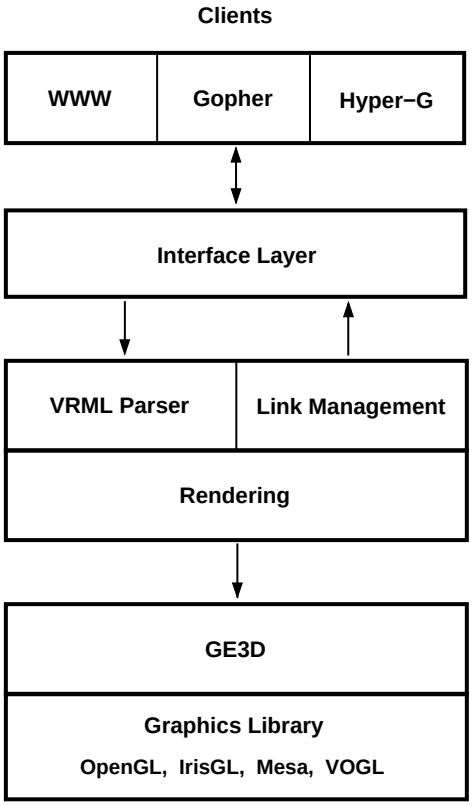

Figure 5.5: VRweb's software architecture, drawn by Michael Pichler.

to the capabilities of OpenGL, the mapping of VRML primitives to the drawing commands of GE3D is straightforward. Using the Inventor library for this purpose would have been easier, but would also have limited the availability of the viewer to platforms on which Inventor is available. The approach taken allows for the optimisation and tuning of the graphics library to the scope of VRML compared to the more general-purpose Inventor library.

Atop this layer, one class is responsible for user interaction. For the X11 version of VRweb, the InterViews [LVC89] toolkit and a library of widgets built on top of it are used to provide menus and other user interface elements. All events, like keystrokes, mouse clicks and movements, are handled and result in function activation or are mapped onto elementary navigational functions provided by the Camera class, depending on the current navigation metaphor. This modular approach allows for a relatively simple port to another window system or platform, as was done for MS-Windows with Visual C++.

The interface to WWW, Gopher, and Hyper-G clients is encapsulated in a separate module. Although they vary in their functionality and capabilities, the different clients are accessed via a common abstract interface.

VRweb for Windows

The design goal of the Microsoft Windows version of VRweb was to have a single source tree and one executable for each platform (Windows 3.x, Windows 95, Windows NT on Intel, Alpha, PowerPC, and MIPS). Therefore, VRweb for Windows is based on Microsoft's WIN32s 32-bit call interface. Using WIN32s has a number of other advantages over the 16-bit Windows programming interface, such as its flat memory model, higher stability, and better device performance.

To render the three dimensional scenes OpenGL is used at a low level and the GE3D library at a higher level, thus staying compatible with the UNIX versions of VRweb. Depending on the system capabilities, either original Microsoft OpenGL (for Windows NT and Windows 95, possibly with hardware acceleration) and/or the public domain library Mesa (software-only rendering) may be used. The Mesa library was enhanced by an additional rendering mode for the Windows driver using RGB colours and a 256 colour palette with indexed addressing. This mode uses the WING library which was designed especially for fast bitmap display.

The architecture of VRweb for Windows is based on the Windows Multiple Document Interface (MDI) and the Microsoft Foundation Classes (MFC). This makes integration into other environments very easy. The VRweb viewer is fully integrated into Amadeus, the Hyper-G authoring tool for Windows. It will also interface with Netscape and Mosaic. A stand-alone version is available using the

same code but not providing any link or inline facilities. Hooks for these functions are provided in the source code, so anyone can implement them for a proprietary target platform. The user interface is similar to that of VRweb for X11, but has Windows look and feel.

5.4 VRweb, Java, and VRML 2.0

It does not seem that it makes sense to support VRML 2.0 scripting in VRweb in its current C++ form, though the static nodes of VRML 2.0 might be supported, depending on the progress of the VRweb development and the performance that can be got out of Java.

To support the scripting facilities in VRML 2.0, it seems to be better to support Java as the scripting language and incrementally re-write part or all of VRweb in Java:

- The GUI, since that is the least portable part of the C++ version.
- In parallel, a VRML 2.0 parser is being written in Java (from scratch).
- The rest of the program logic will be moved to Java.
- Once Java's 3D API is available, the 3D output will be moved as well.

5.5 Features

VRweb is a full featured VRML 1.0 browser supporting amongst others the following features:

- AsciiText/FontStyle.
- Proxy support (for inlines and textures).
- Interactive link creation in conjunction with Hyper-G.
- Automatic normal vector generation.
- Triangulation of non-convex faces.
- Collision detection.
- Grouping, Separators, Coordinates, Normals, Materials, Cameras, Light sources, Transformations, primitive shapes, IndexedFaceSet, IndexedLineSet, PointSet, MaterialBinding, ShapeHints (except concave faces), LOD (level of detail), and Texturing.
- Optional switching to a lower display quality during motion.

- Inline data are gunzipped automatically, redirects are handled, relative URLs are handled properly when the parent URL is given on commandline. The transfer can be interrupted at any time.

5.6 Availability

VRweb is available both in binary and in source code for a variety of platforms. The source code is copyrighted, but is freely available for non-commercial use (see the copyright notice with the distribution for full details). VRweb code is available by anonymous ftp from:

```
ftp://ftp.iicm.tu-graz.ac.at/pub/Hyper-G/VRweb
ftp://ftp.ncsa.uiuc.edu/Hyper-G/VRweb
```

as well as from a number of other mirror sites. There is a VRweb home page with up-to-date technical information at:

```
http://www.iicm.edu/vrweb
```

The VRweb mailing list is the place for questions, suggestions, and discussion of VRweb. To subscribe to the mailing list, send email to:

```
listproc@iicm.tu-graz.ac.at
```

with message body (no subject needed):

```
subscribe vrweb FirstName LastName
```

To unsubscribe, mail "`unsubscribe vrweb`" to the same address. To send mail to all members of the list, simply compose your message or question and send it to:

```
vrweb@iicm.tu-graz.ac.at
```

Chapter 6

Visibility in VRweb

The purpose of this thesis is to investigate how to speed up rendering in VRweb for standard workstations or PCs without graphics hardware support. This should be achieved through calculating the visible and the hidden parts of a scene for a particular camera position (one frame). We will further see how a certain degree of frame coherence can be used so that the calculation does not need to be done for every frame — without noticeable distortion, at least in certain navigation modes.

6.1 The Visibility Strategy

Let's start the considerations with three assumptions:

1. Disabling the Z-buffer results in higher rendering speed because of less calculation time.

2. Rendering time is proportional to the number of polygons the viewing pipeline is fed with.

3. Rendering a polygon takes longer than calculating if it is visible.

When disabling the Z-buffer another rendering technique has to be used to get the correct visibility. The BSP-tree offers a very fast (O(n), n...number of polygons) way to achieve this for static scenes with only the camera position changing. The drawback is an extensive preprocessing phase (only done once or when the scene changes) and an increase of the total number of polygons through splits.

With assumption 1 this should already result in a speed up of the complete rendering process.

Assumptions 2 and 3 lead us to the conclusion that calculating the visible polygons for each frame, based on binary space-partitioning, gives a further increase of the frame rate. These assumptions are discussed further in Chapter 7.

6.2 Programming Environment

The visibility calculation in VRweb was implemented in C++ [Str91] under Linux on a PC Pentium 100 with 16MB RAM. The development was done with a stand alone version of VRweb (release 1.1.2) and compiled with the GNU C++ Compiler version 2.7.0. After having finished the implementation the code was included into the "full" version of VRweb and compiled for all platforms VRweb is ported to with the GNU C++ Compiler v2.6.3.

6.3 Data Flow in VRweb

- The VRML file is parsed in *wrlbuild.C* in a way derived from the freely available parser QvLib by Paul Strauss of Silicon Graphics. The resulting Qv-structure is a tree construct representing the VRML format.

- Each Qv-node has a *draw* method implemented in *wrldraw.C* which performs the rendering in the same order the VRML file was generated. In case of a "material" node the current material setting is changed, in case of a "transformation" node the current transformation matrix is redefined, and so on. If a geometric object node like "indexed face set" executes the *draw* method the viewing pipeline, already initialized with the current material-, light-, and camera settings, is fed with the vertices to be rendered.

  Rendering one frame of the entire scene is done through a draw-method call for the root-node of the Qv-structure, which calls the draw-methods for all its children, and so on.

Implementing a new way of rendering should still preserve the original way, based on Z-buffer visibility calculation. All interface aspects should stay unchanged (eg. picking of objects) — therefore the new rendering method is an additional feature of VRweb without taking any functionality away (see section 6.4).

6.4 New Data Flow in VRweb

The changed rendering strategy necessitates a completely different data flow. Remember, that it is only an additional possibility to use this new method and not a replacing one.

- The Qv-structure is built in the original way.
- Preprocessing. The Qv-structure is parsed like in the original way of drawing the whole scene. A method *buildBSP* does everything needed to build the BSP-tree. Creating the BSP-tree for an existing Qv-structure is accomplished with a buildBSP-call to the root of the Qv-structure.
- Rendering. The created BSP-tree is traversed in *back-to-front* order to achieve the desired correct visibility with disabled Z-buffer. Each node in the BSP-tree holds one or more polygons. A polygon (class Face) owns a draw-method which is actually doing the rendering (a call to GE3D).

  If the visible-face-calculation is activated then the SVBSP-tree algorithm is performed and tags each polygon with *visible* or *hidden*.

6.5 Object's Data Structure

Working at the level of polygons we need a data structure holding faces (a list of vertices) with attributes like the material and the active light sources without storing the same information more than once.

Faces having the same vertices, light sources or material should refer to these vertices, lights or material instead of holding them in their data members.

The introduced data structure is based on indexed face sets (see section 2.3). The advantage of indexing is used when splitting polygons during the creation of the BSP-tree.
Two classes are used to represent this concept: **Face** and **FaceAttr**. The data members of these classes that are relevant at this point:

```
class Face
{
  private:

    int* vert_index_list_;     // list of all vertex indices
    int vertex_num_;           // length of vertex index list
```

```
    int normal_num_;            // 0 or vertex_num_
    vector3D normal_;           // n=(a,b,c); ax+by+cz=d plane equ.
    float d_;                   // value of d
    FaceStatus status_;         // VIS, PARTLY_VISIBLE, NOT_VISIBLE
    FaceAttr* faceattr_;        // face belongs to this faceset
};

class FaceAttr
{
  private:

    QvMFVec3f* vertex_list_;    // list of all vertices
    QvMFVec3f* normal_list_;    // list of normal vectors
    int vertex_num_;            // number of all vertices
    int normal_num_;            // number of all normals
    materialsGE3D* material_;   // one material structure
    AnyLight** light_;          // list of some light structures
    int backface_culling_;      // 0 turned off, 1 activated
    int light_num_;             // number of lights
};
```

A FaceAttr contains the vertices (indexed face set) and provides access to the corresponding material, lighting and the attribute if backface culling should be activated.

A Face holds a pointer to its FaceAttr and the indexlist of integers referring to the position in the vertexlist (in FaceAttr). If a normallist exists for this polygon then the number of normals in Face is set to the same value as the number of vertices.

A normallist stores a normal vector for each vertex of a polygon to allow smooth shading (Phong Shading) which interpolates the normal vectors between the vertices to overcome the "faceted" appearance when approximating a curved surface by a set of polygons. The correct calculation of the normal vectors had to be considered during the creation of new polygons through splitting. The new vertex normal vector results as the linear interpolation between the normal vectors of the two vertices which spanned the original edge (see Figure 6.1). The polygon split influences the output even though the calculation of the vertex normal vectors is performed properly. For more about rendering artefacts see Section 6.9.

Furthermore an instance of FaceAttr only points to the material already put on the heap during parsing the VRML file (see Section 6.3).

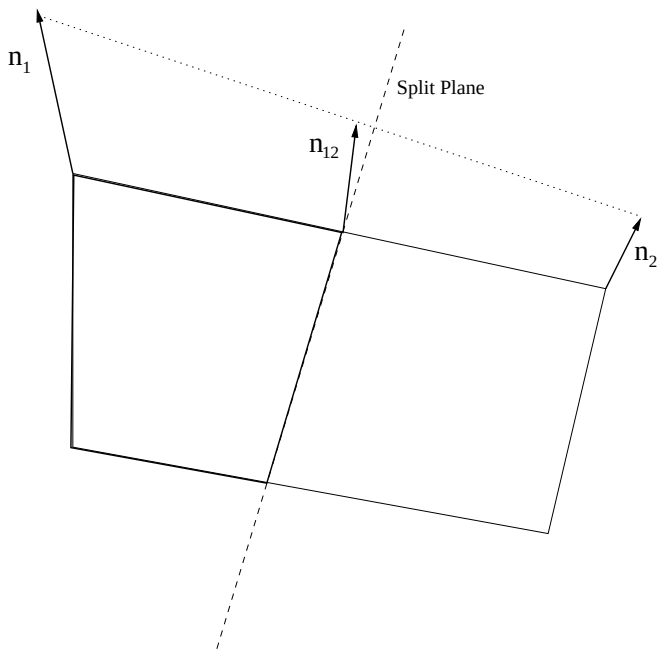

Figure 6.1: Calculation of normal vectors for new vertices.

During the construction of the BSP-tree all used light sources are put on the heap and referred to from the instances of FaceAttr.

6.6 BSP-Tree – Binary Space-Partitioning Tree

As described in Section 3.2.2, the BSP-tree is a data structure holding information about the spatial relation between all polygons in a scene with the drawback of making splitting of polygons necessary and therefore increasing the total number of geometric objects.

6.6.1 Data Structure

The data members of the class BSPTree that are relevant at this point:

```
class BSPTree
{
  private:
```

```
    QvMFFace facelist_; // pointer list to all
                        // faces(on the same plane)
    BSPTree* front_;    // BSP-tree that is completely
                        // infront of the plane, which
                        // is spanned by the facelist
    BSPTree* back_;     // same like front_ but in the back
    BSPTree* parent_;   // a link to the parent node,
                        // nil for the root
};
```

An instance of BSPTree is a node in the tree and holds pointers to its children and to its parent. The scene information is held in a pointerlist referring to one or more polygons lying on the same planar plane. This plane spatially splits the following polygons of the entire scene in two parts, referred to in the children of this instance.

6.6.2 Interface to the BSP-tree

The public methods of the class BSP-tree that are relevant at this point:

```
class BSPTree
{
  public:

    void insert(Face* face);
    void drawBackToFront(const point3D& position, BSPMode mode);
};
```

The BSP-tree is created by *inserting* all polygons of the scene (in arbitrary order) into the root of the BSP-tree (initialized to an empty node). Every node redirects the incoming polygon to its front- or back-tree until it reaches a leaf node. If the incoming polygon does not lie completely in front or in the back of the plane, spanned by the polygons held in the current node, it needs to be split. In this case the original polygon is deleted and the two new polygons are *inserted* into the corresponding child-trees.

To traverse (and draw) the whole BSP-tree in back-to-front order the method *drawBackToFront* needs to be called in the root. The passed camera position determines the resulting order. BSPMode is used to turn visibility calculations on or off, it will be explained in Section 6.7.

6.6.3 Building the BSP-Tree

Schematic code for building the whole BSP-tree. All faces are being inserted into the root of the BSP-tree, assuming that at least the root, already holding one polygon, exists.

```
void BSPTree::insert(Face* face)
{
  // clip generates new faces if the face was split,
  // otherwise the pointer to the original face is returned
  // if the face lies on the same plane both front_face
  // and back_face are nil

  // plane_equation comes from the first polygon in the facelist_.

  Clipping::clip(*face, plane_equation,
                 front_face, back_face, split_type);

  if ((split_type == ONLY_FRONT) || (split_type == SPLIT))
  {
    if (front_) front_->insert(front_face);
    else
    {
      front_ = new BSPTree(front_face);
      front_->setParent(this);
    }
  }
  if ((split_type == ONLY_BACK) || (split_type == SPLIT))
  {
    if (back_) back_->insert(back_face);
    else
    {
      back_ = new BSPTree(back_face);
      back_->setParent(this);
   }
  }
  if (split_type == SPLIT)
  {
    delete face;
    return;
```

```
  }
  if (split_type == SAME_PLANE)
  {
    connect(face);
    return;
  }
}
```

"Connect" simple adds the passed face to the facelist of the BSP-node.
"Clip" performs the Sutherland Hodgman clipping algorithm, explained in Foley et al. [FvDFH90]. This algorithm is somewhat more general than the Cohen-Sutherland algorithm. Not only convex but also concave polygons can be clipped against a plane. Taking the result of one clipping against a plane it can be easily used to clip against the next plane. Taking all planes, bounding a convex 3D polyhedral volume this algorithm can be easily used to clip against 3D objects.

All clipping related functions are implemented in the file clipping.C — Clipping is not a class in object oriented terms but a collection of related functions.

6.6.4 Traversing the BSP-Tree

Schematic code for traversing the whole BSP-tree in back-to-front order, position is the current camera position:

```
void BSPTree::drawBackToFront(const point3D& position, BSPMode mode)
{
  // plane_equation comes from the first polygon in the facelist_.

  if (Clipping::inFront(position, plane_equation))
  {
    if (back_) back_->drawBackToFront(position, mode);
    draw(position,mode);
    if (front_) front_->drawBackToFront(position, mode);
  }
  else
  {
    if (front_) front_->drawBackToFront(position, mode);
    draw(position,mode);
    if (back_) back_->drawBackToFront(position, mode);
  }
}
```

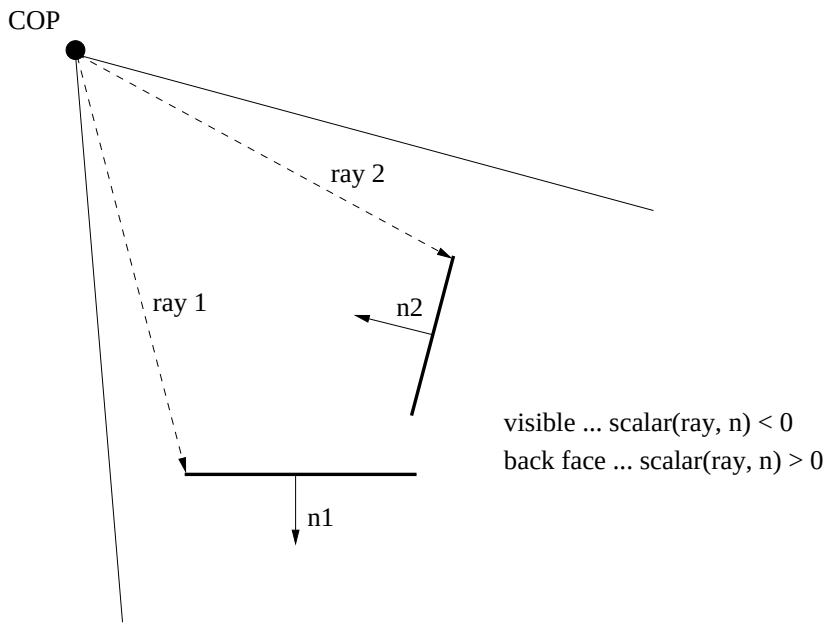

Figure 6.2: Backface Culling.

"draw(position,mode)" sends a draw method to all faces held in the node.

6.6.5 Backface Culling

If backface culling is enabled then the following code is used to determine if a polygon is facing the camera (see Figure /refbackfaceculling.fig):

```
any_vertex ... any vertex of the face
position ... camera position

sub3D(any_vertex, position, n);
scalar = dot3D(n, normal_);

if (scalar < 0) forceDraw();
```

6.7 The Shadow Volume BSP-Tree

As described in Section 3.3.4 the SVBSP-tree is a data structure to determine all visible parts of a scene. To get correct results the visibility must be determined

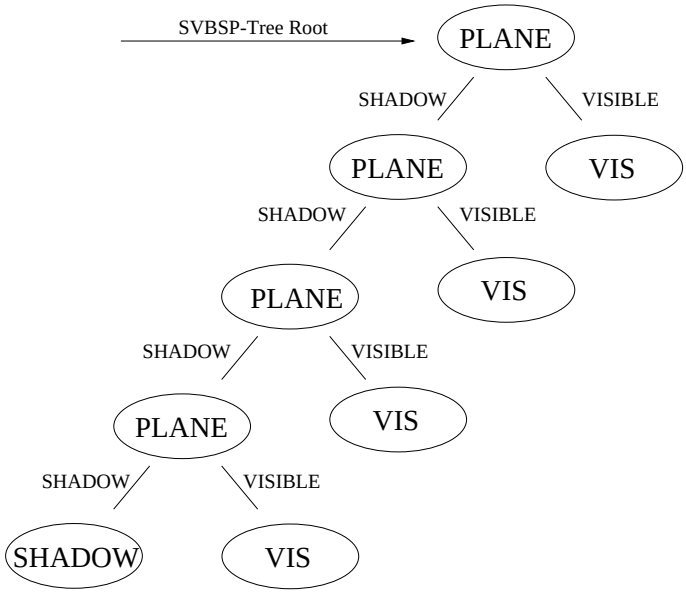

Figure 6.3: The first four sided face in the SVBSP-tree.

for every frame.
I try to give a description of this algorithm without delving too deep into technical details:

- Traverse the BSP-tree in front-to-back order, take the next polygon, do the backface culling.

- Check the visibility of the current polygon against all polygons that lie in front of it — the visibility information of polygons that lie in front is held in the SVBSP-tree. If the current polygon is at least partly visible then it extends the shadow volume (stored in the SVBSP-tree).

How does it actually work?

The SVBSP-tree does not hold vertices but planes that are spanned by the COP (camera position) and one edge of a visible polygon. To represent all effects of a visible four sided polygon, four plane-entries are needed in the SVBSP-tree. See Figure 6.3 how the SVBSP-tree looks like after inserting the first polygon into an empty SVBSP-tree.

Determining Visibility

Lets have a look what happens if another face is checked for visibility using the existing SVBSP-tree in Figure 6.3. Every node in the SVBSP-tree can be in three modes — holding either a plane-equation or the status *visible* or *shadow*. A node *visible* or *shadow* must be a leaf node and a leaf node is not allowed to hold a plane equation.

To determine visibility the current face is filtered through the plane-equations held in the SVBSP-tree until it or its parts (in case of splits) reaches a leaf node. If this reached leaf node has the status *shadow* then this arrived (part of the) face is dropped and not considered any more. If a *visible* node is reached then this part of the face extends the SVBSP-tree, described in Section 6.7.3.

6.7.1 Data Structure

The data members of the class SVBSPTree that are relevant at this point:

```
class SVBSPTree
{
  private:
    Plane plane_;        // shadow plane;
    NodeType type_;      // VISIBLE, SHADOW, PLANE
    SVBSPTree* visible_; // visible child
    SVBSPTree* shadow_;  // child in shadow
};
```

6.7.2 Interface to the SVBSP-Tree

The public methods of the class SVBSP-tree that are relevant at this point:

```
class SVBSPTree
{
  public:
    void replace(Face* face, Face* original_face);
    void filter(Face* face, Face* original_face);
}
```

Every time a face or a part of a face reaches a "visible-node" then this visible node in the SVBSP-tree data structure is *replaced* by a construct like in figure 6.3 representing the visible part of the *filtered* polygon. We need to keep the original polygon for two reasons:

- To tag the original polygon and not the probably split and later deleted ones.
- To determine if all parts of the original face are in shadow or if there is at least one visible part of the face.

6.7.3 Building the SVBSP-Tree

Schematic code for building the whole SVBSP-tree. All faces are *filtered* through the root of the SVBSP-tree. The SVBSP-root is initialized to a *visible* node. All visible parts of faces are inserted into the SVBSP-tree by *replacing* the corresponding *visible* node.

```
void SVBSPTree::filter(Face* face, Face* original_face)
{
  if (type_ == VISIBLE)
  {
     replace(face, original_face);
     return;
  }

  Clipping::clip(*face, plane_equation,
                 visible_face, shadowed_face, split_type);

  if ((split_type == ONLY_FRONT) || (split_type == SPLIT))
     visible_->filter(visible_face, original_face);

  if ( ((split_type == ONLY_BACK) || (split_type == SPLIT))
       && (shadow_->type_ == PLANE) )
     shadow_->filter(shadowed_face, original_face);
}
```

6.7.4 Closer Examination of the SVBSP-Tree

Building the SVBSP-tree is done during a traversal of the BSP-tree in front to back order. The status of a face is initialized with "visible". If at least a part of the face is visible its status is set to "partly visible", if a part of the face reaches a "shadow-node" nothing is changed. If the filtering of one face terminates and it is not tagged as "partly visible" then it is completely in shadow because its status has not been changed. The split parts of a face are only needed

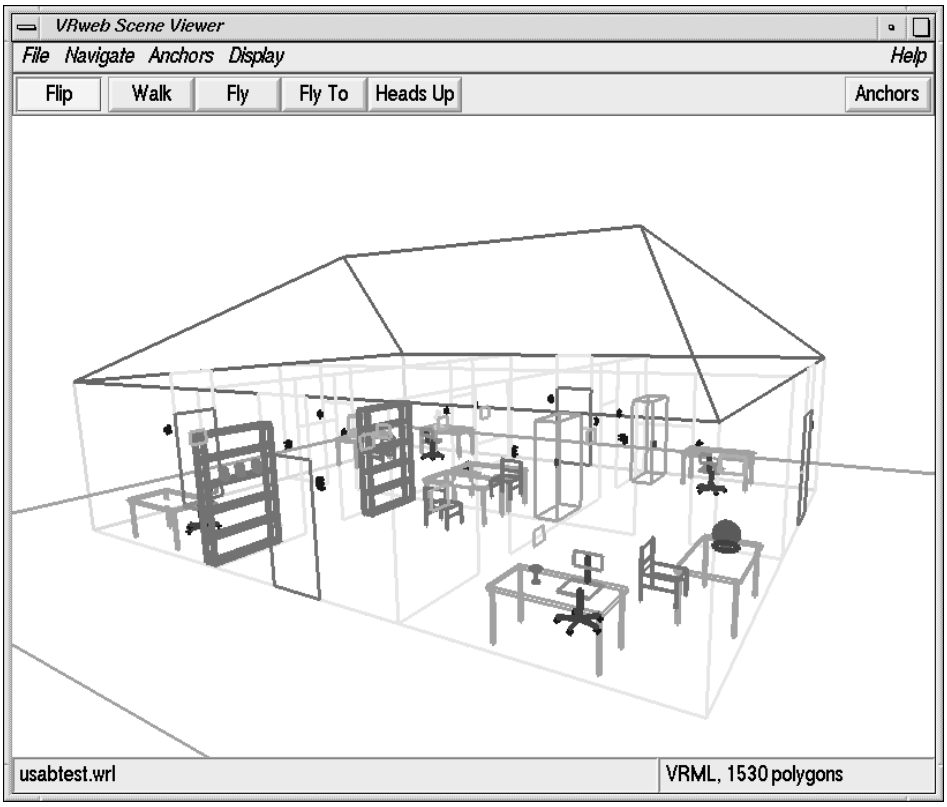

Figure 6.4: The model of a building in wire frame mode – without visibility calculation. Compare with Figure 6.5 .

to extend the shadow volumes, the tag is attached to the original face. If a face is "partly visible" then it is going to be rendered (in a normal back-to-front BSP-tree traversal).

```
face_ptr->setStatus(VIS);
svbsp_root->filter(face_ptr, face_ptr);
if (face_ptr->status() == VIS)
{
  face_ptr->setStatus(NOT_VISIBLE);
  svbsp_root->hidden_face_count_++;
}
```

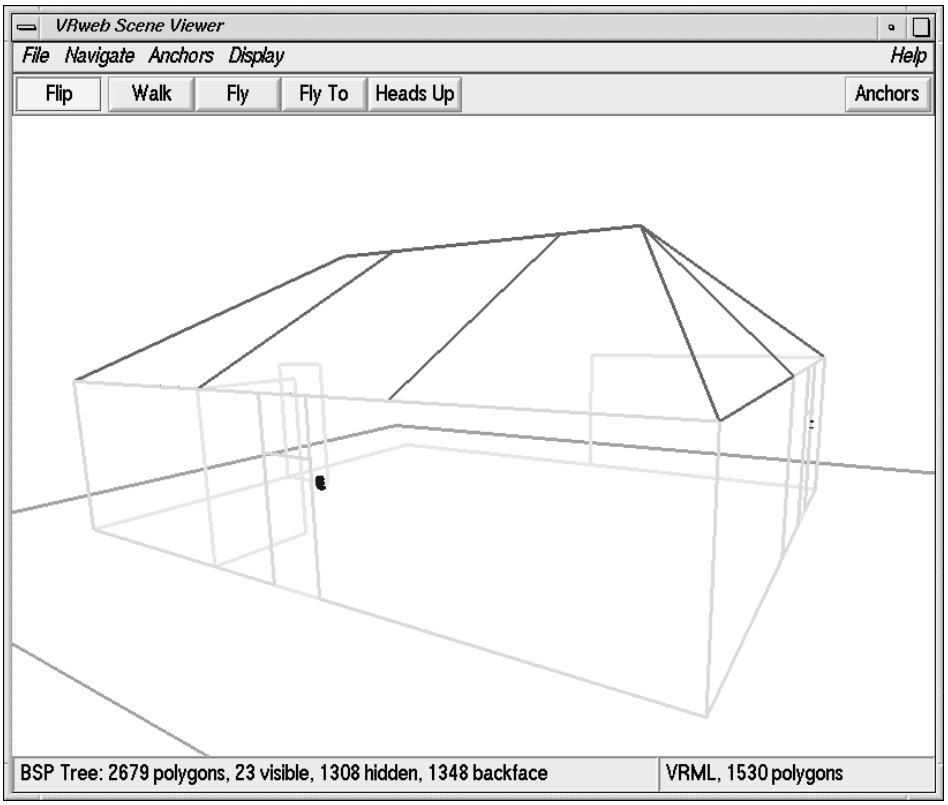

Figure 6.5: The visible parts of the same model as in Figure 6.4 determined through visibility calculation with the SVBSP-tree. Think about the effect the two doors of the building have on the visibility. The same frame is shown in smooth shading mode in Figure 7.1 .

6.7.5 Tuning the Visibility Calculation

During the creation of the SVBSP-tree the incoming faces must be filtered until they reach leaf nodes. The time consumption for building a SVBSP-tree grows rapidly with each new visible part that extends the tree.

Because of the fast growth of the SVBSP-tree with the number of visible faces, we need to find ways to let the SVBSP-tree terminate at a sensible degree of execution. Two possibilities are already implemented:

- Only allow large polygons to extend the SVBSP-tree.
- Only use the first n polygons (from front to back).

The problems that arrise are that the parameters for breaking up the algorithm are highly scene-dependent.

Another feature was implemented to use **frame coherence** during rendering. Because we know exactly which faces are visible and which are not after having calculated the SVBSP-tree for the whole scene, we can use this information and only render the visible polygons for the next x frames. After this number of frames the visibility needs to be recalculated so that the distortions do not have an effect on the user's behaviour.

6.8 The Viewing Volume

To keep the path through the SVBSP-tree short for unimportant faces, the bounding planes of the viewing volume, consisting of the front-, and backclipping plane and the four planes that are spanned by the COP and the four vertices of the viewplane window, are the first ones that get inserted into the SVBSP-tree. Therefore all faces that lie completely outside of the viewing volume can be rejected and marked as shadowed very fast. Therefore the SVBSP-tree is always initialized in a way showed in Figure 6.6.

```
void SVBSPTree::insertViewVolume(Camera* camera)
{
  camera->getViewVolume(view_volume);

  SVBSPTree* clipping_planes = this;
  for (int i=0; i < 6; i++)
    {
       type_ = PLANE;
```

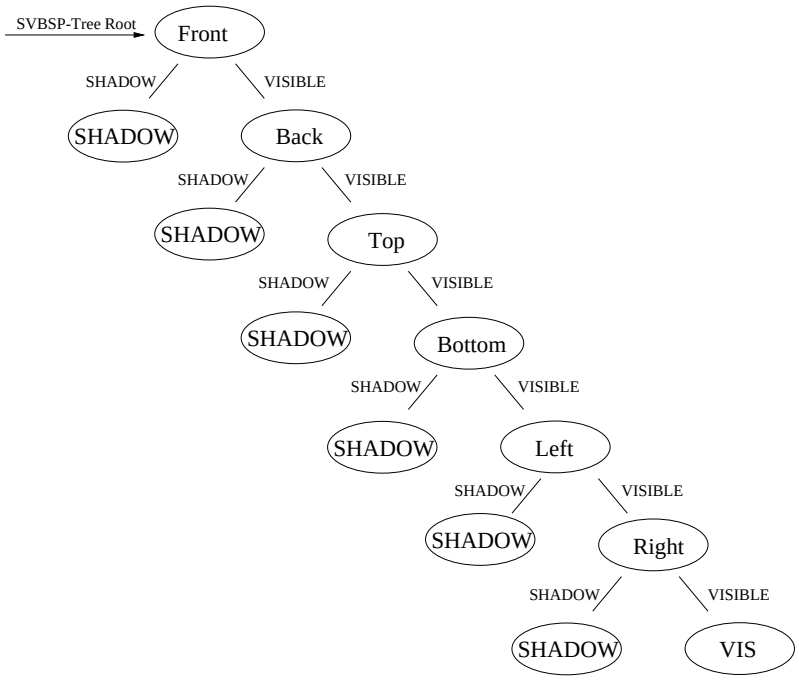

Figure 6.6: The viewing volume in the SVBSP-tree.

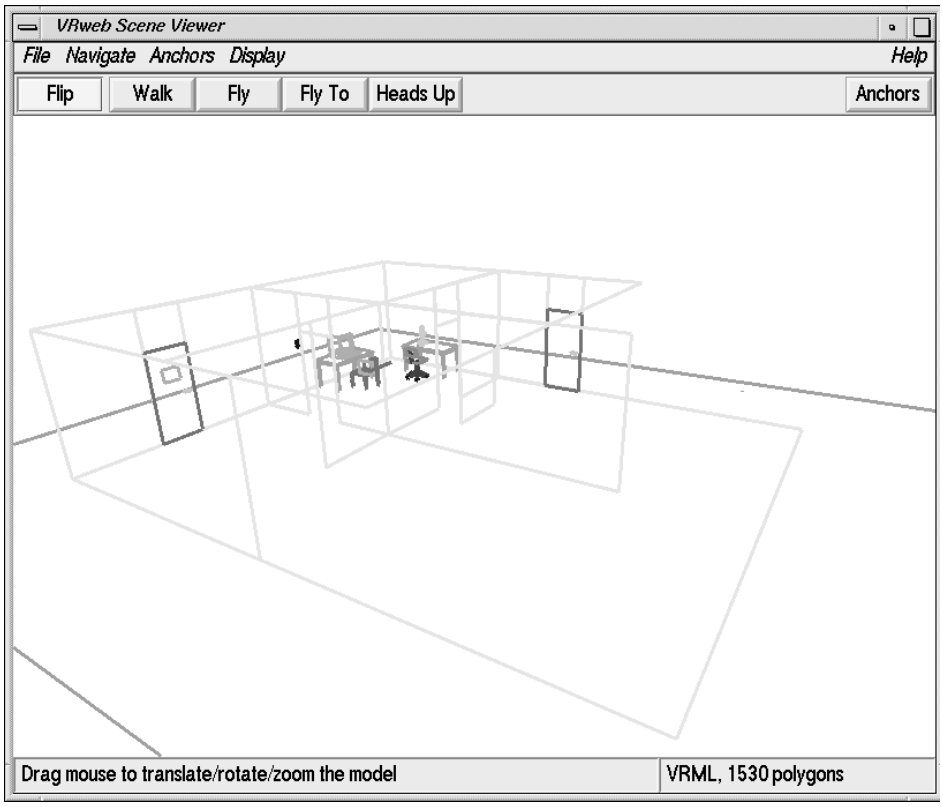

Figure 6.7: The same building as in Figures 6.4 and 6.5 – but the user being in the room in the back looking to the two outer walls of the building. This figure shows the visible faces after visibility calculation with the SVBSP-tree. Compare with Figure 6.8 where the scene is clipped with the viewing volume.

```
        clipping_planes->setPlaneEqu(view_volume[i]);
        clipping_planes->shadow_ = new SVBSPTree(SHADOW);
        clipping_planes->visible_ = new SVBSPTree(VISIBLE);
        clipping_planes = clipping_planes->visible_;
    }
}
```

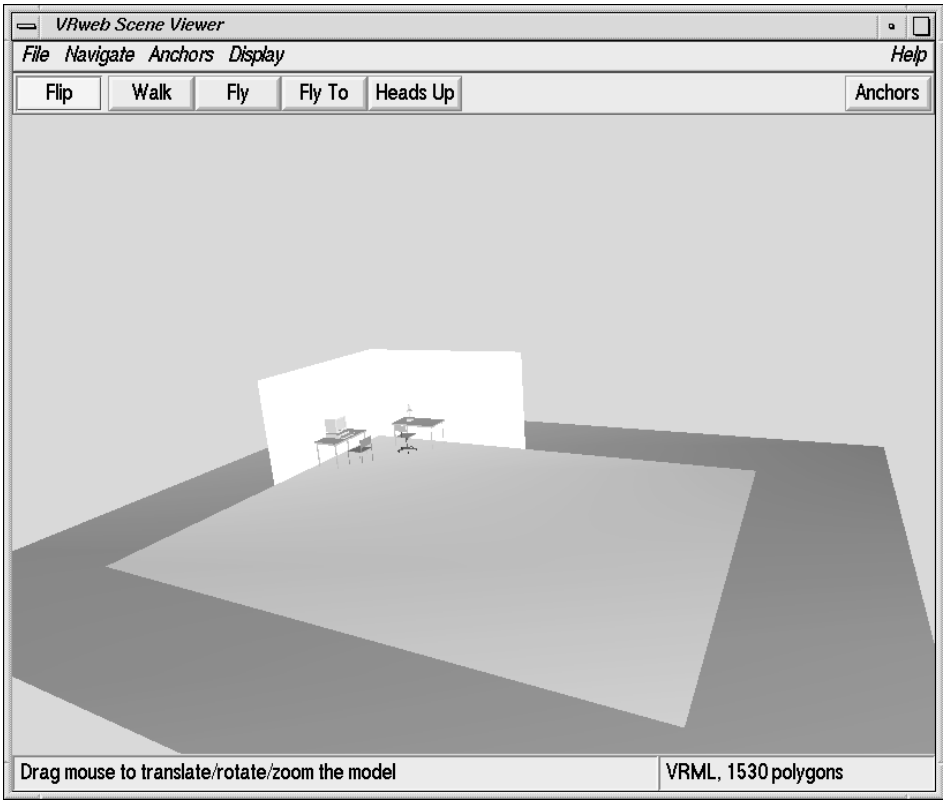

Figure 6.8: The same frame as in Figure 6.7 This Figure shows the visible faces after visibility calculation with the SVBSP-tree and clipping with the viewing volume.

6.9 Artefacts Through Splitting

In two cases one can notice artefacts:

- Flat shading. Is a polygon split and two are rendered instead of the original one then the intensity of the split polygons might be different. Flat shading applies an illumination model that determines a single intensity value once. Values may be calculated for the centre of the polygon, or for the polygon's first vertex. Because the newly created polygons have different centres or might have different first vertices, different colour intensities might be applied. This effect can be best noticed with large polygons.

- Smooth shading. Shading discontinuities can occur with interpolated shading when two adjacent polygons fail to share a vertex that lies along their common edge. Smooth shading (Phong shading) is interpolating the vertex normal vectors to determine the colour intensity for each pixel of the polygon. Due to the splitting while creating the BSP-tree the case that two polygons share an edge but do not share all vertices might occur (see Figure 6.9).

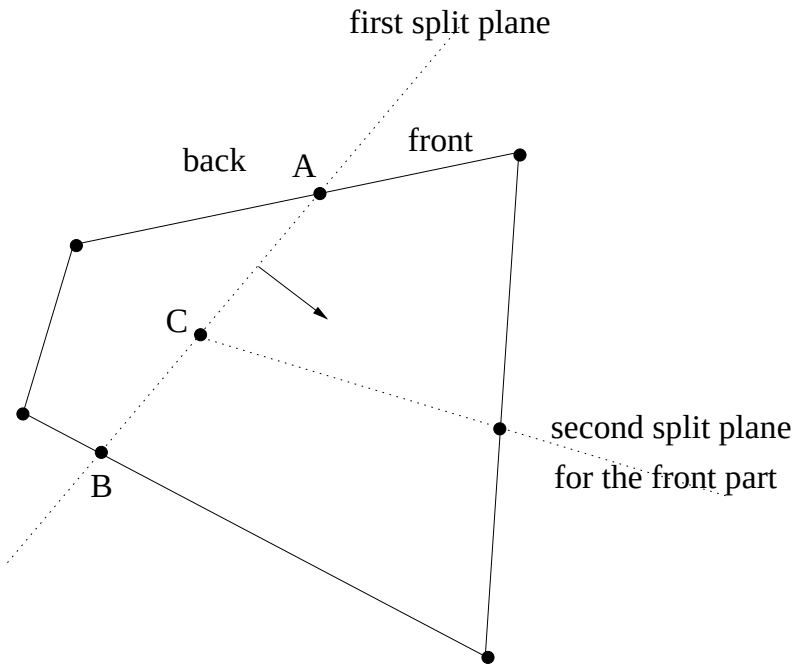

Figure 6.9: Vertex C is shared by the two polygons on the right, but not by the polygon on the left.

Chapter 7

Efficiency Analysis

7.1 The Different Rendering Modes

In this section the standard rendering method using Z-buffering in compared to the new way built on space partitioning (BSP-tree), with and without backface-culling, view-volume clipping, and visibility calculation through the SVBSP-tree. Three different types of 3D scenes are used:

- A model of a house with a number of rooms (without priori knowledge of the layout) — *usabtest.wrl*.
- A model of a room (Keith's office) — *office.wrl*.
- A model of a chemical molecule — *molecule.wrl*.

I used four test environments for the efficiency analysis:

- *usabtest.wrl* from outside (Figure 7.1)
- Inside a room of *usabtest.wrl* (Figure 7.2)
- A complete view of *office.wrl* (Figure 7.3)
- A complete view of *molecule.wrl* (Figure 7.4)

The rendering modes are the following:

- The original way (Z-buffer) – ORIGINAL.
- BSP-tree without backface culling – BSP NO CULL.
- BSP-tree with backface culling – BSP CULL.
- SVBSP-tree without viewing volume clipping – SVBSP NO CLIP.
- SVBSP-tree with viewing volume clipping – SVBSP CLIP.
- Intermediate SVBSP-tree without recalculation of the visibility – INTERMEDIATE.

The following values were measured:

- Number of polygons, the viewing pipeline is fed with – except the case with view clipping, where the number is taken before clipping.
- Number of rejected (hidden) polygons.
- Time to render one frame on a PC Pentium 100, 16 MB RAM, under Linux without graphics acceleration (Mesa) in three rendering modes: smooth shading, flat shading and wire frame.
- Time to render one frame on a SGI Onyx, IRIX 5.3, in the same three rendering modes without (Mesa) and with graphics acceleration (OpenGL).

Rendering times are resolution, colour depth, and window-size dependent. The measured times were taken under same conditions on both the PC and the SGI Onyx, except that on the PC only 16 bit (instead of 24 bit on the SGI Onyx) colour depth was used.

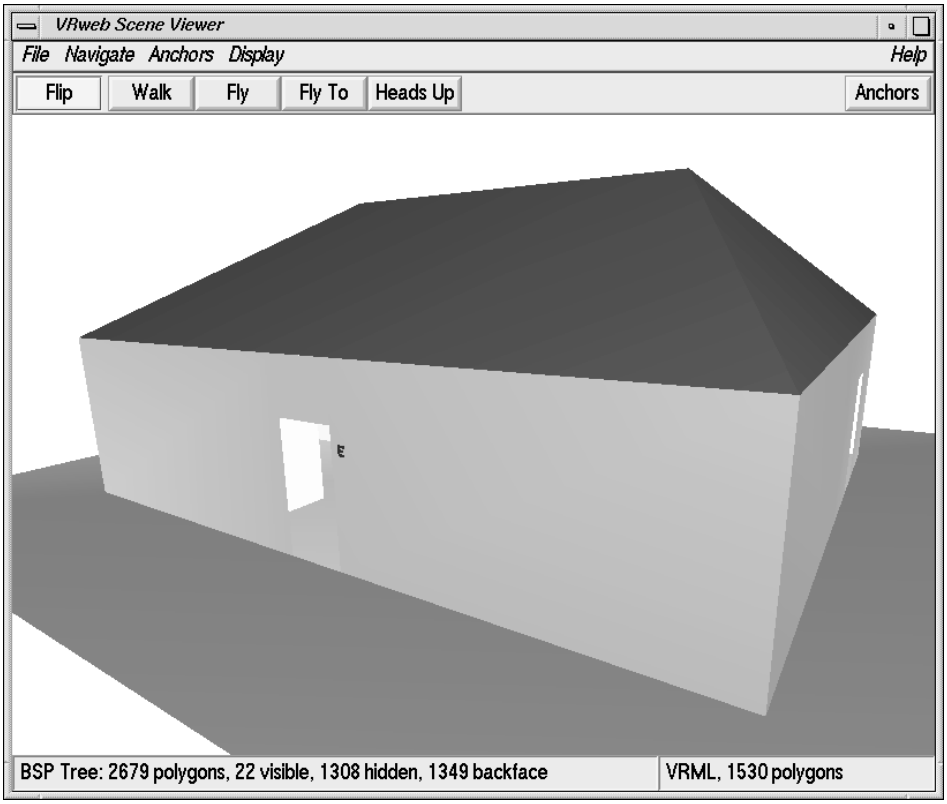

Figure 7.1: Complete view of the same building as in Figure 6.4 .

USABTEST.WRL	ORIGINAL	BSP NO CULL	BSP CULL
visible polygons	1530	2679	1330
hidden polygons	0	0	1349
PC wire Mesa	0.13 s	0.25 s	0.16 s
PC flat Mesa	0.72 s	1.70 s	0.90 s
PC smooth Mesa	1.08 s	2.44 s	1.28 s
Onyx wire Mesa	0.13 s	0.22 s	0.14 s
Onyx flat Mesa	0.98 s	2.46 s	1.43 s
Onyx smooth Mesa	1.24 s	2.76 s	1.57 s
Onyx wire OGL	0.02 s	0.04 s	0.04 s
Onyx flat OGL	0.04 s	0.20 s	0.11 s
Onyx smooth OGL	0.04 s	0.22 s	0.12 s

Table 7.1: Rendering Times of the Frame in Figure 7.1 – Z-Buffer, BSP-Tree.

USABTEST.WRL	SVBSP NO CLIP	SVBSP CLIP	INTERMEDIATE
visible polygons	79	79	79
hidden polygons	2600	2600	2600
PC wire Mesa	0.29 s	0.38 s	0.06 s
PC flat Mesa	0.64 s	0.69 s	0.35 s
PC smooth Mesa	0.89 s	0.96 s	0.60 s
Onyx wire Mesa	0.38 s	0.51 s	0.09 s
Onyx flat Mesa	0.48 s	0.55 s	0.19 s
Onyx smooth Mesa	0.55 s	0.76 s	0.26 s
Onyx wire OGL	0.22 s	0.32 s	0.02 s
Onyx flat OGL	0.22 s	0.32 s	0.02 s
Onyx smooth OGL	0.22 s	0.32 s	0.02 s

Table 7.2: Rendering Times of the Frame in Figure 7.1 – SVBSP-Tree.

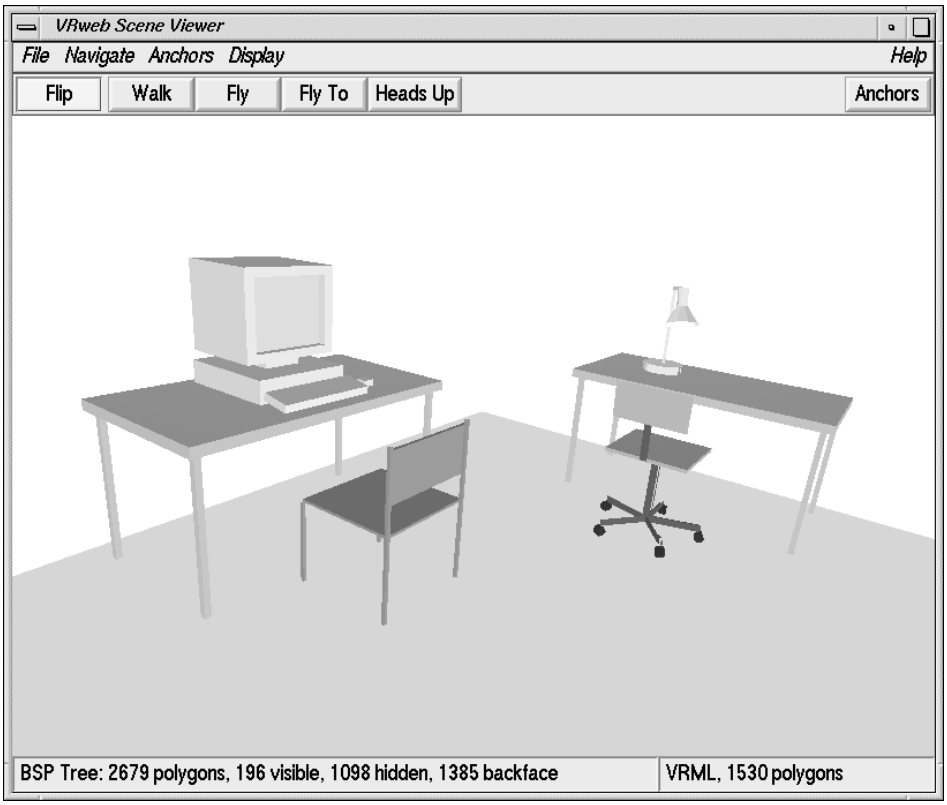

Figure 7.2: Inside the room in the back of the building – see Figure 6.8.

USABTEST.WRL	ORIGINAL	BSP NO CULL	BSP CULL
visible polygons	1530	2679	1330
hidden polygons	0	0	1349
PC wire Mesa	0.12 s	0.20 s	0.13 s
PC flat Mesa	0.66 s	1.45 s	0.77 s
PC smooth Mesa	0.95 s	2.00 s	1.06 s
Onyx wire Mesa	0.10 s	0.21 s	0.15 s
Onyx flat Mesa	0.84 s	2.41 s	1.40 s
Onyx smooth Mesa	1.06 s	2.60 s	1.51 s
Onyx wire OGL	0.01 s	0.03 s	0.03 s
Onyx flat OGL	0.03 s	0.22 s	0.10 s
Onyx smooth OGL	0.03 s	0.22 s	0.10 s

Table 7.3: Rendering Times of the Frame in Figure 7.2 – Z-Buffer, BSP-Tree.

USABTEST.WRL	SVBSP NO CLIP	SVBSP CLIP	INTERMEDIATE
visible polygons	196	196	196
hidden polygons	2483	2483	2483
PC wire Mesa	0.70 s	0.42 s	0.07 s
PC flat Mesa	1.07 s	0.77 s	0.45 s
PC smooth Mesa	1.39 s	1.03 s	0.75 s
Onyx wire Mesa	1.03 s	0.47 s	0.08 s
Onyx flat Mesa	1.50 s	0.75 s	0.32 s
Onyx smooth Mesa	1.59 s	0.81 s	0.40 s
Onyx wire OGL	0.64 s	0.42 s	0.02 s
Onyx flat OGL	0.64 s	0.42 s	0.02 s
Onyx smooth OGL	0.64 s	0.42 s	0.02 s

Table 7.4: Rendering Times of the Frame in Figure 7.2 – SVBSP-Tree.

OFFICE.WRL	ORIGINAL	BSP NO CULL	BSP CULL
visible polygons	727	1338	997
hidden polygons	0	0	341
PC wire Mesa	0.10 s	0.15 s	0.10 s
PC flat Mesa	0.45 s	0.85 s	0.45 s
PC smooth Mesa	0.66 s	1.35 s	0.68 s
Onyx wire Mesa	0.10 s	0.14 s	0.10 s
Onyx flat Mesa	0.54 s	1.01 s	0.59 s
Onyx smooth Mesa	0.68 s	1.27 s	0.67 s
Onyx wire OGL	0.01 s	0.03 s	0.03 s
Onyx flat OGL	0.02 s	0.10 s	0.06 s
Onyx smooth OGL	0.02 s	0.10 s	0.06 s

Table 7.5: Rendering Times of the Frame in Figure 7.3 – Z-Buffer, BSP-Tree.

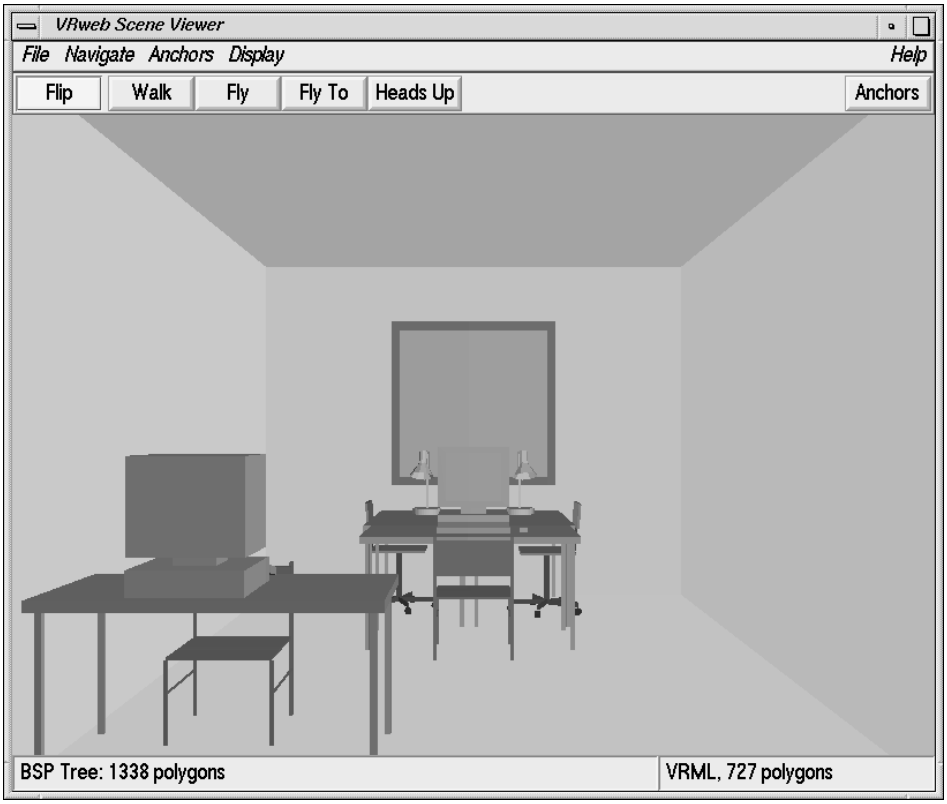

Figure 7.3: A complete view of Keith's office.

OFFICE.WRL	SVBSP NO CLIP	SVBSP CLIP	INTERMEDIATE
visible polygons	315	315	315
hidden polygons	1023	1023	1023
PC wire Mesa	0.64 s	0.66 s	0.08 s
PC flat Mesa	0.90 s	0.93 s	0.35 s
PC smooth Mesa	1.14 s	1.18 s	0.57 s
Onyx wire Mesa	0.77 s	0.72 s	0.09 s
Onyx flat Mesa	1.03 s	1.15 s	0.41 s
Onyx smooth Mesa	1.13 s	1.17 s	0.45 s
Onyx wire OGL	0.64 s	0.69 s	0.02 s
Onyx flat OGL	0.64 s	0.69 s	0.03 s
Onyx smooth OGL	0.64 s	0.69 s	0.03 s

Table 7.6: Rendering Times of the Frame in Figure 7.3 – SVBSP-Tree.

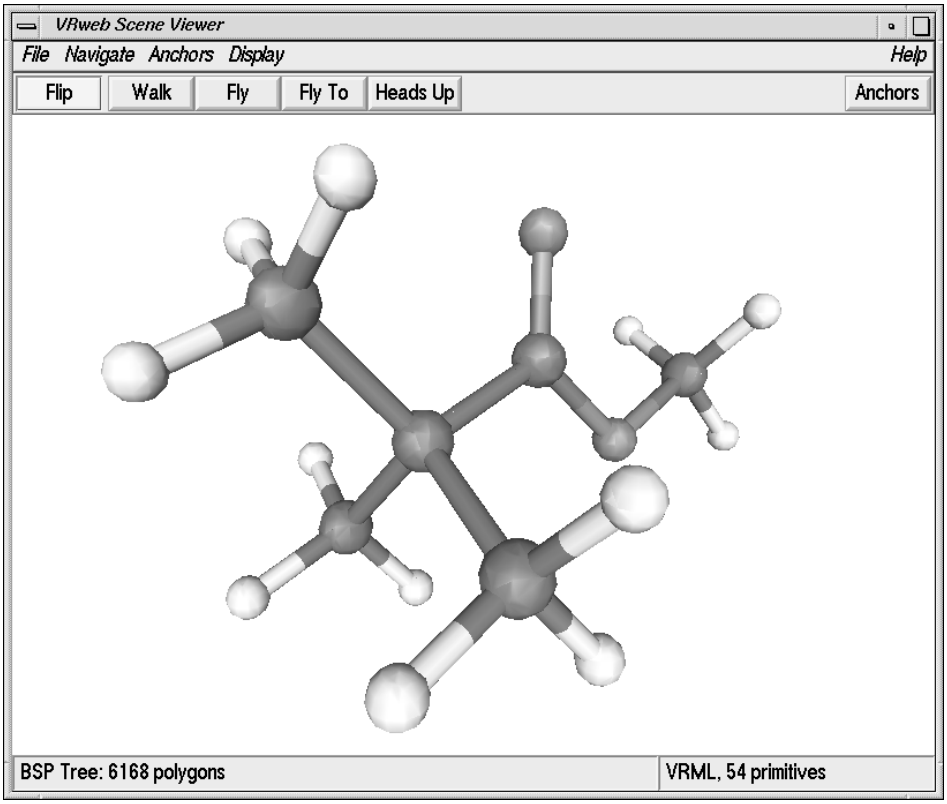

Figure 7.4: A complete view of a chemical molecule.

MOLECULE.WRL	ORIGINAL	BSP NO CULL	BSP CULL
visible polygons	1876	6168	3058
hidden polygons	0	0	3110
PC wire Mesa	0.12 s	0.42 s	0.24 s
PC flat Mesa	0.39 s	1.07 s	0.58 s
PC smooth Mesa	0.53 s	1.42 s	0.72 s
Onyx wire Mesa	0.13 s	0.41 s	0.24 s
Onyx flat Mesa	0.55 s	3.51 s	1.73 s
Onyx smooth Mesa	0.71 s	3.67 s	1.84 s
Onyx wire OGL	0.02 s	0.06 s	0.05 s
Onyx flat OGL	0.04 s	0.30 s	0.15 s
Onyx smooth OGL	0.04 s	0.32 s	0.16 s

Table 7.7: Rendering Times of the Frame in Figure 7.4 – Z-Buffer, BSP-Tree.

MOLECULE.WRL	SVBSP NO CLIP	SVBSP CLIP	INTERMEDIATE
visible polygons	1540	1540	1540
hidden polygons	4528	4528	4528
PC wire Mesa	2.61 s	2.57 s	0.14 s
PC flat Mesa	2.74 s	2.84 s	0.34 s
PC smooth Mesa	2.77 s	2.91 s	0.48 s
Onyx wire Mesa	3.20 s	3.01 s	0.13 s
Onyx flat Mesa	3.91 s	3.76 s	0.88 s
Onyx smooth Mesa	3.81 s	3.76 s	1.03 s
Onyx wire OGL	3.10 s	3.10 s	0.04 s
Onyx flat OGL	3.10 s	3.10 s	0.08 s
Onyx smooth OGL	3.10 s	3.13 s	0.09 s

Table 7.8: Rendering Times of the Frame in Figure 7.4 – SVBSP-Tree.

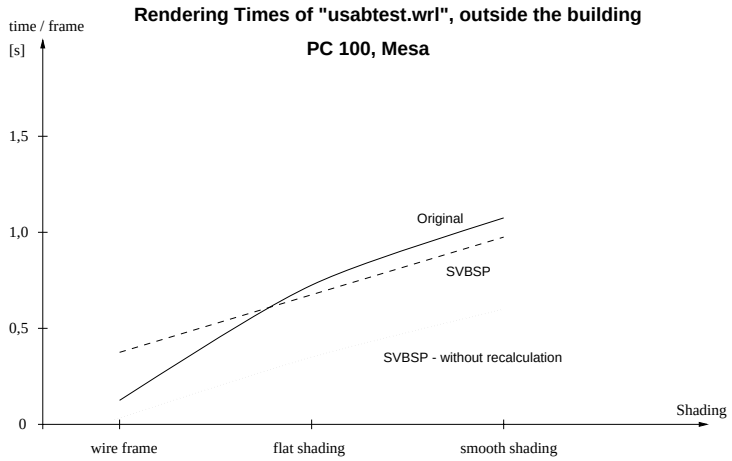

Figure 7.5: Usabtest.wrl outside the building.

Results

Figures 7.5 to 7.8 show the different rendering times of the original rendering method and the SVBSP-tree approach with and without recalculating the visibility every frame for the PC 100. The diagrams do not provide new information but ease the comparison between the original version and the SVBSP-tree method.

One result, that can be easily seen, is that the SVBSP-tree approach is absolutely unsuitable for objects like the molecule. There is no depth complexity and therefore no chance to determine some polygons as hidden to decrease the overall number of polygons. Furthermore the molecule is a good example how big a BSP-tree can get even for a small model, consisting of only 54 primitives or 1876 polygons. Spheres and cylinders that are built of polygons have the characteristic to split everything around them more often if many polygons are used to approximate the primitives. In our example a sphere is built out of 6 slices of 12 segments. This is the reason why the size of the BSP-tree is 6168 polygons instead of 1876 originally.

The generated BSP-tree (without optimization) was in all other cases about twice as big as the original scene, which is an expected result. An optimization of the tree generation would lead to much better results in the case of the use of primitives like spheres, cylinders, or cones, and when used to introduce spatial subdivision. Not in a single case was speeding up achieved through disabling the Z-buffer. Hence perhaps the most important insight from the results of these test is:

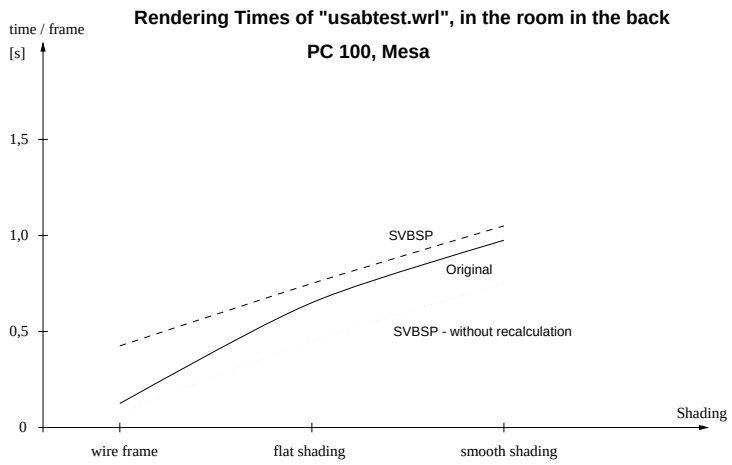

Figure 7.6: Usabtest.wrl inside the room in the back.

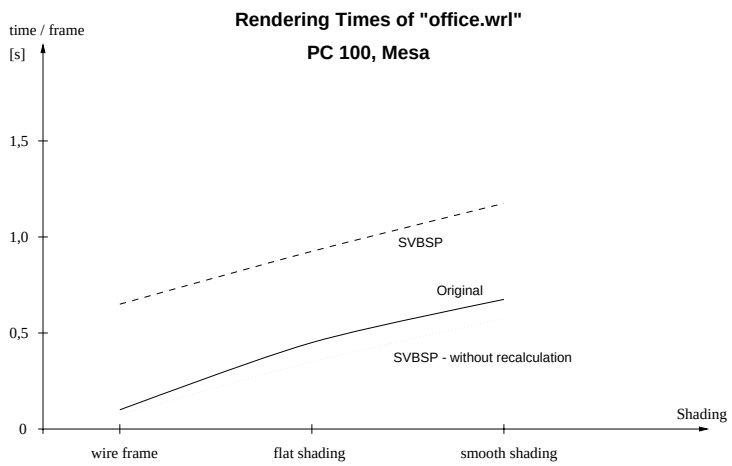

Figure 7.7: Keith's office.wrl.

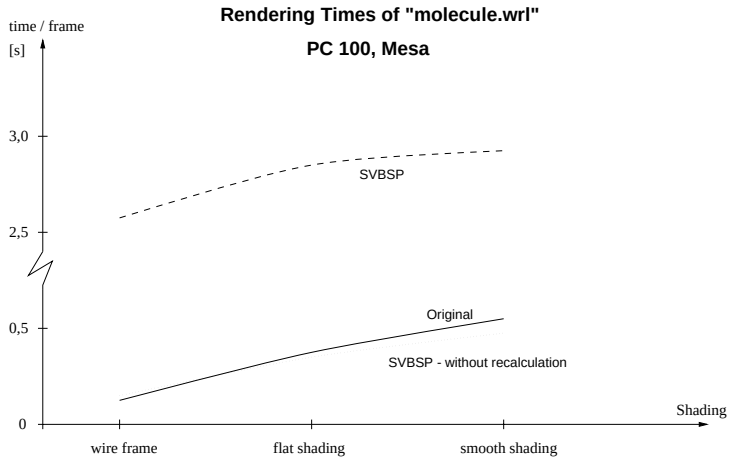

Figure 7.8: The chemical molecule.wrl.

Disabling the Z-buffer does not result in higher rendering speed. The software Z-buffer calculation of the Mesa library is not costly at all.

We can further see that *office.wrl*, a simple model of a room, has not got enough depth complexity to make a visibility calculation useful. A good situation for visibility considerations is standing in front of a wall and the rest of the model is hidden behind this wall. This is not the case if you are in a room and the whole model is only this single room. All this was expected apart from the good software Z-buffer performance. Let us have a look how the visibility algorithms can be used in a building that consists of a couple of rooms with open doors in between – *usabtest.wrl*. In this case the performance of rendering with visibility calculation is better than in the other examples. But the achieved framerates are similar to the framerates in the original version. Only one thing can be used to our advantage. We know which polygons are visible and we know the hidden ones. Remembering frame coherence we can use this information and render the visible polygons for the next n frames without the extensive recalculation for the whole scene. In the *fly to mode*, a movement on a straight line towards a visible destination, we might even render the whole *flight* without recalculation. In the following section I compare this *walkthrough* to a visible destination in the original and the new rendering mode.

7.2 A Walkthrough in Fly-To Mode

In this section a walkthrough from outside the building defined in *usabtest.wrl* – see Figure 7.1 – to the room in the back part – see Figure 7.2 – is compared between the original rendering method, the SVBSP-tree approach recalculating the visibility every frame, and every 10 frames. The walkthrough is performed in the *fly to mode* on a straight line on a PC 100 (see Figure 7.9) and on a SGI Onyx (see Figure 7.10), both rendering using the Mesa library.

USABTEST.WRL	ORIGINAL	SVBSP CLIP	INTERMEDIATE
outside	1.18 s	1.14 s	0.96 s
	1.38 s	1.56 s	0.77 s
	1.43 s	1.40 s	0.85 s
	1.36 s	1.37 s	0.69 s
half way	1.15 s	1.45 s	0.90 s
	1.11 s	1.31 s	0.90 s
	1.08 s	0.97 s	0.60 s
	0.89 s	0.89 s	0.82 s
in the room	0.90 s	0.83 s	0.76 s

Table 7.9: Walkthrough compared on a PC 100, Mesa library.

USABTEST.WRL	ORIGINAL	SVBSP CLIP	INTERMEDIATE
outside	1.04 s	0.62 s	0.79 s
	1.30 s	0.62 s	0.27 s
	1.48 s	0.68 s	0.32 s
	1.38 s	1.06 s	0.28 s
half way	1.44 s	0.98 s	0.38 s
	1.54 s	0.78 s	0.42 s
	1.20 s	0.58 s	0.76 s
	1.02 s	0.51 s	0.35 s
in the room	1.06 s	0.80 s	0.43 s

Table 7.10: Walkthrough compared on a SGI Onyx, Mesa library.

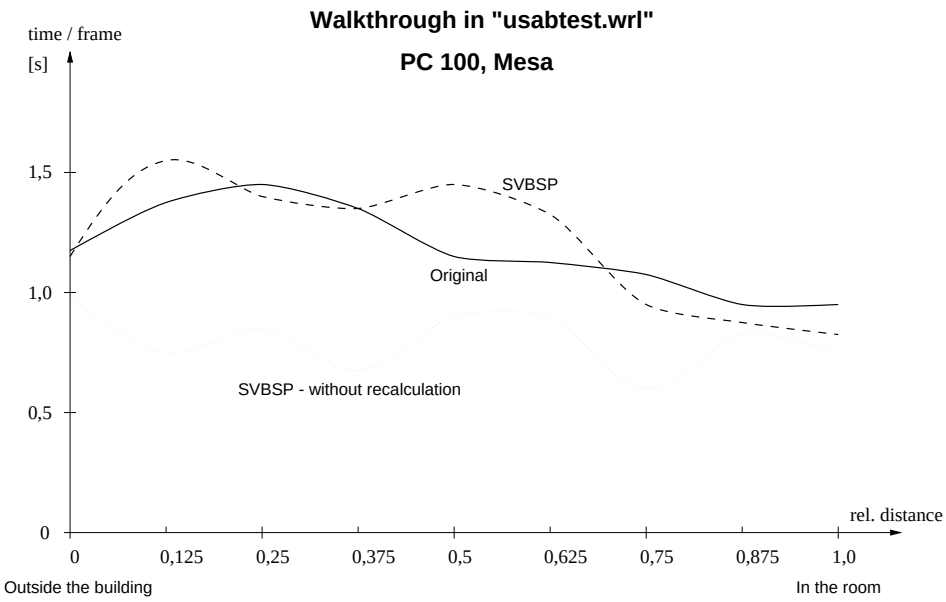

Figure 7.9: Walkthrough compared on a PC 100, Mesa library.

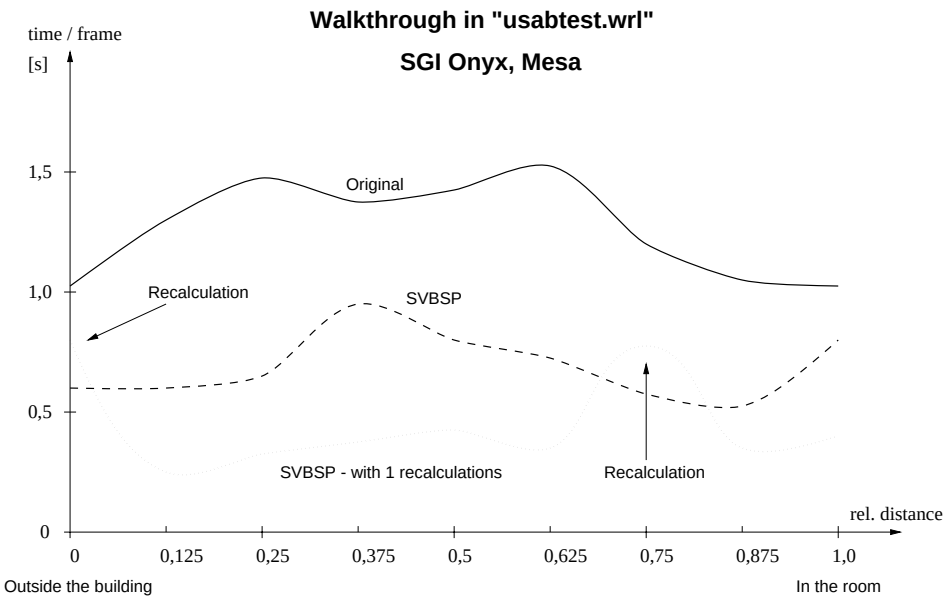

Figure 7.10: Walkthrough compared on a SGI Onyx, Mesa library.

Results

In the case of this *fly to walkthrough* the result with the SVBSP-tree approach using frame coherence is better than the original rendering method. The speeding up of the rendering time is about 30–50% on the PC Mesa version and about 50–75% on the Onyx Mesa version. Walking around in the building defined in *usabtest.wrl*, only 50 to 250 polygons are being rendered compared with 1530 in the original scene. But the visibility calculation is so extensive that the advantage is only small.

7.3 Conclusions

Looking at the three assumptions in turn:

1. Disabling the Z-buffer results in higher rendering speed because of less calculation time.

   The Z-buffer calculation is not costly at all, this assumption was wrong.

2. Rendering time is proportional to the number of polygons the viewing pipeline is fed with.

   That is right, apart from a small offset to initialize the viewing pipeline.

3. Rendering a polygon takes longer than calculating whether it is visible.

   The chosen visibility calculations, although quite simple, turned out to be very expensive and grow rapidly for bigger scenes – and graphics libraries are getting better all the time. The assumptions might be correct for small scenes but not for big scenes. Z-buffer calculation is linear with the number of pixels covered by a polygon, object space visibility calculation is not. How to speed up the visibility algorithm through introducing hierarchical spatial subdivision is explained in Section 8.

Chapter 8

A Framework to Extend VRweb to Achieve Accelerated Walkthroughs of Complex Buildings

8.1 Introduction

Inside a building only a small part of a scene's geometry might be inside the viewing volume. Therefore visibility culling methods are very effective. To compare every single polygon with the viewing volume is much to extensive. Hierarchical viewing volume culling can be realized through the spatial subdivision of the entire building.

Spatial subdivision can be achieved either by splitting the whole scene in different parts or by grouping the entire scene hierarchically in an intelligent way (should be done by the modeler creating the scene). The browser might insert bounding boxes where a *separator* is found in the structure of a VRML file. These bounding boxes could be used to perform a trivial rejection with the viewing volume without the need to split the scene or to build a BSP-tree.

Following the approach to subdivide a model of a building through splitting it, the building's rooms would subdivide the scene with only a few splits being necessary, hence resulting in a good performance of the algorithm. But spatial subdivision can be done very effectively wherever the remaining sub-scene is separable by a single plane into two parts without splitting.

I describe an algorithm, where it is necessary to include information about the layout of the building, using the walls of the rooms as separating planes. The

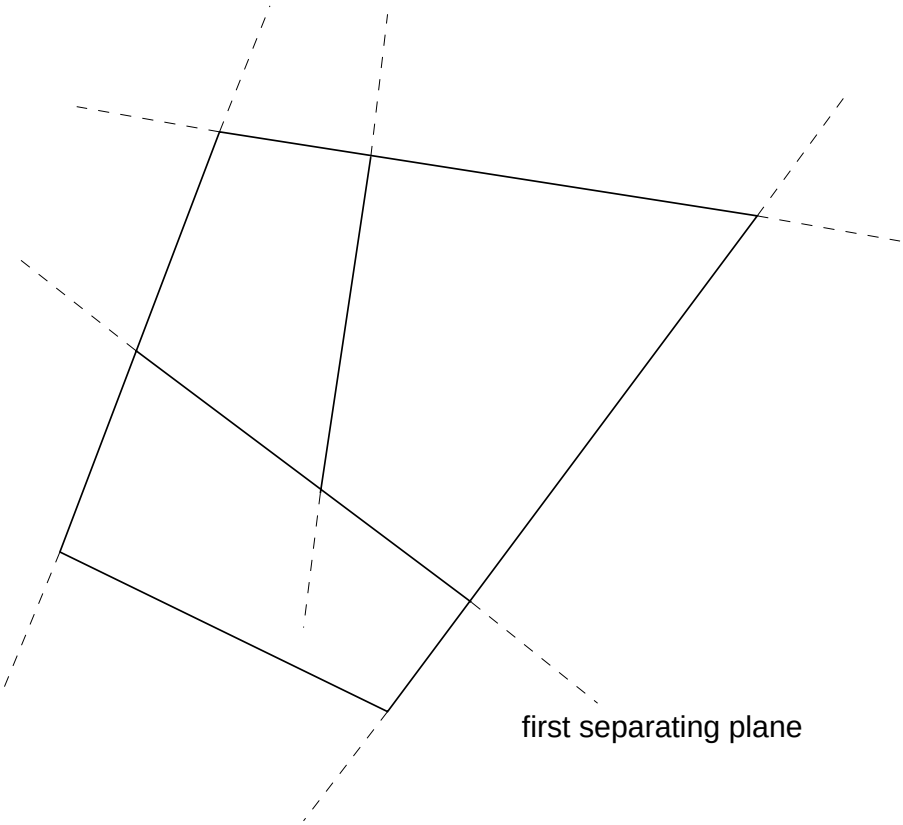

Figure 8.1: This layout fulfils the requirements.

prerequisite for my algorithm is that each of the floors of the whole building (I assume that the floors are separated by parallel planes) is recursively separable into two parts by a plane. The order of these separating planes must be given correctly. This requirement includes that the rooms are convex. Using the walls of the building as separating planes the description of one storey is reduced to a 2D problem since the walls are perpendicular to the floors.

8.2 The Algorithm in Detail

I will explain the algorithm using a layout and a camera position like in Figure 8.2. The bounding planes of the viewing volume are the front-, and backclipping plane and the four planes that are spanned by the COP and the four vertices of the viewplane window (see Section 6.8).

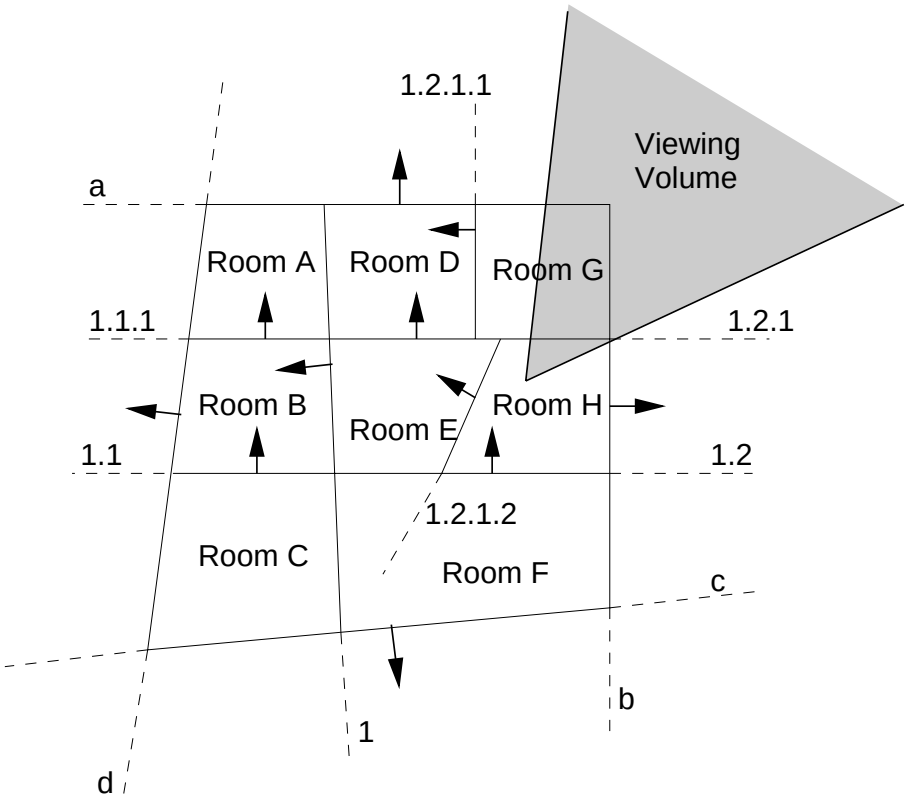

Figure 8.2: This layout and camera position is used to explain the algorithm.

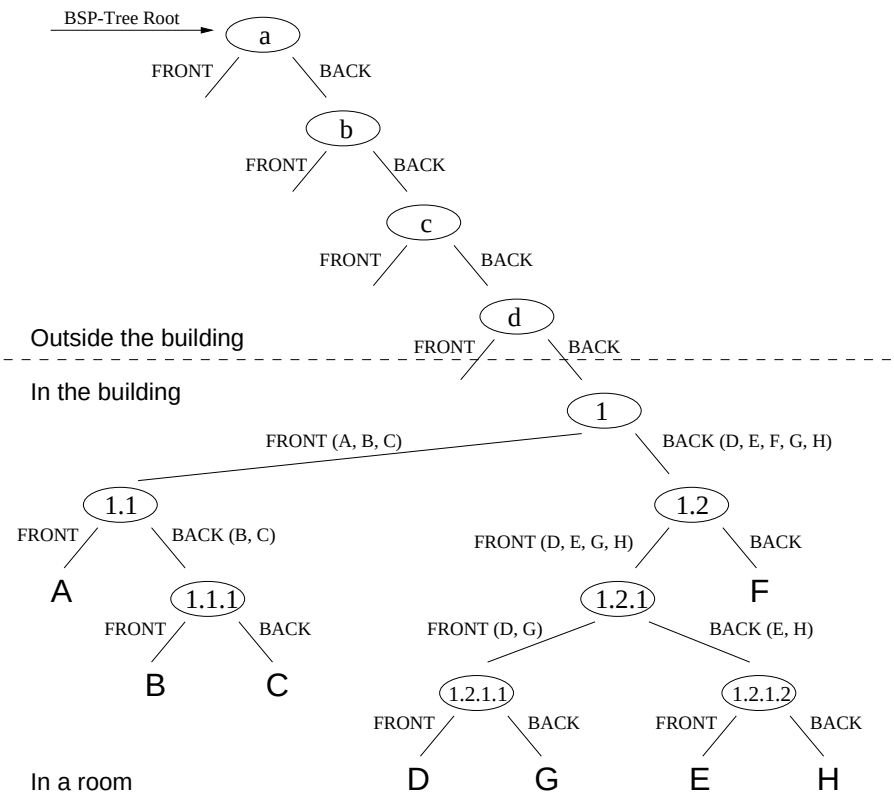

Figure 8.3: The BSP-Tree after inserting the structure of the building into an empty BSP-Tree.

The basic idea of the algorithm is to insert the structure of the building into a BSP-Tree first and leave the details of the scene for filtering through this hierarchical structure held in the initialized BSP-Tree. Due to having short paths in the BSP-Tree from the root to the information about the structure of the building we can quickly reject rooms that lie outside the viewing volume.

The structure is inserted through *logical planes* that represent the outer walls of the building and the walls between rooms. Figure 8.3 shows the BSP-Tree after inserting the structure of the building into an empty BSP-Tree. After the logical planes are inserted, the entire scene is inserted into the BSP-Tree (care must be taken with polygons lying on separating planes!). The complete BSP-Tree itself could easily be inserted into a BSP-Tree of the next hierarchy, let's say some buildings that are themselves recursively separable by planes.

Traversing the BSP-Tree in Figure 8.3 in back-to-front order would result in the

following order (see Section 6.6.4): `A, C, B, F, D, G, E, H.`
Traversing the same Tree in front-to-back order would result in the following order: `H, E, G, D, F, B, C, A.`

Viewing Volume Culling

The path to all nodes in the BSP-Tree that hold logical planes and being *in the building* – see Figure 8.3, nodes numbered from 1 to 1.2.1.2 – gives enough information to determine the bounding box of the corresponding room(s).

Traversing the BSP-Tree with hierarchical viewing volume culling requires testing the corresponding bounding box against the viewing volume when reaching one of these nodes holding logical planes and being in the building. Viewing volume culling can be done during a back-to-front traversal to render the scene without Z-buffering or during a front-to-back traversal to create a SVBSP-Tree. How the visibility test can be done very easily in VRweb with extensive use of already implemented classes is explained in Section 8.3. Traversing the tree in back-to-front order, rejection of hidden rooms would be performed in the following nodes of the BSP-Tree:

- 1 — reject rooms A, B, C.
- 1.2 — reject room F.
- 1.2.1.1 — reject room D.
- 1.2.1.2 — reject room E.

The only rooms that would be rendered are rooms G and H.

8.3 Extensive Use of Available VRweb Implementation

Viewing Volume Culling

The viewing volume is stored in a SVBSP-Tree containing the six polygons bounding the viewing volume like in Figure 6.6 (already implemented). While traversing the BSP-Tree the bounding boxes can be calculated *on the fly*. The bounding polygons for each plane are calculated when building the BSP-Tree for the entire scene and inserted into the facelist of the corresponding nodes.

The visibility test is performed by inserting a facelist containing the polygons of the bounding box into the SVBSP-Tree representing the viewing volume. If all bounding polygons are hidden then the subtree of the bounding box can be

rejected from being rendered. If a part of the bounding box is visible (intersects the viewing volume) then the contents of the corresponding room is at least partly visible. The SVBSP-Tree representing the viewing volume should not be extended in this case, it is only used to reject or accept bounding boxes.

Data Structure

The data structure of a node in the BSP-Tree needs to be extended in the following way:

```
class BSPTree
{
  private:

    QvMFFace facelist_; // pointer list to all
                        // faces(on the same plane)
    BSPTree* front_;    // BSP-tree that is completely
                        // infront of the plane, which
                        // is spanned by the facelist
    BSPTree* back_;     // same like front_ but in the back
    BSPTree* parent_;   // a link to the parent node,
                        // nil for the root
    QvMFFace* front_bounding_box_; // NEW: holding the bounding
                                   // box for the front subtree
    QvMFFace* back_bounding_box_;  // NEW: the same for the back
    int logical_plane_;            // NEW: a tag to determine the
                                   // type of the BSP-Tree node

};
```

The traversal of the final BSP-Tree would need to be rewritten in a way like this:

```
void BSPTree::drawBackToFront(const point3D& position, BSPMode mode)
{
  // plane_equation comes from the first polygon in the facelist_.

  if (Clipping::inFront(position, plane_equation))
  {
    if (back_)
       if ( (logical_plane_ && back_bounding_box &&
```

```
            viewing_volume->visible(back_bounding_box)) ||
            !logical_plane_ )
        back_->drawBackToFront(position, mode);
    if (!logical_plane) draw(position,mode);
    if (front_)
        if ( (logical_plane_ && front_bounding_box &&
            viewing_volume->visible(front_bounding_box)) ||
            !logical_plane_ )
        front_->drawBackToFront(position, mode);
  }
  else
  {
    if (front_)
        if ( (logical_plane_ && front_bounding_box &&
            viewing_volume->visible(front_bounding_box)) ||
            !logical_plane_ )
        front_->drawBackToFront(position, mode);
    if (!logical_plane) draw(position,mode);
    if (back_)
        if ( (logical_plane_ && back_bounding_box &&
            viewing_volume->visible(back_bounding_box)) ||
            !logical_plane_ )
        back_->drawBackToFront(position, mode);
  }
}
```

The class SVBSPTree is extended by the method *visible(BSPTree*)* which returns if at least one of the polygons held on the passed BSP-Tree (representing the bounding box) is visible or not. This method can be easily derived from the already existing method *filter*. This visibility determination works fine as long as the bounding boxes are smaller than the viewing volume. Would the whole viewing volume fit into a *room* then the corresponding bounding box would not be recognized as visible. Therefore the subdivision where the camera is positioned must be rendered in any case.

8.4 Representing the Structure of a Building

The information about the structure of a building is provided by the following planes:

- The floors of the different storeys until the roof.
- The outer walls.
- The walls between the rooms.

The planes representing walls between the rooms for each floor need to be given in the correct hierarchical order. In my example the structure could be represented in the following way, encoded in a schematic VRML syntax:

```
Group{
%floor
      Coordinate3 x x x
                  x x x
                  x x x
%ceiling
                  x x x
                  x x x
                  x x x
%outer wall 1
                  x x x
                  x x x
                  x x x
%outer wall 2
                  x x x
                  x x x
                  x x x
%outer wall 3
                  x x x
                  x x x
                  x x x

%outer wall 4
                  x x x
                  x x x
                  x x x
%separator 1
                  x x x
                  x x x
                  x x x
```

```
                Group{
%separator 1.1
                      Coordinate3 x x x
                                  x x x
                                  x x x
                      Group{
%separator 1.1.1
                            Coordinate3 x x x
                                        x x x
                                        x x x
                      }
                }
                Group{
%separator 1.2
                     Coordinate3 x x x
                                 x x x
                                 x x x
                     Group{
%separator 1.2.1
                           Coordinate3 x x x
                                       x x x
                                       x x x
                           Group{
%separator 1.2.1.1
                                 Coordinate3 x x x
                                             x x x
                                             x x x
                           }
                           Group{
%separator 1.2.1.2
                                 Coordinate3 x x x
                                             x x x
                                             x x x
                           }
                     }
                }
}
```

8.5 Further Extension to this Algorithm

VRML Extension for Indoor Scenes

To store information about the structure of a scene and not only the contents of a scene could help to improve rendering techniques. Becoming a standard, VRML modelers could take care that scenes are created hierarchically in an automatic way, therefore not increasing the load on the person who is creating a new scene. We could even take advantage of keeping hierarchical databases of scenes and objects, holding not only the contents of a scene but also good partitioning planes.

Spatial Subdivision and the SVBSP-Tree Algorithm

The same way the entire tree can be traversed in back-to-front order to render the scene with hierarchical viewing volume culling without Z-Buffering, it can be traversed in front-to-back order to build a SVBSP-Tree for visibility calculation providing on one hand hierarchical viewing volume culling as well and on the other hand a hierarchical trivial rejection of completely hidden subdivisions behind visible polygons.

Dynamic Scenes

Assuming that all objects contained in a building are always in a single room and never in two rooms at the same time (two rooms sharing an object) we could reduce the spatial subdivision BSP-tree algorithm to an algorithm that only separates all polygons of the entire scenes into the corresponding rooms and keeping them in form of lists. Rendering of the visible rooms would be performed with the Z-Buffer like in the original VRweb version. If there are animated objects in the scene then the corresponding objects would need to be deleted in these lists and inserted with the new coordinates if they are in visible rooms. Because VRML offers hierarchical modeling, the entire scene could be split into boxes/rooms during creation and directly stored in a hierarchical way in the VRML file. In this case not even a preprocessing would have to take place before starting the rendering process in the scene viewer.

Chapter 9

Outlook

The biggest issue about VRML 2.0 is that scenes are no longer static. Behaviours, like moving objects, work against the main advantages of BSP-Trees, which is having a static scene with only the camera moving. Therefore BSP-Trees do not seem to be a good approach concerning rendering VRML scenes any more. But they could still be used very well for spatial hierarchical subdivision to perform viewing volume culling as described in Chapter 8.

Level of detail (LOD) can be used to provide a hierarchical structure in terms of importance which seems to be necessary to achieve short rendering times for large outdoor scenes. It might be quite easy to introduce LOD with Z-buffering.

Combining LOD or animated objects with a BSP-Tree based rendering technique requires to separate the corresponding objects with logical planes and building the tree in a way that these objects are *leaves* of the BSP-Tree (see Section 3.3.3). With moving animated objects the corresponding hierarchical subpartitions need to be recalculated for every frame.

As mentioned in Section 5.4, it does not seem that it makes sense to support VRML 2.0 scripting in VRweb in its current C++ form. To support the scripting facilities in VRML 2.0, it seems to be better to support Java as the scripting language and incrememtally re-write part or all of VRweb in Java.

A new VRML 2.0 browser is being written from scratch at the IICM, completely coded in Java, probably using Java3D (see Section 1.4.2) which is intended to serve as a high-level platform-independent 3D rendering API with VRML 2.0 support.

3D graphics provide new ways of presenting information on the Net, whether it is information about the structure of the information like the Harmony Information Landscape or a VRML model of a library where you can open drawers and search for books or Web articles on a certain subject.

Multiperson, structured discussions in computer conferencing systems might be supported by a graphical representation of the flow of arguments while making the participants feel like meeting in a room and sitting together at a table.

Another area of future 3D graphics applications is teleteaching / distance teaching support. On one hand a 3D representation of an object that needs to be examined is easier to understand than a textual or 2D description. On the other hand collaboration of teachers and students and students with each other can be eased by building a virtual school or university. Distance is not a question any more. Students from Australia, Japan, and Austria meet in the virtual lecture room, being even taught in their own language through a multilinguistic Web server. The question how 3D graphics can support communication is a very interesting and not yet well researched topic.

All currently used operating system graphical user interfaces are 2D, the third dimension has not been introduced so far. Imaginative 3D interfaces for drag and drop, file copy, or a directory structure in three dimensions provide hope that user interfaces are really going to be *easy* to use.

3D graphics adds another dimension to the Web – and this additional dimension is so far only used in an already known way, to reproduce real world scenes and make them available for all users on the Net. But many other uses are still to be discovered and invented. That is the work for the future.

Concluding Remarks

In the course of this thesis I investigated visibility techniques to improve the rendering speed of VRML objects and scenes. The results of my work show that the visibility determination in terms of objects being in front or in the back of others is best solved with the standard Z-Buffer approach. Speeding up the rejection of parts of a model being outside the visible area (viewing volume) from being fed to the viewing pipeline would be best done through hierarchical spatial subdivision. Viewing volume clipping of OpenGL or Mesa is performed on a vertex/polygon bases and therefore not optimal.

Hardware graphics support like on a SGI Onyx is still very expensive, but it will only take a short time until 3D graphic chips are cheap and fast enough even for standard PCs.

We want to find algorithms suitable for many different platforms without any special hardware or software prerequisite, that are suitable for level of detail, changing scenes, and visibility calculation. Platform independence might be given by building on Java with Java3D, a Java graphics library with VRML support.

Improving the rendering method without improving the data structure of the 3D description (VRML) file is difficult. Bringing in a-priori knowledge about the model would help to find suitable improved rendering methods for different types of scenes like indoor-, outdoor-, or object scenes.

Bibliography

[AK94] Keith Andrews and Frank Kappe. *Soaring Through Hyperspace: A Snapshot of Hyper-G and its Harmony Client.* In Wolfgang Herzner and Frank Kappe, editors, *Proc. of Eurographics Symposium on Multimedia/Hypermedia in Open Distributed Environments*, pages 181–191, Graz, Austria, June 1994. Springer.

[AKM95] Keith Andrews, Frank Kappe, and Hermann Maurer. *Hyper-G: Towards the Next Generation of Network Information Technology. Journal of Universal Computer Science*, 1(4):206–220, April 1995. Special Issue: Selected Proceedings of the Workshop on Distributed Multimedia Systems, Graz, Austria, Nov. 1994. Available at `http://www.iicm.edu/jucs`.

[And] Keith Andrews. *The VRweb home page.* Available at `http://www.iicm.edu/vrweb`.

[And93] Marc Andreessen. NCSA Mosaic technical summary. Available at `ftp://ftp.ncsa.uiuc.edu/Mosaic/Papersmosaic.ps.Z`, May 1993.

[AP94] Keith Andrews and Michael Pichler. *Hooking Up 3-Space: Three-Dimensional Models as Fully-Fledged Hypermedia Documents.* In *Proc. of East-West International Conference on Multimedia, Hypermedia, and Virtual Reality (MHVR'94)*, pages 11–18, Moscow, Russia, September 1994.

[App] Apple Computer, Inc. *QuickDraw 3D.* Available at `http://www.info.apple.com/qd3d/`.

[App67] A. Appel. *The Notion of Quantitative Invisibility and the Machine Rendering of Solids.* In *Proceedings of the ACM National Conference*, pages 387–393, Washington, DC, 1967. Thompson Books.

[Bla] Black Sun Interactive, Inc. *Products from Black Sun, CyberHub.* Available at `http://ww3.blacksun.com/products/index.html`.

[BLCL+94] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. *The World-Wide Web. Communications of the ACM*, 37(8):76–82, August 1994.

[Cen] Center for Advanced Studies, Research and Development. *i3D - A New Dimension to Hypermedia.* Available at `http://www.crs4.it/~3diadm/`.

[CF89] Norman Chin and Steven Feiner. *Near Real-Time Shadow Generation Using BSP Trees.* In *Proceedings of SIGGRAPH '89, Computer Graphics, Volume 23, Number 3*, New York, July 1989. ACM SIGGRAPH.

[Cro77] F. Crow. *Shadow Algorithms for Computer Graphics.* In *Proceedings of SIGGRAPH '77, Computer Graphics, Volume 11, Number 3*, New York, July 1977. ACM SIGGRAPH.

[Cut74] E. Cutmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Phd dissertation, University of Utah, Computer Science Department, University of Utah, Salt Lake City, UT, December 1974.

[EMNM] Charles Eubanks, Rose McKeon, Dave Nadeau, and John Moreland. *VRML Repository.* San Diego Supercomputer Center. Available at `http://www.sdsc.edu/vrml/`.

[Eyl95] Martin Eyl. *The Harmony Information Landscape: Interactive, Three-Dimensional Navigation Through an Information Space.* Master's project, Graz University of Technology, IICM, Graz University of Technology, Austria, October 1995.

[FKN80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. *On Visible Surface Generation by A Priori Tree Structures.* In *Proceedings of SIGGRAPH '80, Computer Graphics, Volume 14, Number 3*, pages 387–393, New York, 1980. ACM SIGGRAPH.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[GKM93] N. Green, M. Kass, and G. Miller. *Hierarchical Z-Buffer Visibility.* In *Proceedings of SIGGRAPH '93, Computer Graphics, Volume 27*, New York, 1993. ACM SIGGRAPH.

[Hug95] Kevin Hughes. From webspace to cyberspace. Available at `http://www.northcoast.com/savetz/blist.html`, July 1995. Enterprise Integration Technologies, 800 El Camino Real, Menlo Park, CA 94025.

[Int] InterVista Software, Inc. *WorldView.* Available at `http://www.webmaster.com/vrml/`.

[Lin] Greg Linden. *WebView: A new way to search and view pages on the World Wide Web!* Available at `http://www.cs.washington.edu/homes/glinden/WebView/WebView.html`, University of Washington.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. *Composing User Interfaces with InterViews.* *IEEE Computer*, 22(2):8–22, February 1989.

[MA95] Vanessa Mayrhofer and Keith Andrews. *Harmony User Guide: Version 1.2*, November 1995. Available at `ftp://ftp.iicm.edu/pub/Hyper-G/papers/hug.ps.gz`.

[Mau96] Hermann Maurer. *HyperWave: The Next Generation Web Solution.* Addison-Wesley, May 1996. Available at `http://www.iicm.edu/hgbook`.

[MCR90] Jock D. Mackinlay, Stuart K. Card, and George G. Robertson. *Rapid Controlled Movement Through a Virtual 3D Workspace.* In *Proc. SIGGRAPH '90, Dallas, Texas*, pages 171–176, New York, August 1990. ACM.

[Mic] Microsoft Corporation. *Direct3D.* Available at `http://www.microsoft.com/mediadev/graphics/igrap.htm`.

[MMBL] Tamara Munzner, Daeron Meyer, Paul Burchard, and Stuart Levy. *WebOOGL: Integrating 3D graphics and the Web.* Available at `http://www.geom.umn.edu/software/weboogl/`, The Geometry Center, University of Minnesota.

[Nat] National Center for Supercomputing Applications. *NCSA Mosaic Home Page*. See `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html`, University of Illinois.

[Nay93] Bruce Naylor. *Constructing Good Partitioning Trees*. In *Proceedings of SIGGRAPH '93, Graphics Interface '93*, New York, 1993. ACM SIGGRAPH.

[NCS] University of Illinois NCSA. *NCSA Habanero Project Overview*. Available at `http://www.ncsa.uiuc.edu/SDG/Software/Habanero`.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[Net] Netscape. *Live3D*. Available at `http://home.netscape.com/eng/live3d`.

[NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. *A Solution to the Hidden Surface Problem*. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.

[Pau] Brian Paul. *The Mesa 3-D Graphics Library*. Available at `http://www.ssec.wisc.edu/~brianp/Mesa.html`, Space Science and Engineering Center, University of Wisconsin.

[Pes95] Mark Pesce. *VRML: Browsing and Building Cyberspace*. New Riders/Macmillan, 1995.

[Pic93] Michael Pichler. *Interactive Browsing of 3D Scenes in Hypermedia: The Hyper-G 3D Viewer*. Master's project, Graz University of Technology, IICM, Graz University of Technology, Austria, October 1993. Available at `ftp://iicm.tu-graz.ac.at/pub/Hyper-G/doc/pichler[12].ps`.

[POA+95] Michael Pichler, Gerbert Orasche, Keith Andrews, Ed Grossman, and Mark McCahill. *VRweb: A Multi-System VRML Viewer*. In *Proc. First Annual Symposium on the Virtual Reality Modeling Language (VRML '95)*, pages 77–85, San Diego, California, December 1995.

[Rad] Radiance Software International. *Ez3d*. Available at `http://www.radiance.com/Ez3d-VRPro.html`.

[Rob63] L. G. Roberts. Machine perception of three dimensional solids. Lincoln Laboratory, TR315, MIT, Cambridge, MA, May 1963.

[SBGS69] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, NTIS AD700375, Air Force Human Resources Lab, Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, September 1969.

[Sila] Silicon Graphics, Inc. *Cosmo Products*. Available at `http://www.sgi.com/Products/cosmo`.

[Silb] Silicon Graphics, Inc. *WebSpace*. Available at `http://www.sgi.com/Products/WebFORCE/WebSpace/`.

[Sla95] Mel Slater. Interactive computer graphics, describing and interacting with 3D virtual worlds. Course Notes for ICG Semester 2, Queen Mary & Westfield College, London, January 1995.

[Son] Sony Corporation, Inc. *Cyber Passage*. Available at `http://vs.sony.co.jp/VS-E/vstop.html`.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[Sun] Sun, Inc. *Java Overview*. Available at `http://java.sun.com/doc/Overviewjava`.

[TN87] W. C. Thibault and B. F. Naylor. *Set Operations on Polyhedra Using Binary Space Partitioning Trees*. In *Proceedings of SIGGRAPH '87, Computer Graphics, Volume 21, Number 4*, New York, 1987. ACM SIGGRAPH.

[Tor90] Enric Torres. *Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes*. In *Eurographics '90*, North Holland, 1990. Elsevier Science Publishers.

[TS91] Seth J. Teller and Carlo H. Séquin. *Visibility Preprocessing For Interactive Walkthroughs*. In *Proceedings of SIGGRAPH '91, Computer Graphics, Volume 25, Number 4*, New York, July 1991. ACM SIGGRAPH.

[War69] J. Warnock. A hidden-surface algorithm for computer generated halftone pictures. Technical Report TR 4-15, NTIS AD-753 671, University of Utah, Computer Science Department, University of Utah, Salt Lake City, UT, June 1969.

[Wor] Worlds, Inc. *Worlds Products: Worlds Chat, Alpha World, Gamma.* Available at `http://www.worlds.net/`.

[WREE67] C. Wylie, G. W. Romney, D. C. Evans, and A. C. Erdahl. *Halftone Perspective Drawings by Computer.* In *FJCC*, pages 49–58, Washington, DC, 1967. Thompson Books.

[X] Dimension X. *Liquid Reality.* Available at `http://www.dimensionx.com/products/lrindex.html`.