Interactive Ray Tracing of VRML Scenes in Java

by

Brendon C. Glazer

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 1, 1999

Author_____
Department of Electrical Engineering and Computer Science
February 1, 1999

Certified by_____
Seth Teller
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Interactive Ray Tracing of VRML Scenes in Java
by
Brendon C. Glazer

Submitted to the
Department of Electrical Engineering and Computer Science

February 1, 1999

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Java 1.1 programming language is used to create an applet that acts as a browser for Virtual Reality Modeling Language (VRML) version 2.0 scenes. The applet is capable of performing interactive ray tracing of those scenes. A standard Java-enabled web browser can be used to access a web page in which the applet is embedded. The embedded VRML browser offers two views of a loaded scene. One view is rendered by a standard graphics package that uses z-buffering, and can show wireframe or polygonal images. The other view is rendered by ray tracing techniques. Users can navigate through the scene in one of several modes, similar to standard VRML browsers. Parameters of the ray tracing algorithm are exposed to the user so that they may be modified interactively, allowing the user a degree of control over the rendering process during runtime. Additional control is provided by means of adaptive rendering, which renders a static view of the scene to increasingly greater resolutions over time.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

Acknowledgements

The author would like to thank the Computer Graphics Group of the Massachusetts Institute of Technology for providing the opportunity and focus necessary to carry out this work. Thanks in particular go to Professor Seth Teller, under whose supervision, guidance, advice, and encouragement this work was performed. Thanks also go to Professor Leonard McMillan, whose instructional ray tracer formed a solid foundation for the ray tracer presented here; Mark Matthews, for writing a very useful Java graphics package; and the numerous developers and users of Java and VRML, who are the ultimate inspiration for this work. Finally, thanks to the author's family and friends for their enduring support.

Table of Contents

List of Figures

Chapter 1 – Introduction

The rapid and widespread proliferation of the World Wide Web (WWW) in recent years has made information readily accessible to almost anyone. In addition to this wealth of information, numerous new technologies have arisen to support its distribution. VRML is one such technology. It has grown to be the standard basis for representing three-dimensional (3D) scenes and worlds on the web. Several VRML browsers currently exist that are available as plug-ins to common web browsers. Another new technology to emerge around the web is Java. This platform independent, object-oriented programming language has quickly risen to the forefront in the arena of network accessible computation.

This work is an attempt to merge these new technologies in a novel way to produce a tool for VRML content viewing. The goal is not only to create a different variety of VRML 2.0 browser, one which offers users the ability to interactively adjust options directly affecting the viewing experience, but also to demonstrate that ray tracing, as a representative high quality rendering algorithm, can be used to generate

detailed images in interactive environments where traditionally lower quality, more efficient algorithms have dominated. The ray tracing process can be adapted to suit such situations through optimization and compromise. For instance, an incremental rendering process that presents images of increasingly higher resolution and detail over time [4][22] allows the user to view a complete image at all times, without wasting precious computation time on vantage points of little interest. The time required to render a finely detailed image can be spent at an exact eyepoint of the user's choosing. Developing the software as a Java applet allows it to be accessible across platforms through a standard web browser.

The VRML browser presented here supports two views on a scene. The first view is similar to one of those common in other browsers in that it renders the scene using a graphics package built around z-buffering. This process utilizes a storage buffer that contains for each pixel in the image a color value and the depth, the z coordinate of the three dimensional space, hence the name z-buffering, of the associated object. The depth value is used to determine which objects are visible to the synthetic camera and which are occluded. This rendering process typically produces images of fair quality at relatively quick frame rates.

The second view on the VRML world depicts the same image as rendered by ray tracing. Ray tracing as a rendering algorithm traditionally yields high fidelity images at relatively slow speeds. This is because more work is spent determining the color value of each pixel. One or more rays are fired into the scene from the eye point through each pixel. These rays may intersect with an object in the scene, at which time more rays are fired into the scene from the newly found intersection point. At the

expense of running time, very realistic images can be achieved. Figure 1.1 shows a visualization of the ray tracing process.

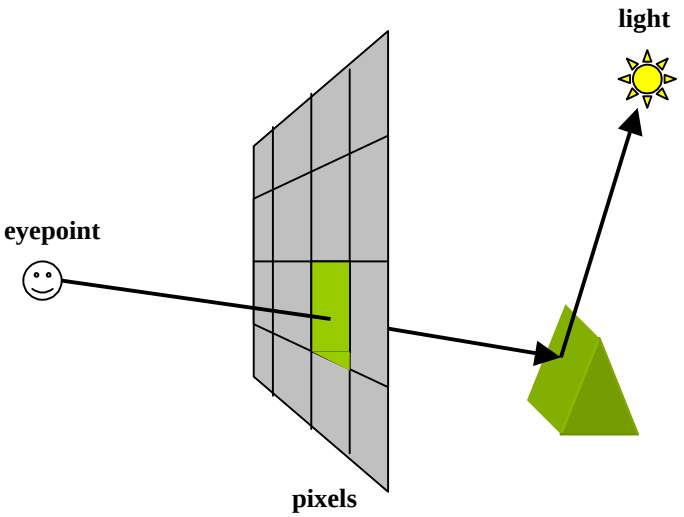


Figure 1.1 - Ray Tracing
The ray tracing algorithm fires rays from the eyepoint through each pixel of the image to determine what object is visible through that pixel and the corresponding color to shade the pixel.

Given the pair of views on a scene, users are able to navigate through the scene in any one of several modes typical of VRML browsers. There exists the ability to examine the scene as if it were encased in a manipulable crystal ball directly in front of the user, or to walk or fly through the scene as if it were in fact a real world environment capable of being traversed.

In addition to displaying the 3D scene, the browser's graphical user interface (GUI) presents users with the opportunity to modify parameters of the ray tracing renderer. There are options to change various aspects of the process, such as the depth

of recursion of the ray tree, the number of rays fired per pixel, and the locations within a pixel through which rays are fired. Furthermore, the user can choose between constant and incrementally increasing image resolution, as well as the frequency and manner in which screen updates occur.

There are several novel concepts to this project. First, the ability to view a VRML scene rendered through ray tracing is a new idea. The standard VRML browsers cannot render scenes to such a high level of detail. This feature provides the potential to achieve a higher level of realism in VRML worlds. Using a common modeling language such as VRML as input to the ray tracer also provides instant access to existing scene files.

Second, apart from the relationship between ray tracing and VRML, the ray tracing aspect of the project itself is innovative. Users are actually able to navigate the scene in real time. It is true that there is greater latency between frames for higher resolutions than for lower ones, but viewing a ray traced scene that is not static is an uncommon experience. A third novel idea is allowing the user to interactively control the rendering process. Various parameters of the ray tracing algorithm can be dynamically modified during runtime and their changes will be quickly evident in the image. Fourth, performance analysis is not typical either, and estimated time bounds on rendering provide clues as to what level of detail is realistic given runtime constraints.

The technical innovations of this work are enhanced by the fact that no other such tool currently exists. The work as a whole has not been carried out before. A ray tracing VRML browser is a new concept.

The success of this work can be evaluated based on the quality of images produced and the amount and ease of interaction possible with the browser. The ray tracer renders views of a scene whose quality rivals or outshines that of the images produced by the z-buffered renderer. At the same time, the view remains responsive to user navigation. Performance of course depends on the parameters of the ray tracing process at any given time. Constrained by current hardware and software limitations (Java is not known for its speed), the ray tracer cannot be expected to outperform its companion z-buffered renderer except for images of very few pixels, but results are encouraging.

The work described here has important implications. It demonstrates the versatility and usefulness of the ray tracing process. Ray tracing is not reserved for high end rendering that must take hours of computation time to complete. Images of significant detail can be achieved in real time. Frames of animation do not have to be prerendered, and interaction is allowed. Advances in computer speeds will certainly not hurt the ray tracing algorithm, but by adaptively ray tracing to various levels of resolution, time is not spent on unnecessary computation. The user may allocate more time to rendering if more detail is desired, or less time if a high frame rate is of key importance. Furthermore, such decisions can be made dynamically during runtime at the user's discretion, when changes are deemed to be beneficial. Parameters of the running process do not have to be set and finalized before the application has started. See Appendix A for an example of a possible user scenario.

Chapter 2 – Previous Work

The different technologies utilized and merged through this work, like ray tracing, VRML, and Java, have each been the topic of previous study. In some cases, the relationships between these technologies have also been researched. This chapter provides a glance into some of the relevant past and current work being carried out to explore how these tools may be improved and used.

The idea of ray tracing has existed for quite a long time, and has been studied in detail over the years. In particular, optimizations and improvements in the running time of the basic algorithm have long been researched to combat lengthy rendering times. One area into which much study has gone is the use of hierarchical data structures for spatial subdivision to reduce the number of ray-object intersection tests that must be performed when firing a ray into a scene. The idea is that a ray only needs to traverse the cells in the data structure with which it intersects, and therefore must only test for possible intersection with the scene objects contained in those cells.

Octrees have been used as one method for subdividing scene space [8]. Beginning with the root node, which contains the bounding volume of the scene, each node is recursively subdivided into eight child nodes of equal size. The recursion ends when a node contains some minimum number of objects. This structure is easy to construct and to use because all nodes at a given depth in the tree are of equal size.

Another commonly used structure is the k-d tree [7][17]. This spatial subdivision tree is built by splitting space with a single axial plane at each node, resulting in two children. The attractive aspect of the k-d tree is that the location of each splitting plane can be determined by arbitrary criteria. For instance, the location may be chosen to minimize the difference between the number of objects on each side of the new plane.

Clearly, the choice of space subdivision technique has impact on rendering time. This relationship has even led to research into methods for characterizing scene complexity and from these measurements determining the optimal subdivision structure to use [16]. Such scene complexity measurements have also been used to predict the best termination criteria to specify when recursively constructing a given subdivision tree [18][19].

A novel idea for spatial subdivision is the use of amalgams [22]. Amalgams are bounding volumes that are capable of being rendered. A ray tracer may choose to traverse an amalgam, in which case it processes the objects inside the amalgam, similar to other subdivision methods. These objects may themselves be amalgams. The ray tracer may instead opt to render the amalgam itself, ignoring its internal complexity.

This has the beneficial effect of reducing the number of intersection tests by eliminating tests with objects internal to the rendered amalgam.

Some study has also gone into attempting to determine in advance how long a scene will take to render. One such probabilistic method takes into account the ratio of subdivision cell surface area to size [15]. This measure is then used to determine the probability that a ray will intersect an object. Probabilistic running time can then be computed.

Recently, research has been done to explore the viability of interactive rendering through ray casting. A new technique, frustum casting, has been developed to speed up the rendering process by taking advantage of the spatial coherence of primary rays fired into a scene [21]. In most scene environments, each object visible from the synthetic eye point occupies an area in screen space that is much greater than one pixel. Such objects are hit by all of the rays fired through those pixels that depict the image of the object. Restated in terms of spatial coherence, there is a group of rays occupying some solid angle that intersects with the object. Frustum casting attempts to capitalize on such coherence by firing rays into the scene as a bounded group. If only one object occupies the entirety of the viewing frustum, then all of the primary rays will hit that object and no conditional intersections must be calculated. If no object is visible within the frustum, then none of the rays fired will hit any objects that may be in the scene. When multiple objects intersect with the viewing frustum, the frustum is subdivided into four child frusta, and each child attempts to traverse the scene as a whole. This recursive process continues until frusta are reduced to the size of one pixel, in which case the frustum is effectively a single ray. The frustum casting system developed also

utilizes progressive rendering. The viewed image is initially rendered at low resolution, but will be rendered at progressively higher resolutions if the image remains static for some period of time. The combined optimizations of such techniques as progressive rendering and frustum casting allow the ray caster to operate at interactive frame rates. This interactivity provides users with a measure of flexibility in determining image characteristics during runtime. Scenes are navigable at some fixed image quality or at a user-specified frame rate. A detailed user interface displays information about the state of the rendering process and allows parameters of the ray caster to be modified dynamically.

The relationship between VRML and Java has also been studied, although not quite as extensively as ray tracing. The VRML specification itself outlines a set of Java classes and methods to use for connecting Java code to a VRML scene [3]. Interacting with the scene then results in callbacks to sections of code, which can have influence on the scene itself. This concept has been taken a step further to suggest that VRML can be made to act as a user interface to Java applications [6]. Working in the opposite direction, Java toolkits have been designed to enable developers to create VRML content programmatically. JVerge [5], created by Justin Couch at The Virtual Light Company, is a set of Java classes based on the VRML node hierarchy that allows instantiation of those nodes within Java code. Such toolkits are designed to make the task of creating VRML content and browsers easier. In fact, VRML browsers written in Java do exist. For example, VRwave [23] is a browser written entirely in Java, developed by the Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology, Austria.

Core to a VRML browser is its 3D graphics package. While Sun's Java3D specification [20] had been posted at the time this work was being performed, a finished product had not been released to the public in time to use. Apart from Sun's 3D application programming interface (API), packages are available that are based on OpenGL by SGI. One package that provides some of OpenGL's functionality is JGL [10], by Mark Matthews at the Purdue University CADLAB.

Java has furthermore been used to create several functioning ray tracers. Several simpler ones have been written as class projects. Matt Armstrong and Yi Ma at the University of California at Berkeley have developed one that has several nice features, and appears to be of good quality [1].

Although much work has been done with Java, VRML, and ray tracing as individual tools, the exploration of combinations of these technologies has been limited. This work is an attempt to change that. Ray tracing is used to render VRML scenes, instead of scenes written in a new modeling language crafted specifically for the ray tracer. VRML scenes are capable of being rendered to fine detail through the ray tracer. The use of Java not only displays the versatility of that language, but also makes the resulting applet easily accessible over the web. The merging of these distinct technologies can give rise to further study of possible combinations of dissimilar tools.

Chapter 3 – Design

The ray tracing VRML browser presented here has relatively loose design constraints. A general analysis of the required software components yields a handful of distinct modules. Most obvious are the two separate graphical renderers. The polygon renderer, which uses a technique such as z-buffering, provides an efficient and approximate representation of the VRML scene. The ray tracing renderer provides a highly detailed more exact representation of the scene. Another necessary software module is the VRML parser. This portion of the browser is responsible for reading in the VRML scene file and converting it into a software representation with which the renderers can work. A subtler piece of the whole is the security model, which allows the applet to operate outside of the restrictions placed on it by the web browser. Finally, the graphical user interface (GUI) serves as the front end to the software, displaying information for the user and accepting input. These components are discussed further below.

3.1 – Polygon renderer

This graphics engine assumes the responsibility of displaying an accurate representation of the VRML scene through algorithms suited for the task. This renderer must provide images of reasonable detail at efficient frame rates. While some leeway is afforded the ray tracer due to its unusual role as a real-time renderer, the polygon graphics package must always show the user a visually clear picture of the VRML environment. This gives the user a good idea of location with respect to the scene objects at all times so that navigation can be performed with some knowledge of direction or destination. When the ray tracer may be displaying an image of poor resolution, it is the duty of this renderer to convey enough visual information about the scene so the user does not feel hopelessly lost or confused about the environment. In order to accomplish this, the polygon renderer can use common efficient algorithms, such as z-buffering, and approximations, in terms of shading. The renderer used here is Mark Matthews' JGL graphics package [10].

Not required, but a useful feature, is the ability to toggle this renderer between wireframe and polygon views. Shaded polygons show a much more accurate image of the scene, but may have undesirably lengthy latency between frames. A wireframe view will typically take a much shorter time to refresh, and therefore can provide a faster means of navigation.

3.2 – Ray tracer

The ray tracer does not need to be overly complex. It is only required to perform the basic functions of the ray tracing algorithm, such as ray shooting and

testing for intersections with objects. One necessity, though, is the ability of the ray tracer to accept all of the primitive geometric constructs that make up VRML objects. These include cones, cylinders, boxes, and spheres. More complex geometric entities, like elevation grids, indexed face sets, and extrusions, can be built with planar polygons, and this work takes advantage of such construction.

In addition to standard ray tracer functions, this renderer may provide additional options. One extra can be the ability to do jittering. This allows the ray tracer to shoot rays through random points across a pixel, instead of spacing the rays uniformly. Other options might be optimizations to speed up ray-scene intersection tests. Spatial subdivision schemes can significantly reduce running times.

3.3 – VRML parser

The applet needs some way of getting information contained in the VRML scene file into the rendering software. This gateway consists of the VRML parser. The parser is capable of reading the textual VRML file and transforming the information contained therein into a hierarchical scene graph structure. This tree-like structure can then be traversed to draw the scene. Any renderer can be capable of drawing the scene once an appropriate traversal method has been written for that particular renderer. This module has a straightforward purpose, and takes care of the file input task.

3.4 – Security

Web browsers, in particular Netscape Navigator, commonly restrict the capabilities of Java applets in order to maintain reasonably secure operation. This small

arena of allowed operation is known as the Java sandbox. Two restrictions in particular that hinder the work presented here are the inability of the applet to read files stored on the machine running the applet and the inability to make connections to other computers across the network. These problems arise when the user attempts to load a VRML file, whether it is located on the local machine or on a remote machine. In order to allow the applet to work as intended, the browser must be made to grant these two requested privileges. The security model can overcome these obstacles by simply overriding browser security altogether. A smarter and more secure way of gaining the desired capabilities is to surpass only the two hindering restrictions, thereby leaving remaining security measures intact.

3.5 – GUI

The GUI is the user's interface to the software, and has several responsibilities. First, and perhaps most important, is its ability to display relevant information in a clear and concise manner to the user. This information consists of the images output by the two rendering modules as well as state information. State information can be the current parameters of the rendering algorithms, settings for viewing options, or other similar items of interest to the user.

In addition to being able to display data, the GUI must be capable of handling user input to the applet. Parameters and settings can be modified during runtime, so information should be displayed in text boxes, as choices among radio buttons, or as a selections from dropdown menus. One of the most important types of input to the applet is the method of navigation through the VRML world. Movement by clicking

and dragging the mouse provides great functionality along with ease of use, and the GUI must be able to accommodate such interaction.

Chapter 4 – Implementation Details

Putting together the various components that comprise the ray tracing VRML browser takes a good deal of work. This chapter attempts to explain how different pieces fit together and what features are provided.

4.1 – GUI

The GUI displays information to the user and accepts input from the user. The screen is vertically divided into three general areas. The top area holds file information, including which file is currently being viewed. The middle section contains the images of the VRML scene as rendered by the two graphics engines, along with a status box to inform the user about tasks being performed by the rendering algorithms. The bottom area displays algorithmic parameters and option settings. A visualization of the applet's GUI is provided in Figure 4.1.1.

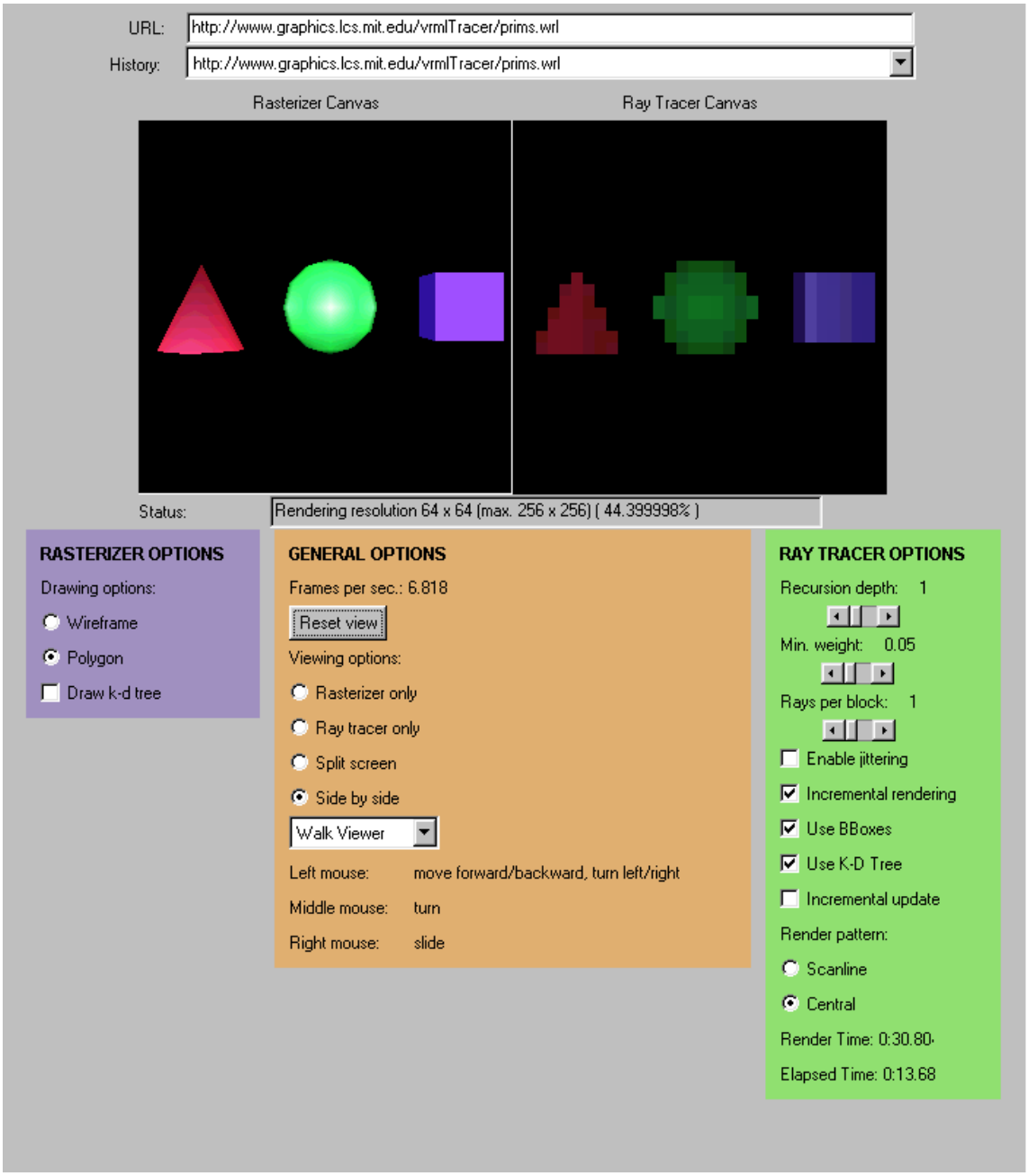


Figure 4.1.1 - Applet
This is a scaled down image of the applet as it would appear in a web page.

File information, located at the top of the applet, refers to Uniform Resource Locators (URLs) of VRML files that have been or currently are loaded for viewing in the current applet runtime. The URL fields appear much like those found in typical

web browsers. There is a text box where new URLs may be entered, as well as a dropdown list box that contains the URLs of all files that have been loaded during this runtime. When a new URL is entered into the text box, it automatically gets added to the list of visited URLs in the history dropdown. This makes it convenient for the user to revisit previous files or just keep track of what files have been viewed. Furthermore, the history dropdown can be preloaded with sample VRML files so that users do not have to have scene files on hand to experience the applet. Appendix C explains how such parameters can be passed to the applet.

Views on a loaded VRML scene are presented in the middle of the screen. This is where the act of navigation takes place. Navigation is accomplished by simply clicking and dragging the mouse in either one of the two views. The views capture all mouse movement while a mouse button is held down, provided the button was pressed while the mouse pointer was located inside one of the views. The response of the synthetic camera to any mouse movement depends on the current mode of navigation. Each mouse button is assigned a separate resulting motion in each of the modes. The different types of navigation will be discussed below.

Directly under the two VRML views is a status box. This is nothing more than a text box that accepts no input. Its purpose is to inform the user as to what task a renderer is performing as it is running. For instance, when loading a scene, the status box shows when the VRML file is being parsed, when spatial subdivision structures are being constructed from the scene, and when timing estimates are being taken. Once a scene is loaded, the status box displays what resolution the ray tracer is currently rendering and what percentage of that resolution has been computed so far. In essence,

the status box is a progress indicator informing the user when and what work is being done.

The bottom portion of the GUI, which holds all other data pertaining to rendering algorithms and viewing options, is further subdivided horizontally into three sections. One section displays general viewing options. These options effect how a scene is viewed and navigated. A group of radio buttons control which image is being shown by the renderers. The two renderers can render the same image side by side, or they can each render half of one image, producing a split screen effect. It is also possible to turn off completely one of the renderers, so computation time can be devoted to the remaining task.

This area, too, contains a dropdown list box from which the navigation mode can be selected. There are three available modes: examination viewer, walk viewer, and fly viewer. The examination viewer treats the scene as a tangible object in front of the user. The scene can be rotated, panned or zoomed. The walk and fly viewers allow the user to feel immersed in the scene. Flying allows navigation in any direction, while walking constrains the viewer to a horizontal plane. However, the plane of constraint can be moved up or down by the user. In each of the navigation modes, there are textual hints present that show which mouse button is responsible for which type of movement, so that navigation is as easy as possible. Figure 4.1.2 shows what effects each mouse button has in the different navigation modes.

	Examine Viewer	Walk Viewer	Fly Viewer
Left mouse	rotate	move forward/backward, turn left/right	move forward/backward, turn left/right
Middle mouse	zoom	turn	turn
Right mouse	pan	slide	slide

Figure 4.1.2 - Navigation Modes
Movement is accomplished by dragging the mouse while holding down one of the mouse buttons. The type of motion achieved depends on the mouse button pressed and the navigation mode.

Another control in this area is a button to reset the two views back to their original states. It is possible that the user may navigate to some undesired vantage point, at which time pressing this button will result in both views reverting to their starting positions. The last bit of information shown here is the instantaneous frame rate of the displays. As the user navigates through the scene, a running average of the time it takes for the images to redraw after movement occurs is tallied. This average is used to calculate the frame rate, which is how many redraws are occurring per second.

The other two areas in this lower section of the applet are devoted to options and information pertinent to one or the other renderer. One area is about the polygon renderer, while the other is about the ray tracer. In each, the various options can be changed by means of radio buttons, sliders, and checkboxes. The various types of parameters and data shown here will be discussed further in the sections for their respective renderers.

4.2 – Security

The security module is used to overcome some of the limitations placed on Java applets by web browsers, in particular Netscape Navigator. As previously mentioned, the applet needs the ability to read VRML files stored on the local computer as well as files located on remote machines across the network. These two functions are not allowed under standard security restrictions imposed on Java by web browsers. The method used here to grant the desired privileges is to electronically sign with a digital certificate the Java archive (JAR) file containing the applet code. The digital signature acts as a sort of guarantee to the user that the applet does not contain malicious code. Therefore, it can be trusted to act only as expected when Netscape security privileges are granted by the user. Microsoft's Internet Explorer is a bit more flexible with security restrictions, and the signed applet will run under that browser without extra prompting or required work by the user.

Netscape Signing Tool version 1.0 [14] was used to digitally sign the JAR file. This command-line program takes a digital certificate and the directory containing the Java class files to sign and performs two steps. First, it signs each file with the certificate. Second, it packs the class files into one convenient JAR file. The digital certificate used to sign the applet may come from any trusted certificate provider. For the purposes of this work, a test certificate that can be generated by the Signing Tool itself was used.

However, a signed JAR file alone does not overcome security obstacles. Each of the desired privilege restrictions must be explicitly overcome in the applet's code.

The Netscape Capabilities Application Programming Interface (API) [9] makes this exceedingly easy to do. Using the API it takes only simple Java method invocations to enable each of the desired privileges. Privileges may be granted and denied in code at any point.

In order to load a web page containing the signed applet, Internet Explorer users have only to point the browser to the page. Navigator users must go through a necessary process of downloading and installing the digital certificate into the local copy of Netscape before the applet will run. This process is described in more detail in Appendix B. Once the certificate is imported into Navigator, the browser will automatically recognize, each time the applet is loaded, that it is requesting specific security privileges to be granted. Netscape will pop up dialog boxes explaining to the user what privileges are requested and what certificate has been used to sign the applet. See Figure 4.2.1 for an example of what these dialog boxes look like. It is the user's option to grant or deny each of these requests on an individual per-privilege basis. In this case, if the two requested privileges are granted, then the applet will be able to read specified VRML files and will run as intended. If either privilege is not granted, the applet will load, but will not be able to run.

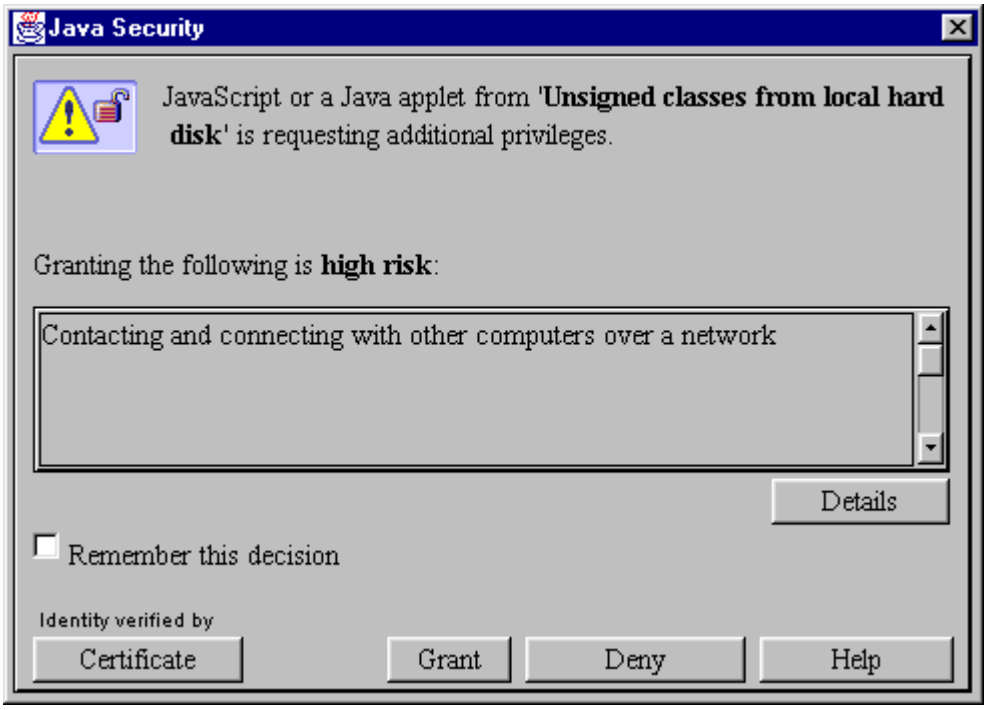


Figure 4.2.1 - Netscape Security Permission Dialog
Netscape will pop up dialog boxes similar to the one above in order to process the applet's request for security privileges. The user must click the "Grant" button in each dialog in order to use the applet.

4.3 – VRML parser

Reading a VRML file into the applet is done through the use of a VRML parser. While there are a few available, the particular one used is IICM's *pw*. This parser was used by IICM in their Java VRML browser, VRwave. Fortunately, it is also available for download as a standalone package. The *pw* parser is quite extensive, covering most of the VRML specification. It also provides framework methods for traversing the hierarchical scene graph structure output by the parser. These traverser methods can be modified to perform any purpose useful in processing the data in the tree, such as

displaying the information in the scene on screen or inserting scene objects into a spatial subdivision structure like an octree.

4.4 – Renderers

The two graphical renderers form the basis for the entire applet. Each does a significant amount of processing to massage the information contained in a VRML scene file into something that can be drawn and manipulated on screen. Currently, not all VRML nodes are processed by the renderers. See Appendix D for a list of supported nodes.

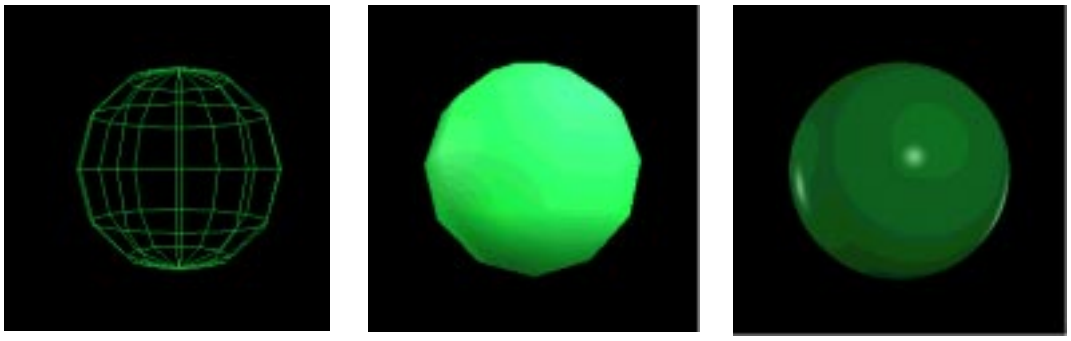
4.4.1 – Polygon

Mark Matthews' JGL graphics package [10] is used here for polygon and wireframe rendering. It is similar to OpenGL, and provides appropriate functionality and performance for this work. It is capable of drawing both wireframe figures and solid shaded polygons, and maintains reasonably fast refresh rates.

Before any drawing can take place, the graphics engine must first calculate exactly what is to be drawn. For instance, a VRML scene might contain a sphere. However, JGL does not inherently know how to draw a sphere. The shape must be broken down into a discrete set of vertices, which can then be divided into planar polygons for rendering. Therefore, when a VRML scene is first loaded, the applet performs a traversal of the scene graph to compute all the information that will later be necessary to draw the scene. This build traversal stores information about objects in the scene in data structures corresponding to the type of object. Traversing a sphere node

in the scene graph creates a new sphere-specific data structure. This might include the radius, as well as a list of vertices and polygons comprising a discrete tesselation of the sphere. Such processing is not trivial, and needs only to be performed once to calculate data required to draw the sphere. The build traversal routine performs similar tasks for other VRML nodes, storing data in separate structures designed for the nodes.

When called upon to render an image of the scene for the user, another traversal routine is called, this time with the purpose of calling appropriate JGL methods on the data previously computed to draw objects on the screen. As an example, consider the sphere. The draw traverser can easily produce a sphere on screen by running through the list of polygons making up the calculated discretization of the sphere and drawing each of them in turn. See Figure 4.4.1.1 for examples of both a wireframe and a solid sphere rendered by the polygon renderer.



(a) **(b)** **(c)**

Figure 4.4.1.1 - Spheres
(a) A wireframe sphere and (b) a solid sphere as rendered by JGL. (c) A sphere rendered by the ray tracer is shown for comparison.

This renderer can switch between wireframe and polygonal views with ease. The user chooses the type of view via a choice between radio buttons on the applet. This sets a flag in the draw traversal method that dictates whether vertices are to be connected by simple lines or composed into shaded planar polygons. In the case where polygons are drawn, refresh rates can be too slow to allow for quick navigation of the scene. Therefore, during periods of navigation the renderer always draws wireframe models on screen. During periods of static viewing, solid polygons are again drawn.

The ability to draw wireframe figures also proves useful for showing spatial structures on screen, such as the k-d tree optimization used by the ray tracer. This structure encompasses the scene, and would occlude all scene objects if drawn opaquely. The user may choose to view the k-d tree by marking a checkbox in the GUI. When this box is checked, a wireframe outline of all the k-d tree's nodes are drawn on top of the current image of the scene. The view of the k-d tree can be toggled on and off as desired.

4.4.2 – Ray tracer

The ray tracer used here was built on top of an existing skeletal framework written by Leonard McMillan [11]. Features were added and modified as necessary to produce an interactive renderer capable of yielding results in real time.

Similar to the polygon renderer, the ray tracer must gather information about the VRML scene from the scene graph. Here, also, a build traverser is used to go through the scene graph and create appropriate objects that the ray tracer is capable of rendering. These objects contain material and position data that the ray tracer needs in

order to accurately represent the shapes on screen. The ray tracer does not have the ability to traverse the scene graph each time it must draw the scene. It cannot push and pop matrices and materials on and off a stack to keep track of appearance and positional data as can the polygon renderer. All the information pertaining to a single object must be stored with that object.

As objects are created for the ray tracer during build traversal, bounding boxes are computed for each geometric entity. These bounding boxes are stored with their contained objects. Both world space and object space bounding boxes are computed. World space bounding boxes can be used to insert objects into a spatial subdivision structure, while object space bounding boxes can be used for ray intersection testing. Typically, it is much easier and faster to test whether a ray intersects with a box shape than with any given object. So when the ray tracer checks for a ray intersection with an object, it can first test the ray against the object's bounding box. If the ray does not intersect with the bounding box, then no costly test against the actual object is necessary. Only in the case that the ray intersects with the bounding box does a test have to be performed against the object. So using bounding boxes for preliminary intersection testing makes the ray tracer more efficient. However, whether or not the renderer actually does take advantage of the bounding boxes is up to the user. A checkbox in the GUI gives the user the option to turn off bounding box intersection testing so that differences in rendering times can be observed.

Once all objects for the ray tracer have been created from the scene graph, a spatial subdivision structure can be created to further optimize ray-scene intersection testing. In this work, the structure used is a k-d tree [2]. The k-d tree is built by first

computing the bounding box for the entire scene. This forms the root node of the tree. Each node is then subdivided by a splitting plane to minimize the sum of objects straddling the splitting plane and the difference between the number of objects wholly on either side of the plane. This recursion terminates when a node in the tree only contains one object, or when no splitting plane can be found to distribute objects evenly between child nodes.

When shooting a ray into the scene, the ray tracer must first calculate the entry node of the ray into the k-d tree. The ray is then tested against all the objects in that node for possible intersections. If no intersection is found, the ray tracer walks along the ray into the adjacent node, again testing for intersections with objects in that node. This walk along the ray continues until either an intersection is found or the ray exits the k-d tree altogether. See Figure 4.4.2.1 for an example of walking along a ray through a k-d tree, which is shown in two dimensions for simplicity, but can easily be extended to three dimensions.

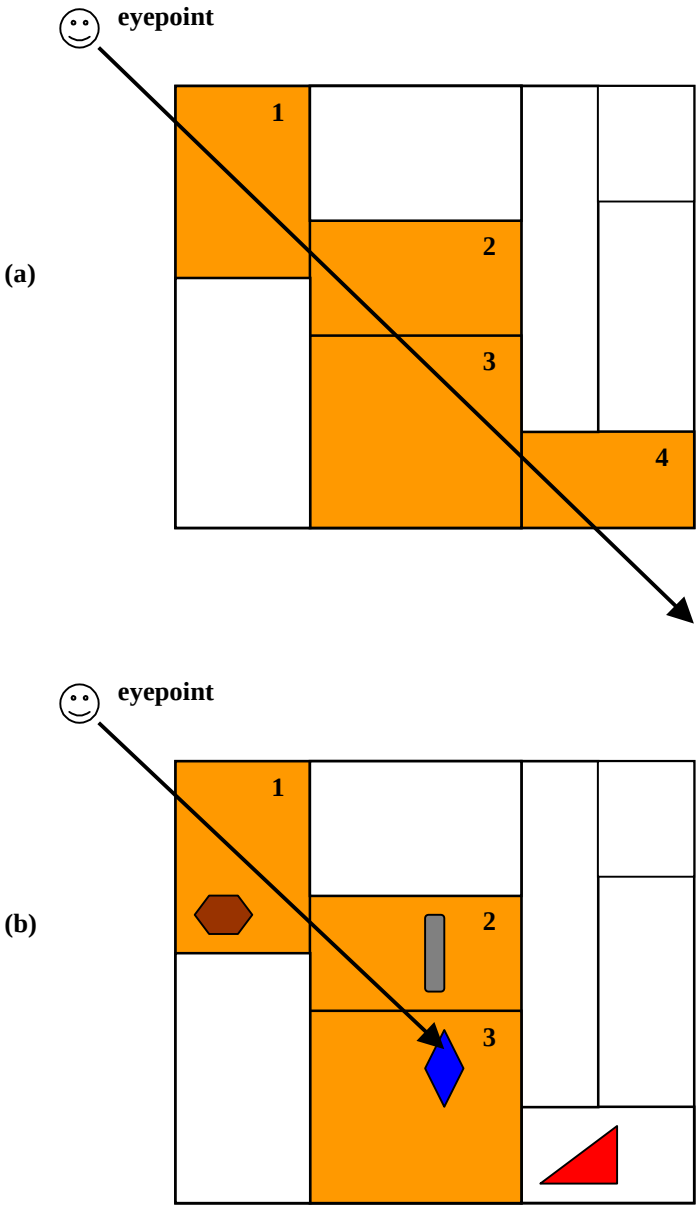


Figure 4.4.2.1 - Shooting a Ray Through a K-d Tree
The ray passes through the shaded cells in the k-d tree. It is tested for intersections against objects one cell at a time, in the order indicated by the cell numbering. (a) The ray does not hit any objects, and passes completely through the tree. (b) The ray hits an object in cell #3 and does not have to be tested against any further cells.

Such spatial subdivision of objects speeds up the computationally intensive task of finding ray-scene intersections. Bounding box testing can be used in combination with the k-d tree for added benefit. However, the user has the option, through a checkbox control, of turning off the k-d tree and forcing the ray tracer to perform naïve ray-scene intersection tests. This causes each ray to be tested against all objects in the scene to find the closest intersection.

Since the k-d tree for each scene is calculated, its root node, which is the scene's bounding box, can be used to center the scene in the view. The synthetic camera can be placed at a location away from the scene and set to look at the scene. This is beneficial because the user does not have to go searching for objects in the scene. The contents of the environment are right in front of the user, and can be explored at will.

Other parameters of the ray tracer are also available for modification during runtime. The renderer is capable of firing reflection rays from intersection points on objects to gather more shading information about that point. At each intersection point, rays are fired to light sources in the scene to gather direct lighting information. Reflection rays provide data from indirect lighting, as reflected off of other objects on the scene. This iterative process goes on until one of two termination criteria is met. First, a maximum depth determines for how many iterations the reflection process may continue. Figure 4.4.2.2 graphically depicts how different values for maximum depth affect the ray tracing algorithm. Second, if a reflection ray does not add significant shading information, falling below the value of some minimum weight, then no more reflection rays will be fired from that point. These two termination criteria are

represented on screen by numerical values as well as sliders to change those values. The user can alter them dynamically to achieve a desired rendering state.

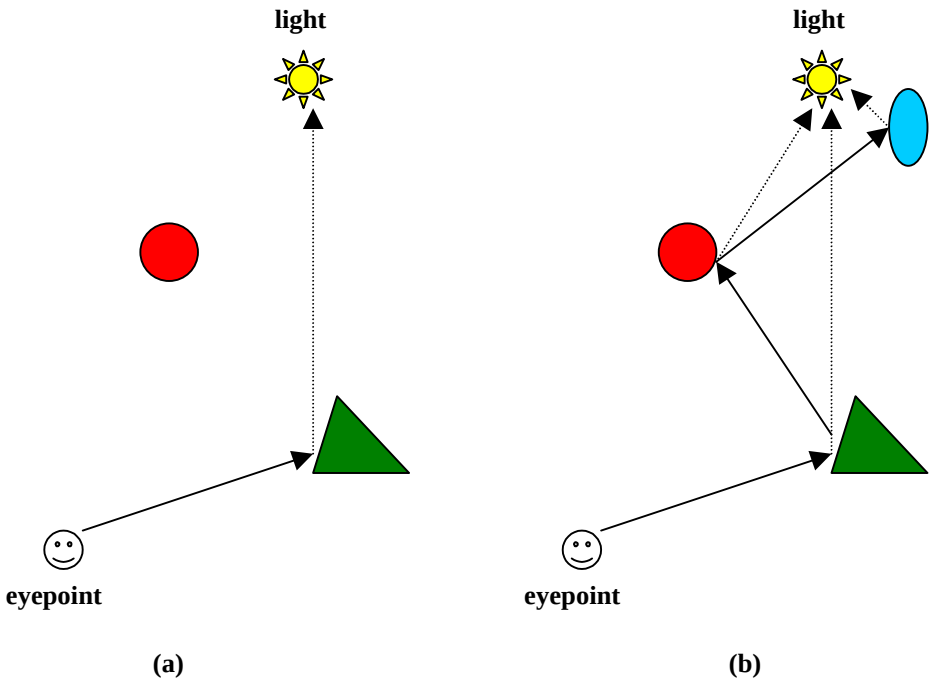
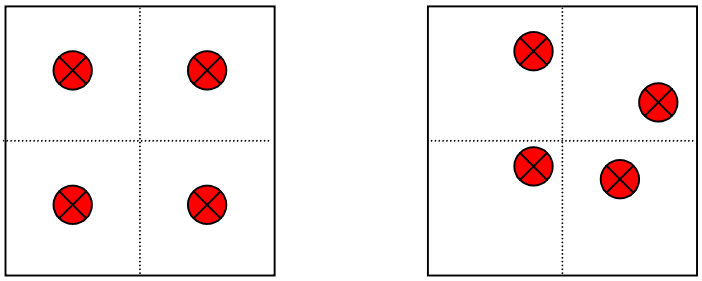


Figure 4.4.2.2 - Maximum Depth
(a) Maximum depth equals one. (b) Maximum depth equals three. More detailed shading information can be obtained with greater depth values. At each intersection point, rays are fired from the point to light sources in the scene to get shading information from direct lighting. Reflection rays provide shading information from indirect lighting being reflected off of other objects. The dashed arrows indicate rays being fired directly to the light source from intersection points.

Users are also given the ability to modify the number of rays fired per pixel, or per block of pixels, through another slider on the GUI. A user may want to increase the number of rays fired per pixel in order to achieve anti-aliasing effects in an image. By default, the ray tracer will then fire the set number of rays through a pixel, evenly spacing the rays across the pixel. The pixel is uniformly subdivided, and each ray is fired through the center of one of the resulting subdivisions. So if four rays are fired,

they will be fired in a uniform two by two pattern. The option to perform jittering, controlled by a checkbox on screen, causes the ray tracer to randomly choose where each ray will be fired through a subdivision of a pixel. Figure 4.4.2.3 shows the difference between enabling and disabling jittering when multiple rays are fired through each pixel. This randomness can aid anti-aliasing efforts also.



(a) **(b)**

Figure 4.4.2.3 - Jittering
(a) Four rays are fired per pixel, evenly spaced across the pixel. (b) Four rays are again fired per pixel, but this time jittering is enabled. New random locations at which rays intersect the pixel will be calculated for each pixel.

One of the most interesting features of the ray tracer is its ability to perform incremental, or progressive, rendering. This process allows the ray tracer to render images of increasing resolution over time. The ray tracer engine itself runs on its own thread, separate from the applet's thread. Each time the user navigates the scene, the ray tracer thread restarts. During periods of user inactivity, the thread continues to run unhindered. This rendering thread is designed to render very low resolution images at first, and to increase the resolution as time goes on. For instance, at the beginning of the thread's life-cycle, it will split the viewable scene into four blocks of pixels,

creating a two-by-two image. Each block is treated as a single pixel. So the ray tracer will fire as many rays through each block as it would through a single pixel at maximum resolution. The ray tracer then averages and displays only one color for that entire block. When this two-by-two image is complete, the ray tracer next splits each existing block of pixels into four more blocks. So the two-by-two image becomes a four-by-four image, with each of the resulting 16 blocks displaying its own color. Rendering continues in this fashion until each block only consists of a single pixel. Figures 4.4.2.4 shows an example of progressive rendering.

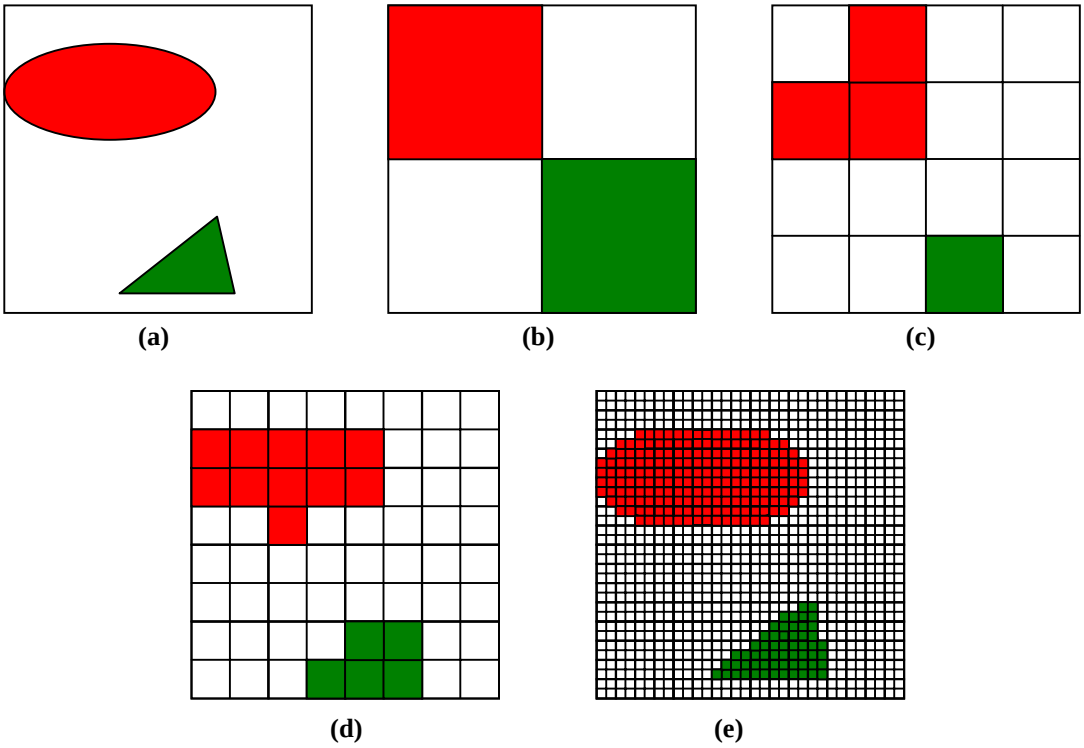
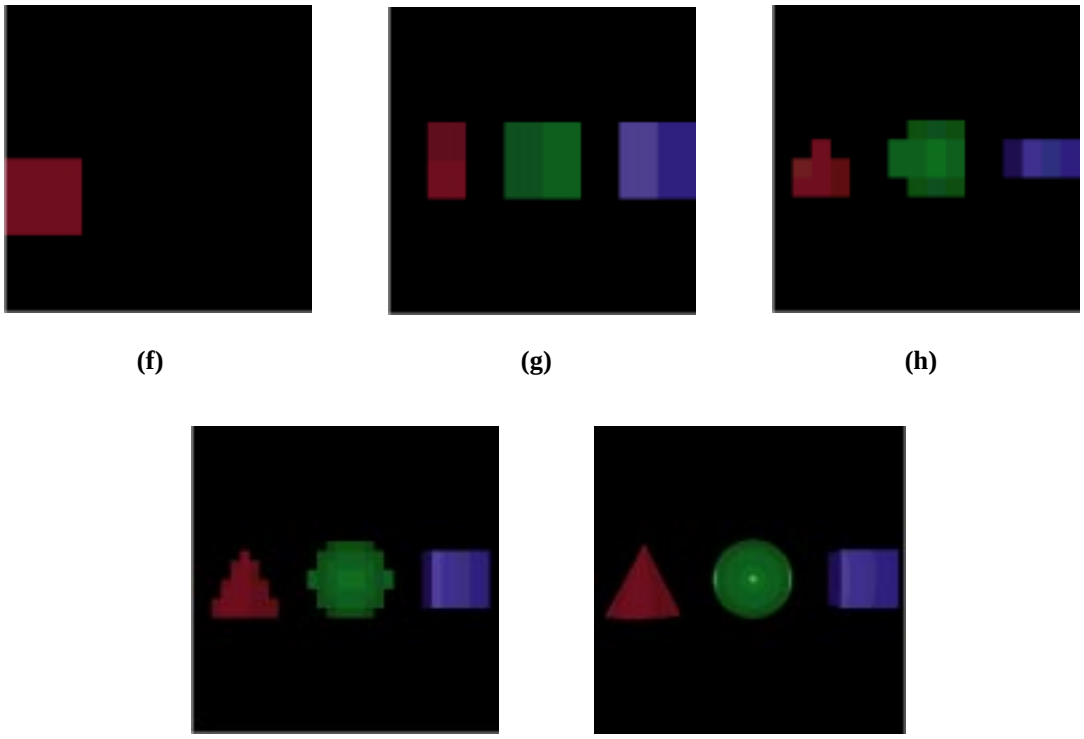


Figure 4.4.2.4 - Progressive Rendering
An example of progressive rendering. (a) This is how the finished scene should look. (b), (c), (d) The first three resolutions of progressive rendering of the scene. (e) Several resolutions later, as the size of each block approaches one pixel.
(continued on next page)



(f) **(g)** **(h)**

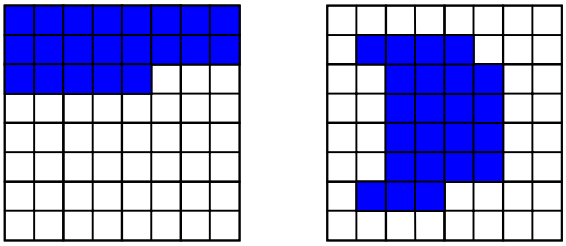
(i) **(j)**

Figure 4.4.2.4 continued – (f) – (j) This is a similar rendering progression done by the applet.

This method of rendering has the benefit of being able to quickly render an image of the entire scene, although that image may be quite crude. However, the user might not be looking for fine detail at every point in the scene. The user can quickly move through a scene to find a location of interest, at which point navigation ceases and rendering time is spent where it is most desired. Computation time does not get wasted on arbitrary pixels. The user influences allotment of computational resources, and frame rate for navigational purposes remains responsive. Of course, the ray tracer can be set to render on a pixel-by-pixel basis if the user so chooses. Incremental rendering is toggled by a checkbox on the applet.

The method of updating an on-screen image is also modifiable. Through a checkbox, the user may elect to have the ray tracer refresh its image after it processes each pixel or block, or to update the image only after every pixel or block for a given resolution has been computed. This gives the user the choice to let the ray tracer finish an image before displaying it or to be able to watch the ray tracer's progress graphically as it performs its tasks. However, setting the ray tracer to update the screen after each block is rendered significantly increases the time it takes to completely render the scene.

Users can set the pattern in which blocks or pixels in an image are computed, too. There are radio buttons on the applet that allow the user to choose between a scanline or central pattern. The scanline pattern causes the ray tracer to render pixels in a left-to-right, top-to-bottom arrangement. Choosing central causes the ray tracer to first render pixels at the center of the image, then expand outward to finish at the edges of the image. The idea behind this pattern is that the most interesting part of an image is probably near the center of the image. The central pattern causes this part of the image to be rendered first. So when the image is being updated after each pixel is processed, the center of the image will achieve greater detail before the rest of the image. See Figure 4.4.2.5 for an example of how an image is updated using each of these methods.



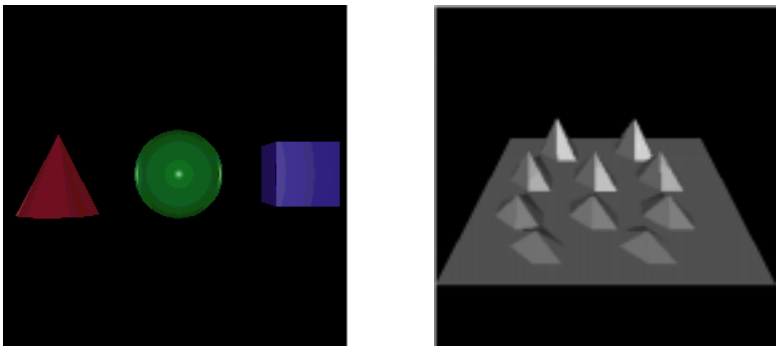
(a) **(b)**

Figure 4.4.2.5 - Image Update Patterns
The two methods of updating an image are shown. (a) The scanline pattern updates the image from left to right, top to bottom. (b) The central pattern updates the image in rectangular patterns, beginning with the center pixels and expanding outward to the edges.

The ray tracer also attempts to provide the user with an estimation of how long it will take to render a scene at the current resolution given current parameters. As the ray tracer renders a scene, timing measurements are taken to record how long it takes to render one pixel or one block of pixels in the image. For each resolution, these measurements are merged into a running average of the time it takes to render one block of pixels for that resolution. An estimate of how long it will take to completely render the image can be calculated based on the total number of blocks in the image at the current resolution. The number of blocks multiplied by the running average at any moment in time produces this instantaneous time estimate. The time estimate is continually updated as the ray tracer completes each block in the image. This time, along with the amount of time elapsed since the ray tracer began rendering at the current resolution, is displayed on screen for the user's convenience.

Chapter 5 – Results

One of the main concepts that this work is intended to convey is that ray tracing does not have to be a static process. It can be adapted to become a dynamic algorithm where user interaction is allowed. Indeed, the ray tracer developed here shows that the role of the ray tracer as a real-time rendering tool is a viable assignment. Navigation of the environment is possible through arbitration between image fidelity and refresh rate. The renderer is capable of displaying the best possible representation of a scene in any given allotment of time. As the user moves through a scene, the ray tracer does not have the time commitment necessary to show much detail in its images. The user is forcing the ray tracer to achieve interactive frame rates by conceding image resolution. However, when movement ceases and the view of the scene becomes static, the ray tracer will be allowed to continually update its display, finally yielding the sort of high definition image one expects from a ray traced depiction of a scene. Figure 5.1 shows fully rendered sample images.



(a) **(b)**

Figure 5.1 - Rendered Images
Sample images of scenes that have been completely rendered by the ray tracer. When given enough time, the ray tracer can produce images of high fidelity. (a) These three primitives took a little over 4 minutes to render completely. (b) This extrusion was completed in just about 5 minutes.

The lower resolution images of the scene can serve as previews of how the finished image will appear. This affords the user the time and ability to render precisely the desired view of a scene, instead of wasting hours of computational time after which the resultant image might not be exactly what is wanted. Corrections to the location and orientation of the camera can be made before resources are spent.

The parameters of the rendering algorithm, in addition to camera information, are also modifiable during the rendering cycle. When changes are made to those values, results are almost immediately apparent. These runtime options provide a very interactive environment in which the user maintains control of the rendering process throughout the cycle. Image characteristics can be dynamically tailored to suit appropriate purposes.

The work described here also has value in terms of novelty. There is no other existing tool that performs the same tasks. The applet succeeds in bringing together

various distinct technologies. It is able to provide representations of VRML scenes that exceed other VRML browsers with respect to image quality. It also maintains an interactive setting in which the usually static ray tracing process can be carried out. Using VRML as the modeling language for the ray tracer makes it easy to craft new scenes. A new modeling or scripting language does not have to be designed or learned, and existing VRML files are valid candidates for the renderer's use. Assembling this tool for deployment over the web also opens new doors for the technology on display. There is virtually no download or installation required, and even knowledge of VRML, Java, or ray tracing is unnecessary. The applet can be appreciated for its technical achievements, or it can be enjoyed simply on an aesthetic basis as an interesting graphical viewer. See Appendix E for online locations of the software and this document.

5.1 – Limitations

Although this work is very encouraging, it does have its limitations. One of the most glaring limitations to the software is its slow performance. This drawback cannot be helped completely. Ray tracing is an intrinsically slow process. It takes a great amount of computation that cannot be avoided. While there are further optimizations that can be made, the process will not yield stellar performance. Furthermore, Java as a programming language lags behind other languages in terms of sheer runtime speed. In time, advances in hardware and Java technology will help boost exhibited performance levels. See Figure 5.1.1 for a chart of the times taken to progressively render the sample scene depicted in Figure 5.1 (a).

Blocks in image	Pixels per block	Time to render	Time per:	
			Block (sec.)	Pixel (sec.)
4 (2 x 2)	16384 (128 x 128)	0 min., 0.06 sec.	.015	9.2e-7
16 (4 x 4)	4096 (64 x 64)	0 min., 0.16 sec.	.010	2.4e-6
64 (8 x 8)	1024 (32 x 32)	0 min., 0.28 sec.	.004	4.3e-6
256 (16 x 16)	256 (16 x 16)	0 min., 0.99 sec.	.004	1.5e-5
1024 (32 x 32)	64 (8 x 8)	0 min., 3.63 sec.	.004	5.5e-5
4096 (64 x 64)	16 (4 x 4)	0 min., 14.72 sec.	.004	2.2e-4
16384 (128 x 128)	4 (2 x 2)	1 min., 0.74 sec.	.004	9.3e-4
65536 (256 x 256)	1 (1 x 1)	4 min., 4.14 sec.	.004	3.7e-3

Figure 5.1.1 - Render Times
This table shows times taken to render progressive resolutions of a scene consisting of three primitives, shown in Figure 5.1 (a). These times reflect running the applet through Microsoft's Internet Explorer 4.0 on a 200 MHz machine, with the applet option to show incremental screen updates turned off. The ray tracer is shooting one ray per block, depth is set to one, and minimum weight is set to 0.05. For this simple scene, ray tracer render times recorded when using the k-d tree do not show significant differences from times taken when not using the k-d tree.

Poor performance results in slow refresh rates. However, it also manifests itself in the form of delay in responding to user interaction. Navigation at times is noticeably bumpy. This can occur when a complex scene is being viewed and it takes a long time for the polygon renderer to display it. Similarly, there sometimes is a lag between the time the user marks a checkbox or radio button on the GUI and the time when the

change can be observed in the images. This can be attributed to the ray tracer thread monopolizing computation cycles.

The large amount of computation that the applet performs is memory intensive. The applet must maintain scene information, including object data and spatial subdivision structures, in memory. Therefore, the complexity of scenes capable of being viewed through the applet depends on the amount of memory available to the machine being used.

Another limitation to this work, aside from performance and memory issues, is the small set of VRML nodes currently recognized by the browser. VRML scenes can be very dynamic, with scripted motions and sensors to detect user interaction with the world. For instance, the drawer of a desk might slide open when a user clicks on it. However, the browser does not process such dynamic nodes. The applet provides a dynamic viewing experience of static scenes, where scene objects do not change with time relative to one another. Also, object coloring is rather simple in the applet. Texture mapping and by-vertex coloring, which are available through VRML, are currently not handled by the polygon or ray tracing renderers.

One last limitation to the applet is the small size of the rendered images. In the interest of achieving real-time interaction, the size of the images is kept small so that the ray tracer can complete rendering in some reasonable amount of time. If performance of the renderer could be significantly improved, it would be possible to have larger viewing areas that could be rendered in similar lengths of time.

The limitations discussed here are not insurmountable. Further work to extend and optimize the applet can reduce and possibly eliminate some of them. The applet does satisfy reasonable expectations, but improvements are always possible.

Chapter 6 – Conclusions

The results obtained from performing this work are encouraging. It is possible to create a ray tracing renderer, which accepts as input scenes described in a common modeling language such as VRML, that can be run as an interactive graphical tool. The algorithm, previously considered solely in the realm of static processes [13], is in fact shown to be quite dynamic. Real time ray tracing can be a viable option when choosing a rendering method for some non-critical applications. At the very least, this concept warrants further research.

One of the major hurdles that had to be overcome for the completion of this work is the sandbox in which web browsers constrain Java applets' activities. The notion of restricting applet operation to achieve secure interaction is worthwhile. However, the steps necessary to overcome this obstacle are tedious and can be confusing. The spirit of Java is platform independence, but browser independence should be considered also. Methods performed here to ensure applet operation under Netscape Navigator result in browser-specific code and certificate installation. It seems

reasonable to suggest that a simpler way be proposed to deal with security issues. Perhaps such capabilities could be included in the Java language specification, and native code of any sort would then become transparent to the developer and user. Standardization of security models would allow for a richer development arena.

All in all the work done here satisfies expectations. Distinct technologies were merged into one cohesive functioning unit. Performance, in terms of rendering speed, is adequate, although not exceptional. Other technical aspects and innovations of the work, such as the quality of images produced and the level of interactivity possible with the algorithms and environment, make up for the high latency that can be encountered. Achievements here show promise of accomplishments in future related work, and it is anticipated that continued research will yield positive results.

Chapter 7 – Future Work

The work detailed here is a functioning whole, but can be extended and improved upon in several ways. Foremost in the group of possible enhancements are optimizations to the renderers, particularly the ray tracer. That engine could be streamlined to allow for faster intersection testing. It might be useful to incorporate other spatial subdivision structures, in addition to the k-d tree, and determine probabilistically which would be most advantageous to use for a given scene.

Another worthwhile expansion to the applet would be to allow for handling of a larger subset, or even the whole, of the VRML specification. Currently, only a small portion of the specification is utilized, including geometric, appearance, and transformational nodes. In addition to those nodes, others could be covered, such as the viewpoint node and the various types of sensors. A viewpoint defines a location from which to view the scene within the environment, and sensors allow for a greater level of interaction with the scene. It would be interesting also to see appearance

enhancements. Transparency and by-vertex coloring could be included. Furthermore, texture mapping would be a welcome addition to the browser's display capabilities.

There are also features that would add to the general usability of the applet. For instance, a headlight that could be toggled on and off would be an improvement over the current use of default static lighting in scenes. It might be useful to be able to selectively render an arbitrary area of a scene, too. This could be accomplished by dragging the mouse across the image to render a bounded region, or even just those pixels traversed by the mouse's movement. Another feature that could prove worthwhile is some sort of printing option. Users could choose to print the image displayed by either or both of the renderers directly through the applet, possibly saving the output as a bitmap.

More complex analysis of algorithmic performance could provide meaningful data to the user. Information about past and present performance could yield access to historical trends. Graphs of data such as frame rate might show how navigation speed is affected by changes in algorithm parameters. Calculating low level temporal benchmarks could form the basis for better bounds on estimation of rendering time. Instead of using the time to fire a ray into the scene, computations could be based on the time to perform mathematical operations like multiplies or additions. Measurements like this that also take scene characteristics and viewer position into account could be even more precise.

Currently, the browser runs quite easily under Internet Explorer, but only runs under Navigator once a digital certificate has been downloaded and installed into the local copy of Netscape. A reasonable idea for improvement would be to find an

alternative, simpler method of getting the applet to run under Navigator. A way might be found that would eliminate the tedious process of certificate installation and make the browser security issue completely transparent to users.

One other improvement that could be made to the software would be to transform the ray tracer into a distributed renderer [12]. The ray tracer could assign blocks of pixels to be rendered by various threads, and these threads could then run on different processors in the same computer, or on entirely separate machines. This has the possibility of greatly improving rendering speed, and could allow for better viewing of complex VRML scenes.

There are numerous other areas in which work on the applet can be performed. Some modifications are suited to short-term projects, while others are more substantial and could comprise topics of in-depth research. Limitations on the work are only imposed by imagination, and a definitive end to possibilities is not easily envisioned.

Appendix A – Example User Scenario

A typical session using the ray tracing VRML browser begins when a user points a Netscape Navigator web browser to the URL of a web site containing a page in which the VRML browser is embedded. If this is the user's first time accessing the applet, the digital certificate required for viewing it must first be downloaded and installed into Netscape's certificate database. Assuming this has been done, Netscape will pop up dialog boxes querying whether or not the user wishes to grant the applet the security privileges that it is requesting. After these privileges have been granted, either on a temporary or permanent basis, use of the applet can proceed.

Once the applet is completely loaded, it begins rendering a default VRML scene. The user may wish to inspect this scene. From the dropdown list of navigation modes, the user chooses "Examine Viewer". By clicking and dragging the left mouse button on one of the two images of the scene, the environment is rotated in front of the user. The left mouse button is then released, and the right mouse button pressed in order to zoom in on the scene to view one of the objects more closely.

Next, the user clicks the “Polygon” radio button, under Drawing Options on the GUI, to enable the polygon renderer to draw a shaded polygonal representation of the scene instead of the currently displayed wireframe representation. This done, the user is better able to compare the difference in image quality between the two views. Realizing that the ray tracer is capable of rendering even more detail, the user moves the slider for “Rays per block” to nine, and changes the slider for “Recursion depth” to three.

Now the user has grown tired of this scene, and wants to view another file. After glancing through the list of files preloaded in the history dropdown, the user decides to enter the URL of a new VRML file directly into the URL text box. When the URL is typed, the user presses the Enter key and loading of the new scene begins. When the first images appear, the user observes that this file contains a complex scene. The ray tracer shows that it will take quite a long time to render the scene with reasonable detail. The user then turns off the ray tracer by clicking the “Rasterizer only” option under Viewing Options, so that navigation in the polygon renderer will be faster. The user changes the mode of navigation to “Walk Viewer” and proceeds to walk through the scene until a point of interest is found.

The user next checks the box marked “Draw k-d tree” to see what the k-d tree for such a complex scene looks like. After this, the user turns the ray tracer back on by clicking on the viewing option “Side by side”. Not interested in watching the ray tracer incrementally render at increasing resolutions, the user un-checks the box marked “Incremental rendering”. The ray tracer begins rendering the scene pixel by pixel, starting with the center of the image.

The user watches the ray tracer run for a while, until the status box shows that 25% of the scene has been rendered. This central 25% of the image shows the detail that the user wanted. Having seen enough of this scene, the user decides to quit the applet for the time being. This is done by either pointing Navigator to another web site or by exiting the application altogether.

Appendix B – Installing the Digital Certificate in Netscape

Before the applet presented here can be used in Netscape, the digital certificate used to sign the Java archive must be installed into Netscape's certificate database. This is a one-time task, but it can be fairly tedious. The steps below will attempt to walk the user through this process of importing the certificate into Netscape. It is assumed that the certificate is available through a hypertext link on a web page, as is the case at the applet locations given in Appendix E.

Step 1 – Save the certificate to the local disk, with the extension ".cacert".

This can be done a number of ways. The user can **hold down the Shift key and left-click on the link**, or **right-click on the link and choose the "Save Link As…" option from the pop-up menu**, or **left-click on the link to view the certificate and then choose "Save" from the "File" menu at the top of the Navigator window**.

It is important to note that **the certificate must be saved with the filename extension ".cacert"**. So two possible names might be **"x509.cacert"** or

“digcert.cacert”. The part of the name preceding the extension can be anything of the user’s choosing.

Step 2 – Import the certificate into Netscape.

Step 2.1 –

This process can be initiated by **either dragging and dropping the certificate icon into a Navigator window** or **by pointing the browser to the local certificate’s URL**. If performing either of those two actions does not result in some security dialog boxes being displayed, then the user must skip to Step 3 before attempting Step 2 again.

Step 2.2 –

When the browser recognizes that a certificate is being imported, it will pop up some dialog boxes to advance through the process. The user can press the “Next” button on each dialog box until arriving at a checkbox labeled **“Accept this Certificate…”. The user must check this box before going on!** Figure B.1 shows what this dialog box should look like.

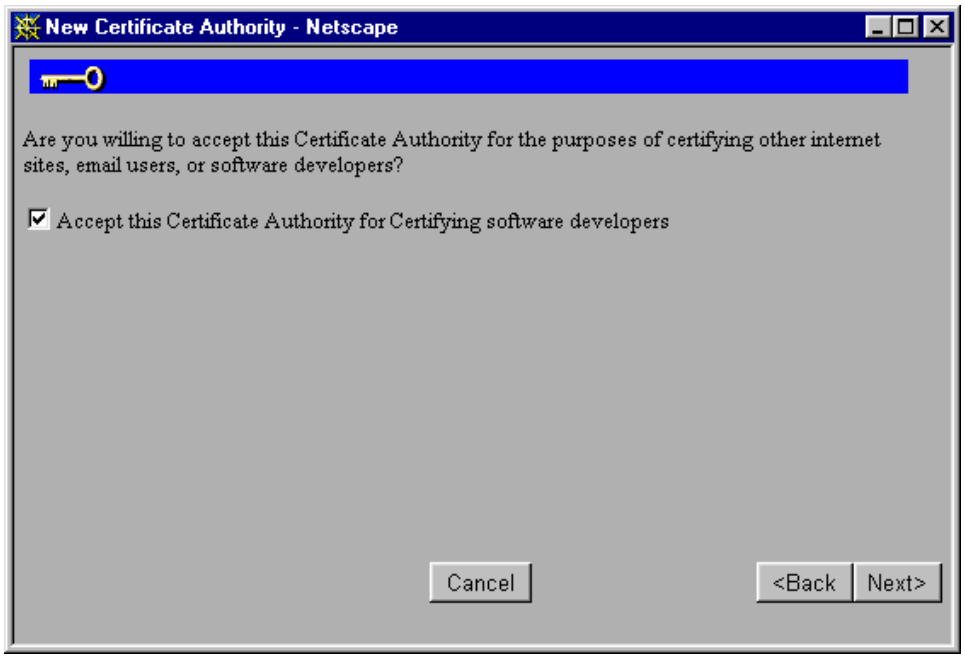


Figure B.1 – "Accept Certificate" Dialog
The user must mark the checkbox in this dialog, as shown here, in order to correctly import the digital certificate.

Step 2.3 –

From the dialog box with the above checkbox, the user may press "Next" through a couple more dialogs until asked for a **nickname for the certificate**. The certificate can be named anything, but it should be something that will later be identifiable, such as **"VRMLTracer Certificate"**. See Figure B.2 for a depiction of this dialog box.

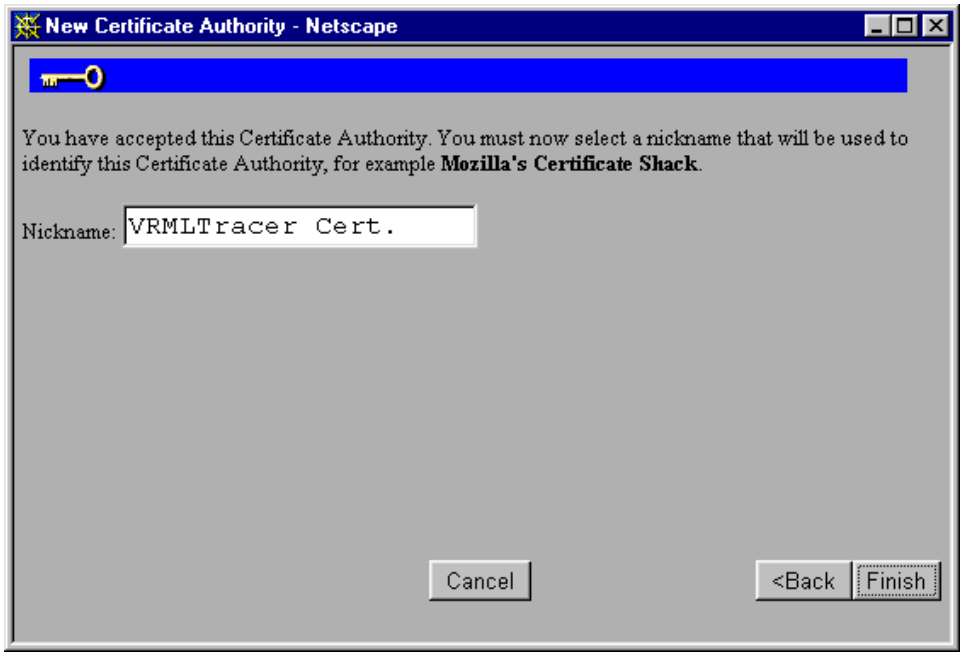


Figure B.2 - "Certificate Nickname" Dialog
The user enters a nickname by which the digital certificate will be recognized. In this example, the certificate is nicknamed "VRMLTracer Cert."

Step 2.4 –

Once a nickname is given to the certificate, the user can **complete the import by pressing "Finish"**. If the steps up to this point have been completed successfully, then the browser is ready to view the applet, and **Step 3 below can be ignored**. It is also safe to delete the copy of the certificate that the user saved to the local computer because Netscape keeps an internal copy itself.

Step 3 – Edit Netscape's preferences to add the MIME type "application/x-x509-ca-cert".

Step 3.1 –

If Netscape does not recognize the cacert file as a certificate, then it must be made to interpret the extension as such. First, **select "Preferences" from the "Edit" menu.**

Step 3.2 –

Next, **double click on "Navigator"** in the column on the left side of the dialog box.

Step 3.3 –

Highlight "Applications" under this heading.

Step 3.4 –

Now, on the right side of the dialog, **click the "New Type" button**.

Step 3.5 –

Under **"Description of type**", the user may write anything, but it should be something recognizable, like **"x509 Digital Certificate"**.

Step 3.6 –

For **"File extension**", the user must enter **"cacert"**.

Step 3.7 –

Then for **"Mime type"** the following must be entered, **"application/x-x509-ca-cert"**.

Step 3.8 –

The user must now fill in **"Application to use"**. The **application must be the Netscape executable**, and may be **selected by pressing "Browse" and navigating to the correct file**. The entire path to the executable should be enclosed in quotes, and should be followed by a space, then "%1" (percent one). See Figure B.3 for an idea of how the "New Type" dialog should appear.

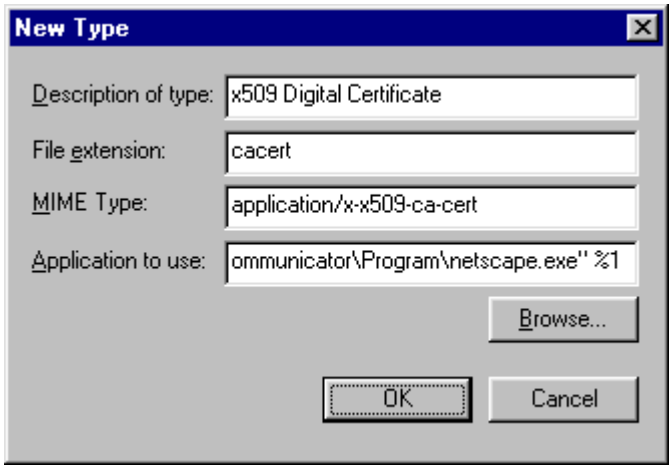


Figure B.3 - "New Type" Dialog
Here is the dialog through which the user allows Netscape to recognize the digital certificate file, filled in as needed.

Step 3.9 –

The user can finally **press the “Ok” button** at the bottom of the dialog and must then **proceed back to Step 2 above**.

Appendix C – HTML Tags

The applet is viewed through a web browser, and is therefore embedded in a web page. Typically, the way to do this is to use the HTML APPLET tag. A simple tag containing the applet might look as follows:

```
<APPLET     archive="VRMLTracer.jar"
            code="vrmltracer/VRMLTracerFrame.class"
            width=700
            height=800>
</APPLET>
```

The “width” and “height” variables determine the size of the applet on the web page. The “archive” variable provides the name of the JAR file where the applet’s class files are stored. The browser knows which class file contained within the archive to load, because it is specified by the “code” variable.

It is possible to pass arguments to an applet through the use of PARAM HTML tags within the APPLET tag. For this applet, optional PARAM tags provide a default VRML scene file to display when the applet is first loaded and a list of other sample VRML files to be loaded in the applet's history dropdown box. The applet is initialized with the default VRML file bound to the "defaultURL" variable. This variable holds the URL of the file to load when the applet begins execution. If no default URL is specified, the applet will not load a VRML scene when it is first loaded. VRML files are loaded in the history dropdown through variables of the type "sampleURL*", where the * is a placeholder for numbers greater than or equal to zero. The sample URLs must be numbered consecutively beginning with zero, and they will be loaded into the history list in numerical order. When no sample URLs are given, the history dropdown will initially be empty.

As an example, consider the following applet tag:

```
<APPLET     archive="VRMLTracer.jar"
            code="vrmltracer/VRMLTracerFrame.class"
            width=700
            height=800>
<PARAM name="defaultURL"
      value="http://www.graphics.lcs.mit.edu/vrmlTracer/prims.wrl">
<PARAM name="sampleURL0"
      value="http://www.graphics.lcs.mit.edu/vrmlTracer/box.wrl">
<PARAM name="sampleURL1"
      value="http://www.graphics.lcs.mit.edu/vrmlTracer/prims.wrl">
<PARAM name="sampleURL2"
```

```
      value="http://www.graphics.lcs.mit.edu/vrmlTracer/cylinder.wrl">
</APPLET>
```

This tag loads the applet as in the previous example, but with the addition of some initialization information. The applet starts with the “prims.wrl” file displayed. It also has three files already listed in its history dropdown. The files “box.wrl”, “prims.wrl”, and “cylinder.wrl” are preloaded into the history list in that exact order, reading from top to bottom. So a user unfamiliar with the applet is treated to a scene being displayed when the applet begins, and has a choice of selected other scenes to sample simply by opening the history menu and highlighting one of the URLs listed there.

Appendix D – Supported VRML Nodes

Figure D.1 shows a list of VRML nodes supported by the ray tracing and polygon renderers, as indicated. It does not include all possible VRML nodes.

Node	Ray tracing renderer	Polygon renderer
Appearance	Y	Y
Billboard		Y
Box	Y	Y
Cone	Y	Y
Cylinder	Y	Y
DirectionalLight	Y	Y
ElevationGrid	Y	Y

Figure D.1 – Supported VRML Nodes
This list of VRML nodes shows which nodes are supported by one or both renderers. Only those nodes handled by one or both renderers are listed.
(continued on next page)

Node	Ray tracing renderer	Polygon renderer
Extrusion	Y	Y
Group	Y	Y
IndexedFaceSet	Y	Y
IndexedLineSet		Y
Inline	Y	Y
LOD	Y	Y
Material	Y	Y
PointLight	Y	Y
PointSet		Y
Shape	Y	Y
Sphere	Y	Y
SpotLight		Y
Switch	Y	Y
Transform	Y	Y

Figure D.1 continued – Supported VRML Nodes

Appendix E – Online Material

The work presented here is grounded in the notion of online accessibility. The applet, along with complete source code and this document, can be found at http://web.mit.edu/~bglazer/www/index.html. This work can also be accessed through the MIT Computer Graphics Group web site, at http://www.graphics.lcs.mit.edu/vrmlTracer/index.html.

References

[1] Armstrong, Matt, and Ma, Yi, Java Ray Tracer, http://robotics.eecs.berkeley.edu/~mayi/CS184/, 1997.

[2] Bentley, Jon L., "Multidimensional Binary Search Trees Used for Associative Searching," Communications of the ACM, Vol. 19, pp. 509-517, 1975.

[3] Carey, Rikk, Bell, Gavin, and Marrin, Chris, "Virtual Reality Modeling Language, (VRML97)," ISO/IEC 14772-1:1997, Copyright 1997 The VRML Consortium Incorporated, 1997.

[4] Clark, James H., "Hierarchical Geometric Models for Visible Surface Algorithms," Communications of the ACM, 19(10), pp. 547-554, October 1976.

[5] Couch, Justin, JVerge, http://www.vlc.com.au/JVerge/, Copyright Justin Couch, The Virtual Light Company.

[6] Descartes, Alligator, "Interfacing Java and VRML," UK Java Developer's Conference paper, Copyright 1996 Hermetica, November 1996.

[7] Fussell, Donald S., and Subramanian, K. R., "Fast Ray Tracing Using K-d Trees," Technical Report, University of Texas, Austin, Number CS-TR-88-07, March 1, 1988.

[8] Glassner, Andrew S., "Space Subdivision For Fast Ray Tracing," IEEE Computer Graphics and Applications, 4(10), pp. 15-22, October 1984.

[9] Java Capabilities API, http://developer1.netscape.com:80/docs/manuals/signedobj/capsapi.html, Copyright 1998 Netscape Communications Corporation, 1998.

[10] Matthews, Mark, JGL - An OpenGl-like Graphics Class Library for Java, http://www.ccm.ecn.purdue.edu/~mmatthew/java/, Copyright 1997 Mark Matthews, Purdue University CADLAB, 1997.

[11] McMillan, Leonard, Instructional Ray-Tracing Renderer, for UNC COMP 136, http://graphics.lcs.mit.edu/~mcmillan/comp136/Project5/RayTrace.java, Fall 1996.

[12] Muuss, Michael John, "RT & REMRT: Shared Memory Parallel and Network Distributed Ray-tracing Programs," Proceedings of the 4th Computer Graphics Workshop (CGW87) 86-98, 1987.

[13] Muuss, Michael John, “Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models,” Proceedings of BRL-CAD Symposium '95, Aberdeen Proving Ground, MD, 5-9 June 1995.

[14] Netscape Signing Tool 1.1, http://developer1.netscape.com:80/software/signedobj/jarpack.html, Copyright 1998 Netscape Communications Corporation, 1998.

[15] Reinhard, Erik, and Kok, Arjan J. F., and Jansen, Frederik W., “Cost Prediction in Ray Tracing,” Eurographics Rendering Workshop 1996, pp. 41-50, Springer Wein, June 1996.

[16] Scherson, Isaac D., and Caspary, Elisha, “Data Structures and the Time Complexity of Ray Tracing,” The Visual Computer, 3(4), pp. 201-13, December 1987.

[17] Subramanian, K. R., “Adapting Search Structures to Scene Characteristics for Ray Tracing," Ph.D. Dissertation, Department of Computer Science, The University of Texas at Austin, December 1990.

[18] Subramanian, K. R., and Fussell, Donald, “A Cost Model for Ray Tracing Hierarchies,” Technical Report, Dept. of Computer Sciences, Univ. of Texas at Austin, Number, TR-90-04, March 1990.

[19] Subramanian, K. R., and Fussell, Donald S., “Factors Affecting Performance of Ray Tracing Hierarchies,” Technical Report, University of Texas, Austin, Number CS-TR-90-21, July 1, 1990.

[20] Sun Microsystems, Inc., “Java 3D API Specification Version 1.1 Alpha 02,” Copyright 1998 Sun Microsystems, Inc., April 1998.

[21] Teller, Seth, and Alex, John, “Frustum Casting for Progressive, Interactive Rendering,” Technical Report, Laboratory for Computer Science, Massachusetts Institute of Technology, Number, TR-740, January 1998.

[22] Thompson, Kelvin, and Fussell, Donald, “Time Costs of Ray Tracing with Amalgams,” Technical Report, Dept. of Computer Sciences, Univ. of Texas at Austin, Number, TR-91-01, January 1991.

[23] VRwave, http://www.iicm.edu/vrwave, Copyright 1997 by the Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology, Austria, 1997.