Reality is not enough any more....

ACKNOWLEDGEMENTS

I take this opportunity to express my sincere thanks and deep gratitude to Dr Santanu Chaudhury, my project guide, for his help and invaluable guidance during the course of my B.Tech. project. This project would not have been the same without his vision and guidance.

I must also thank Ms Nandini Srivastava, Senior Scientific Officer, Department of Computer Science and Engineering, for allowing me access to the Internet, and for making my source code available on the Web. I would also like to thank Mrs Shashi at the Computer Centre Library, IIT Delhi, for giving me access to the library, and for being there whenever I needed help. And finally I express my gratitude to all those who were a source of encouragement and motivation.

Amit Goel
Indian Institute of Technology, Delhi
May 1997

CERTIFICATE

This is to certify that the thesis *Web-based Virtual Reality Simulation* submitted by Amit Goel (93133) in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Electrical Engineering at the Indian Institute of Technology, Delhi, is a record of the student's bonafide work carried out by him under my supervision and guidance during the period July 1996 to May 1997.

The results and contents embodied in this thesis have not been submitted to any other university or institute for the award of any other degree or diploma. I hereby approve this thesis for submission to the Institute towards the award of the B.Tech. degree.

Dr Santanu Chaudhury
Department of Electrical Engineering
IIT Delhi

ABSTRACT

This thesis describes Web3D, a browser capable of visualizing virtual 3D scenes on the Internet. Web3D consists of a new interpreted language which is based on a few simple, yet powerful constructs that allow programmers to describe three-dimensional scenes and animations in a non-immersive virtual world. 3D scenes are modelled using the four 3D geometric primitives -- cubes, cones, cylinders, and spheres; light sources; and an observer placed at a given location and looking in a given direction in space. Visual realism is enhanced by shading and texturing. Besides just describing graphical objects and their attributes, Web3D is capable of embedding these objects within text documents, and linking these documents by means of hypertexts. Hyperlinks can be to other virtual worlds, or to HTML documents, in which case Web3D invokes a standard WWW browser. Web3D, too, can be invoked by a properly configured Web browser.

Web3D has been developed in C programming language. It runs on UNIX, and requires X Window Motif user interface toolkit. Web3D source code is freely available for non-commercial use, providing a platform for research and experiment.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

The Internet, as we all know, is a highway of information, and browsers like Netscape and Mosaic give us a platform from which to access this information. Existing World Wide Web (WWW) browsers are based on the HyperText Markup Language (HTML), a language that is capable of presenting information in the form of plain 2D text, images, and hyperlinks. For years, HTML has been the "language of the Web", but as HTML became popular, the need was felt for a new language that could be used to specify 3D scene descriptions and WWW hyperlinks -- an analog of HTML for virtual reality. In 1994, the Virtual Reality Modeling Language (VRML) was conceived as a file format for describing 3D interactive scenes and objects. It could also be used to create three-dimensional representations of complex scenes such as illustrations, product definitions, and virtual reality presentations; and it could be used in conjunction with the World Wide Web.

Early on, the designers decided that VRML would not be an extension to HTML. HTML was designed for text, not graphics. Also, VRML requires even more finely tuned network optimizations than HTML; it was expected that a typical VRML scene would be composed of many more "inline" objects and served up by many more servers than a typical HTML document.

Moreover, HTML was an accepted standard. To impede the HTML design process with VRML issues and constrain the VRML design process with HTML compatibility concerns would be to do both languages a disservice. As a network language, VRML would succeed or fail independent of HTML. It was also decided that, except for the hyperlinking feature, the first version of VRML will not support interactive behaviours. This was a practical decision intended to streamline design and implementation. The first release of VRML (VRML 1.0) featured only static worlds hyperlinked with the World Wide Web. VRML 1.0 parsers designed in C/C++ were made available to the public. Our task was to take up the source code of one of these public-domain parsers, and extend it to include specifications for various kinds of animations. A "browser" was also required to be implemented, that would be the interface for drawing 3D graphics on the screen.

However, the process of understanding third-party code turned out to be more difficult than we had envisaged. After struggling with the VRML 1.0 parser code for some time, we decided to quickly develop a parser ourselves, for a language which had a syntax similar to that of VRML 1.0, which would have the basic constructs required to describe 3D objects on the graphics screen. Then we could go ahead and implement constructs for various kinds of animations.

Design Criteria

Our browser was designed to meet the following requirements:

- Platform independence
- Extensibility
- Ability to work well over low-bandwidth connections

Platform independence: Platform independence essentially implies that one can browse our script files on any platform as long as one has the interpreter (browser) for that

platform. At IIT, we were able to successfully implement our browser for three platforms available to us: the Sun SPARCstations, Silicon Graphics Indy, and the DRS6000. We hope that our source code compiles successfully on other systems not available to us at IIT.

Extensibility: Extensions to our language must be easy to implement, so that a user can make use of constructs that are not a standard part of our language by declaring their prototypes within the script file. These constructs must be recognised by our interpreter as being external, and must be interpreted based on their prototype. However, as implemented, extensions to our language require a little effort from the part of the user. In order to add constructs to our language, a user needs to modify the source code slightly and then recompile. The changes required are minimal, and our technique almost meets the design criteria.

Ability to work well over low-bandwidth connections: Typically, our browser application should retrieve the source code written in the 3D-scene-description language over a network, interpret this source code locally and display the 3D output by the brute power of the processor at its own end. This means that only the source code for describing 3D scenes needs to be transferred over the network, and since this source is plain text, it is far less bulky than graphical bitstreams. Besides, plain text transfers also lend themselves to greater compression, which network protocols usually take advantage of. So, having to transfer lesser data over a network for browsing 3D worlds was a major design motivation for our browser.

Related Work

Design of Web3D was inspired by the ongoing research on VRML throughout the world. This research is still in its nascent stage, and we expect to be one of those involved with this

evolving new technology of virtual reality on the Web. Here we present an overview of some of the VRML browsers currently available. We use the term “browser” to refer to software which itself retrieves files across the Internet, and “viewer” for software which relies on another supporting application to perform file retrievals.

WebSpace from Silicon 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 the walk viewer). WebSpace binaries are freely available with a supported commercial version.

WebView from San Diego Supercomputer Center is a publicly available VRML browser for SGI systems, available as source code based on the Inventor library. WebView has four viewing styles (examiner, fly, plane, walk) and an integrated editing facility. It is intended as a public development and test platform, but is limited to SGI platforms under UNIX.

WorldView from InterVista Software is targeted to empower standard PCs for real-time applications. All network communications is built into this standalone browser, which does not rely on a cooperating Web browser. WorldView is available for all Windows platforms.

WebFX from Paper Software is a VRML browser for the Windows environment. It incorporates IRC 3D chatting and physically-based navigation metaphors (including collision detection) as well as VRML authoring facilities.

i3D from the Center for Advanced Studie, Research and Development in Sardinia exploits the rendering capabilities of high-end machines. This VRML viewer includes a preprocessing optimisation phase and time-dependent rendering facilities to guarantee constant frame rates even for very large scenes.

WebOOGL 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.

VRweb, a joint project between IICM, NCSA, and the University of Minnesota, is designed to work with multiple information systems, namely WWW, Hyper-G, and Gopher, as well as on a variety of platforms. 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.

CHAPTER 2

LANGUAGE SPECIFICATION

The language for describing 3D scenes has been developed. It is an interpreted language, and the interpreter has been written in C for UNIX. This language aims at minimising the amount of code a user needs to write in order to describe 3D scenes in space.

Any 3D scene is composed of objects. Each such object is called a "node". A node has the following characteristics:

- What kind of object it is. A node might be a cube, a sphere, a camera, a light source.

- The parameters that distinguish this node from other nodes of the same type. For example, each Sphere node might have a different radius, and different surface properties. These parameters are called "fields". A node can have 0 or more fields.

2.1 General Syntax

The syntax chosen to represent these pieces of information is straightforward -- node names must not begin with a digit, and must not contain spaces or control characters, single or double quote characters, backslashes, curly braces, the plus character or the period.

The '#' character begins a comment -- all characters until the next newline or carriage return are ignored. The only exception to this is within string fields, where the '#' character will be part of the string.

Blanks, tabs, newlines and carriage returns are whitespace characters wherever they appear outside of string fields.

2.2 Coordinate System

Web3D assumes a cartesian, right-handed, three-dimensional coordinate system. By default, objects are projected onto a two-dimensional device by projecting them in the direction of the positive Z axis, with the positive X axis to the right and the positive Y axis up. A camera or modelling transformation may be used to alter this default projection.

The standard unit for lengths and distances specified is meters. The standard unit for angles is degrees.

2.3 Fields

There are two general classes of fields -- fields that contain a single value (where a value may be a number or a string),

and fields that contain multiple values (three components of a vector, three components of colour, etc.). Each field type defines the format for the values it writes. Multiple-valued fields are written as a series of values separated by whitespace or commas. No enclosing brackets are required for these parameters, and no field can have zero number of parameters.

Fields can be of various kinds:

Boolean - A field containing a single boolean (true or false) value. Booleans may be written as 0 (representing FALSE), or 1 (representing TRUE).

Color - A triple-value field containing a colour. Colours are written to file as an RGB triple of floating point numbers in standard scientific notation, in the range 0.0 to 1.0.

String - A field containing an ASCII string (sequence of characters). Strings are written to file as a sequence of ASCII characters in double quotes. Any characters (including newlines) may appear within the quotes. To include a double quote character within the string, precede it with a backslash.

Vector - Field containing a three-dimensional vector. Vectors are written to file as three floating point values separated by whitespace or commas.

2.4 Nodes

Web3D's scripting language defines several different classes of nodes. Most of the nodes can be classified into one of two categories -- shape or property. Shape nodes define the geometry in the scene (cubes, cones, spheres, cylinders). Conceptually, they are the only nodes that draw anything.

Property nodes affect the way shapes are drawn. Nodes may contain zero or more fields. Each node type defines the type, name, and default value for each of its fields. The default value for the field is used if a value for the field is not specified in the source file. The order in which the fields of a node are read is not important; for example, “Cube { width 2 height 4 depth 6 }” and “Cube { height 4 depth 6 width 2 }” are equivalent.

Here are the nodes grouped by type. The first group are the shape nodes. These specify geometry:

AsciiText, **Cone**, **Cube**, **Cylinder**, **Sphere**

The second group are the properties. These can be further grouped into properties of the geometry and its appearance, cameras and lights, and any other constructs that affect the visualization environment.

Animation, **DirectionalLight**, **Fog**, **Material**, **OrthographicCamera**, **PerspectiveCamera**, **PointLight**

The **Separator** node is used to group objects together.

2.5 Description of Nodes

Shape nodes

AsciiText - This node represents strings of text characters from the ASCII coded character set. The first string is rendered with its baseline at the top left corner of the browser's view window. Text is rendered from left to right, top to bottom in the font set by “font”. Since the rendering of text is only two-dimensional, the z-coordinate of

“position” is ignored. The z-component has been maintained to allow scope for implementing 3D text some time in the future.

SYNTAX/DEFAULTS

```
AsciiText
{     font  1          # number of font. Total fonts=4
      size  1          # maximum size=6
      style  1         # 1=regular; 2=italicized
      weight  1        # 1=medium; 2=bold
      color  1 1 1     # RGB triplet
      intensity  1.0   # maximum value 1
      position  0 0 0  # pixel position on screen
      string  “Web3D”  # string to be displayed
}
```

The “string” property of AsciiText needs elaboration. This property is alone responsible for getting some presentable text rolling on the screen. It handles titles, various fonts and styles, and hypertexts all in one, so all the other properties that appear along with this property are near redundant. The syntax for handling the string property can be best explained by an example:

```
AsciiText
{     string “$F4$S6$BGraphics vs Animation :$B$S2$F2\nThe \
      essential difference between $Igraphics$I and \
      $Ianimation$I is the addition of the temporal dimension.\
      $VGraphics <graphics.wrl> is the modelling of objects, \
      whereas $Vanimation <animation.wrl> adds temporal \
      information to these objects.”
}
```

Any text appearing between two "$B"'s is written out in bold, and between two "$I"'s is written out in italics. "$Fn" is used to change the number of font to be used for ensuing text, whereas "$Sn" changes the size of ensuing text. The "$V" is used to define hypertexts. For example, in the above case, "Graphics" and "animation" are hypertexts. These will appear underlined in cyan colour, and clicking on them will bring up the files "graphics.wrl" and "animation.wrl" respectively for browsing. Line breaks are given by the "\" character, as in C. This is a special character that must be escaped in case it needs to be printed literally. For example, to print a "\" in the string, one needs to type in "\\". In the same way, "$" is a special character and needs to be escaped. So one needs to type "\$" to display the dollar literally. Same for the quote sign. However, the "#" is not a special character. It will appear literally wherever typed. Only outside strings can it be used to comment out lines.

Cone - This node represents a simple cone whose central axis is aligned with the y-axis. By default, the bottom face of the cone is centred at (0,0,0) and has a size of 200 in all three directions. The cone has a radius of 100 at the bottom and a height of 200, with its apex at 200 and its bottom at 0. The cone has two parts: the sides and the bottom. The orientation of the cone in space is determined by the "tilt" field. This field takes as parameters three floating point values representing tilt about the three coordinate axes.

SYNTAX/DEFAULTS

```
Cone
{     bottomRadius  100     # radius of the flat face
      bottomCentre  0 0 0   # centre of bottomface in space
      height  100           # height of cone
      tilt  0 0 0           # tilt about x,y,z axes in degrees
}
```

Cube - This node represents a cuboid aligned with the coordinate axes. By default, the cube is centred at (0,0,0) and measures 200 units in each dimension, from -100 to +100.

SYNTAX/DEFAULTS

```
Cube
{     width  100
      height  100
      depth  100
      centre  0 0 0    # geometric centre of cube in space
      color  1 1 1     # RGB triplet
      tilt  0 0 0      # tilt about x,y,z axes in degrees
}
```

Cylinder - This node represents a simple capped cylinder centred around the y-axis. By default, the cylinder is centred at (0,0,0) and has a default size of -100 to +100 in all three dimensions. The cylinder has three parts: the sides, the top (y = +100) and the bottom (y = -100). One can use the radius and height fields to create a cylinder with a different size.

SYNTAX/DEFAULTS

```
Cylinder
{     radius  100
      height  200
      centre  0 0 0    # geometric centre of cylinder
      color   1 1 1    # RGB triplet
      tilt  0 0 0      # tilt about x,y,z axes in degrees
}
```

Sphere - This node represents a sphere. By default, the sphere is centred at the origin and has a radius of 100.

SYNTAX/DEFAULTS

```
Sphere
{     radius  100
      centre  0 0 0
      color  0.6 0.6 1
      tilt  0 0 0      # tilt about x,y,z axes in degrees
}
```

Property nodes

Animation - This node is used to set various parameters relating to the animation of 3D objects in space. For example, it can be used to set the duration (in seconds) for which animation proceeds, and to set the velocity of objects in the 3D scene.

SYNTAX/DEFAULTS

```
Animation
{     duration  30          # default=15 secs
      velocity  1.0         # min=0; max=1
}
```

DirectionalLight - This node defines a directional light source of constant intensity, that illuminates along rays parallel to a given three-dimensional vector.

SYNTAX/DEFAULTS

```
DirectionalLight
{     on  1                  # for TRUE
      direction  0, 0, -1    # Vector
      intensity  1.0         # min=0   max=1
      color  1 1 1           # RGB triplet
}
```

Fog - This node allows users to simulate fog and haze in the virtual scene. It allows one to model 3D scenes with greater visual realism. Users can define the intensity of fog in a 3D scene by setting certain parameters belonging to this node. For example, users can define the visible distance in a scene, so that the lesser this distance, the hazier the scene becomes. Users can even define the colour of the fog, whose default is grey (0.6,0.6,0.6). Objects in a scene typically assume the colour of the fog as they go farther from the camera. This is quite a crude simulation of fog & haze as given in Graphics Gems II. Better implementations may be expected in future releases of Web3D.

SYNTAX/DEFAULTS

```
Fog
{     visibleDistance  2000
      color  0.6 0.6 0.6
}
```

Material - This node defines surface material properties. All objects appearing after this node in the source code will inherit the properties set by this node.

SYNTAX/DEFAULTS

```
Material
{     gloss  1              # 0=dull; 1=glossy
      reflectiveness  1 1 1 # RGB triplet
      shininess  1 1 1      # RGB triplet
}
```

OrthographicCamera - An orthographic camera defines a parallel projection from a viewpoint. This camera does not diminish objects with distance, as a perspective camera does. The viewing volume for an orthographic camera is a rectangular parallelepiped (a box). A camera can be placed in a Web3D world to specify the initial location of the observer when that world is entered. Web3D will typically modify the camera to allow a user to move through the virtual world interactively.

SYNTAX/DEFAULTS

```
OrthographicCamera
{     position  0 0 1000          # in space
      orientation  0, 0, -1       # direction of view
}
```

PerspectiveCamera - A perspective camera defines a perspective projection from a viewpoint. The viewing volume for a perspective camera is a truncated right pyramid. By default, the camera is located at (0,0,1) and looks along the negative z-axis; the position and orientation fields can be used to change these values. The focal distance of the camera is typically the distance between the centre of projection and

the view plane. So greater this distance, larger the object appears.

SYNTAX/DEFAULTS

```
PerspectiveCamera
{     focalDistance  300
      position  0 0 1000          # in space
      orientation  0, 0, -1       # direction of view
}
```

PointLight - This node defines a point light source at a fixed location in space. A point source illuminates equally in all directions; that is, it is omni-directional.

SYNTAX/DEFAULTS

```
PointLight
{     on  1                 # 0=off; 1=on
      intensity  1          # min=0; max=1
      location  0 0 1000    # in space
}
```

2.6 An Example

```
# Code for describing a red sphere revolving in circular orbit
# about the y-axis, as viewed by an approaching camera. The
# scene is lit by a parallel light source throwing light
# parallel to vector (1,0,-1) in three-dimensional space


Separator {
# light the scene
DirectionalLight
{     on  1                 # light source is switched on
      direction  1, 0, -1   # shining from viewer into scene
}

# set up the camera
PerspectiveCamera
{     focalDistance  200
      position  0 0 800-50t # approaching along -z direction
      orientation  0, 0, -1
}

# set up animation parameters
Animation
{     duration  30     # seconds
      velocity  1      # min=0; max=1
}

# render a sphere
Sphere
{     radius  100
      centre  300cos(t) 0 300sin(t)    # revolving about y-axis
      color  1 0 0                     # red sphere
}

}  #end
```

CHAPTER 3

THE INTERPRETER

The interpreter, developed in C for UNIX, has two major components – the *lexical analyser* and the *parser*.

3.1 The Lexical Analyser

The lexical analyser is responsible for checking for valid "tokens" in the source file that needs to be interpreted. A token is the smallest recognised string of characters in the language specifications. For example, a "{" is a recognised token. Similarly, all objects such as "Separator", "Cone", "radius", etc. are tokens. So if a "Cobe" appears in the input source file in place of a "Cone", it will be rejected by the lexical analyser, and the program will report an error message.

Naturally, for the purpose of validating and invalidating tokens, the system requires a sort of dictionary for reference. This dictionary is called the "symbol table", and

for our purposes, it is implemented as arrays of strings, one containing the list of valid nodes, and the other containing the list of valid fields, as shown below:

```
char nodes[MAX_NODES][20] =

{     "Separator", "AmbientLight", "Animation", "AsciiText",
      "Cone", "Cube", "Cylinder", "DirectionalLight", "Fog",
      "Material", "OrthographicCamera", "PerspectiveCamera",
      "PointLight", "Sphere", "SpotLight", "TextProperties"
};

char fields[MAX_FIELDS][20] =

{     "bottomCentre", "bottomRadius", "centre", "color",
      "depth", "direction", "duration", "focalDistance",
      "font", "gloss", "height", "intensity", "location", "on",
      "orientation", "position", "radius", "reflectiveness",
      "roughness", "shininess", "size", "solid", "string",
      "style", "tilt", "velocity", "visibleDistance", "weight",
      "width"
};
```

The tokens are case-sensitive. For example, whereas *Sphere* is a valid token, *sphere* is not. Each token has a unique integer value associated with it. Since integer comparison takes much less time as compared to string comparisons, this method is preferred. For the purpose of associating integer values with tokens, we define "macros" for each of the tokens. These are placed in a separate header file called "tokens.h". A section of the file is reproduced below:

tokens.h:

```
/* Single character tokens */

                          /* token value for    */
#define ERROR        -1   /* unrecognised input */
#define EOI           0   /* end of input       */
#define LB            1   /* left braces        */
#define RB            2   /* right braces       */
#define LP            3   /* left parenthesis   */
#define RP            4   /* right parenthesis  */
#define NUMBER        5   /* real numbers       */
#define TEXT          6   /* text strings       */
#define PLUS          7
#define MINUS         8
#define MULTIPLY      9
#define DIVIDE       10
#define EXP          11
#define COS          12
#define SIN          13
#define              14   /* time variable 't'  */
#define COMMA        15
#define EXPRESSION   16

/* Macros for objects (nodes) */

#define SEPARATOR          17
#define AMBIENTLIGHT       18
#define ANIMATION          19
#define ASCIITEXT          20
#define CONE               21
#define CUBE               22
#define CYLINDER           23
#define DIRECTIONALLIGHT   24

.....
```

The lexical analyser translates the input stream into a sequence of tokens -- a form that is more manageable by the parser. It uses a simple, buffered input system, getting 512 bytes at a time from the input stream, and then isolating tokens one at a time. Another 512-byte buffer is fetched only when the current buffer is exhausted. The main advantage of a buffered system is speed. Computers like to read data in large chunks. Generally, the larger the chunk, the greater the throughput. This is especially so when the buffer size gets above the size of a disk cluster. Reading from unbuffered I/O needs very frequent disk accesses, which can slow the program down considerably.

Another important issue that has to do with speed is the *lookahead* and *pushback* feature. Lexical analysers often have to know what the next input character is going to be without actually reading past it. They must "look ahead" by some number of characters. Similarly, they often need to read past the end of the token in order to recognise it, and then "push back" the unnecessary characters onto the input stream. Consequently, there are often extra characters that must be handled specially. The special handling is both difficult and slow when one is using single-character input. Going backwards, however, is simply a matter of moving a pointer.

The Finite State Machine for lexical analysis

Figure 1 shows a part of the transition diagram for our Finite State Machine (FSM) that recognises the tokens as valid sequences of characters:

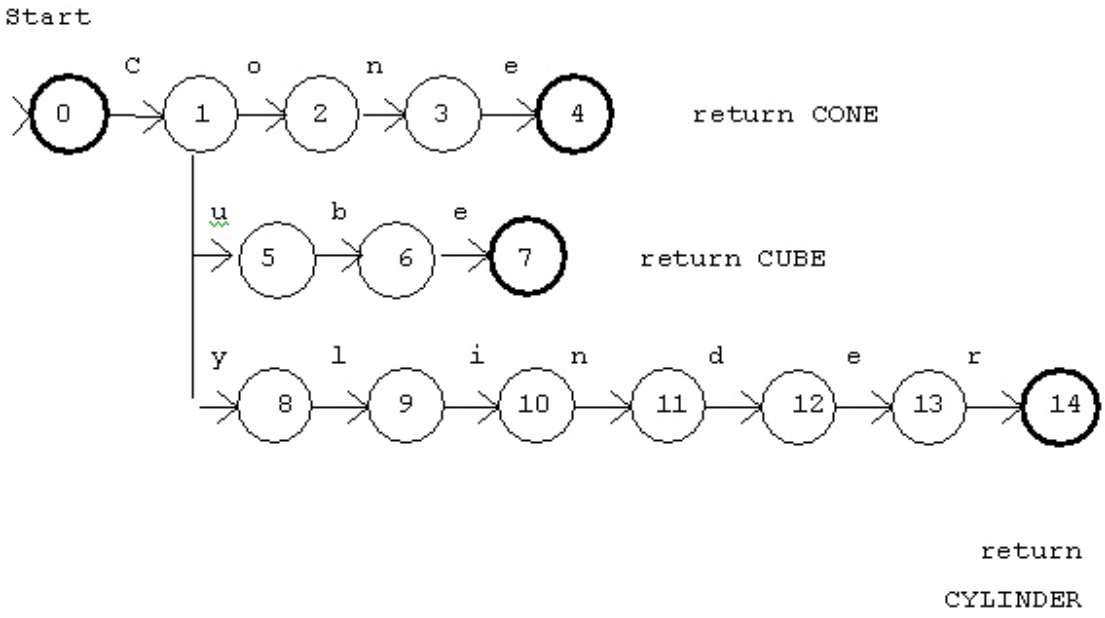

Figure 1 Transition diagram of FSM (in part)

The circles are individual states marked with the state number, which can be any arbitrary number that identifies the state. State 0 is the start state, and the machine is initially in this state. From the start state, reading a 'C' from input causes a transition to state 1. From state 1, an 'o' gets the machine to state 2, and a 'u' gets the machine to state '5', and so on. The self-loops around the accepting states indicate the action associated with the acceptance of the token. This is the integer value to be returned to the parser.

3.2 The Parser

As the name specifies, this function "parses" the input stream, i.e. it scans the input stream character by character. It calls the lexical analyser repeatedly to verify if strings scanned are valid tokens. Then the parser checks if the sequence of these tokens is according to the rules of the language. These rules are defined by a grammar, which

specifies ALL possible and valid sequences of tokens in a source file to produce a meaningful program.

For example,

```
Cube { radius  1 }
```

contains all valid tokens, but even then the property radius makes no sense for a cube. It is the work of the parser to report this error as soon as it comes across the radius property for a Cube.

The parser generates no code, it just parses the input. The implementation is in the form of a “recursive-descent parser”, i.e. it parses the input stream with the help of a set of highly recursive functions. For example, the production:

```
Expression  →  Expression + Expression
```

is implemented by the following subroutine:

```
void expression()
{
      advance();       /* read the next token in the stream */
      if (operator())  /* check if token is an operator */
      {
            advance(); /* advance to the next token */
            expression();
      }
      else
            error();   /* report a parse error */
}

int operator()
{
      return     ( ((match(PLUS)) || (match(MINUS)) ||
                 (match(MULTIPLY)) || match(DIVIDE)) ||
                 (match(EXP)) );
}
```

The function match() returns a boolean value. If the token passed as argument is matched, it returns “true”, otherwise it returns “false”.

3.3 The Grammar

Web3D's 3D-scene-description language is based on the following grammar:

```
Separator  →   Node { Parameters } | Separator

Node       →   AsciiText | Cube | Cone | Sphere | ...

Parameters →   Field  Value | Field  Expression | Parameters

Expression →   Expression + Expression |
               Expression - Expression |
               Expression * Expression |
               Expression / Expression |
               Expression ^ Expression |
               sin(Expression) |
               cos(Expression) |
               (Expression) |
               Value |
               t

Field      →   radius | height | width | color | ...

Value      →   Number | -Number | Number.Number

Number     →   Digit | Number

Digit      →   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The parser is a straightforward implementation of this grammar.

CHAPTER 4

MODELLING THREE-DIMENSIONAL SPACE

In order to describe a scene consisting of a set of geometrical objects placed in particular positions and orientations in three-dimensional space, we define an arbitrary but fixed coordinate system for three-dimensional space -- this we call the ABSOLUTE system. Next, the coordinates of the vertices of a particular object are defined in some simple way, usually about the origin of the ABSOLUTE system. This we call the SETUP position for that object. Polygonal facets within the object are defined by specifying, and giving the order of, the vertices forming their corners.

Each particular object must be moved from its SETUP position to the desired position in space, its ACTUAL position. The matrix which relates the SETUP and ACTUAL positions for a given object may be calculated using one, or a combination, of the transformations described above. The object is moved by pre-multiplying the column vector form of each of its defining vertices with the same SETUP to ACTUAL matrix. The vertex coordinates are still specified with respect to the ABSOLUTE system, whether they are in SETUP or ACTUAL position. Facet

relationships such as co-planarity and the order of vertices, are preserved with the transformed vertices. Naturally, different objects will have unique SETUP to ACTUAL matrices.

In order to illustrate further our algorithm, we will start our description by using a single cube. The SETUP vertex coordinates of the cube are defined to be the 8 vertex triples (1,1,1); (1,-1,1); (1,-1,-1); (1,1,-1); (-1,1,1); (-1,-1,1); (-1,-1,-1) and (-1,1,-1); numbered 1 through 8, as shown in Figure 2. The six facets are thus the set of four vertices 1,2,3,4; 1,4,8,5; 1,5,6,2; 3,7,8,4; 2,6,7,3; and 5,8,7,6. The peculiar ordering of the vertex indices in the facet definitions is to ensure that, when viewed from outside, the vertices are in anti-clockwise orientation. This is important so that we can decide which facets are visible to the observer, and which are not. If the outward-bound normal of a polygonal surface (a plane) makes an acute angle with the direction of view, the surface is visible, otherwise it is not, and must not be rendered. Matrix transformations are then used to calculate the ACTUAL position of the cube in space, relative to the ABSOLUTE system.

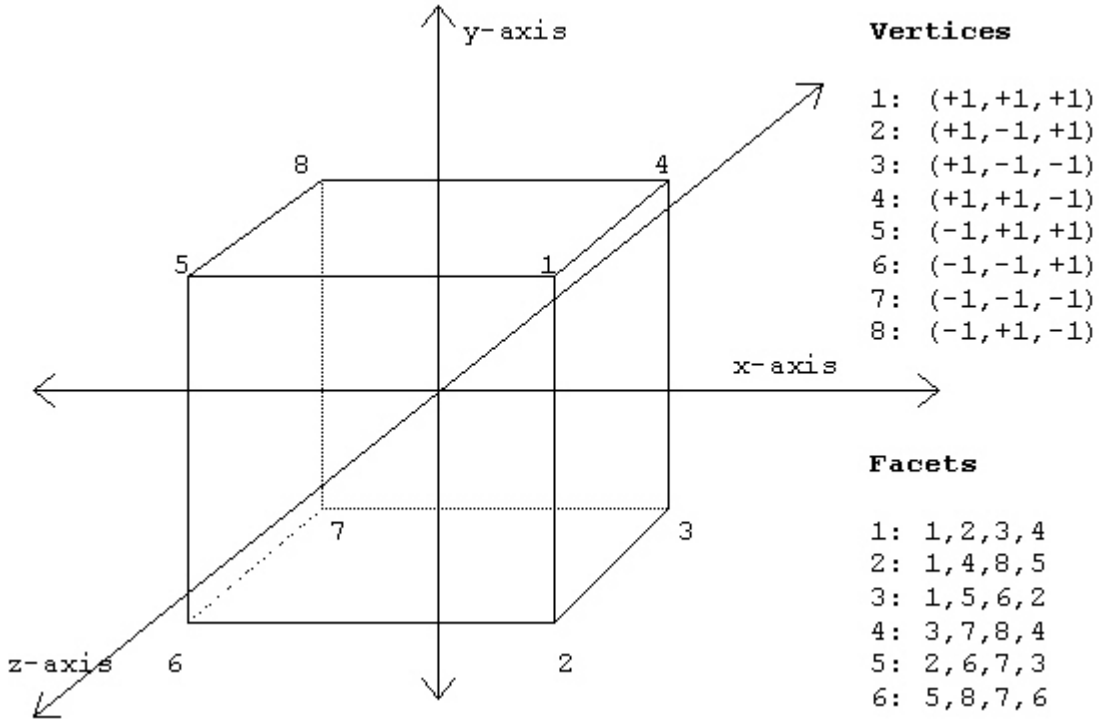

Figure 2 SETUP coordinates of a cube

4.1 The Observer

We now introduce the concept of an observer. Our eventual aim is to represent, in the graphics viewport, a three-dimensional scene as viewed by a person standing at a given position and looking in a given direction, with position and direction specified relative to the ABSOLUTE system. Imagine someone having a graphics screen firmly fixed in front of their face, and as they walk, run, jump, fly, somersault through space, they can only view that space through the screen. It is these images that we will simulate on the graphics viewport, so that the observer sitting comfortably in front of the screen can experience the same sights as our energetic "space-traveller".

We shall assume that the information about the scene (the model, the observer, and the light source) will be stored initially in terms of 3D vector coordinates in the ACTUAL position relative to the ABSOLUTE coordinate system. The eye of the observer (the camera) is placed at a position in space, defined by another 3D vector, relative to the ABSOLUTE axis looking in a fixed direction in space.

Matrix transformations are used to calculate the coordinates of the vertices relative to a new triad of axes, called the OBSERVER system, which has the camera at the origin and direction of view along the negative z-axis. These new values are called the OBSERVED position of the vertices of the object.

4.2 Orthographic Projection

A "parallel" projection is characterised by having parallel lines of projection, and is a projection under which points in three-dimensional space are projected along a fixed direction onto any plane not parallel to those lines. The *orthographic* projection is a special case whereby the lines of projection

are perpendicular to the plane. We can choose the view plane to be any plane with normal vector along the line of sight (the line of projection). This means that we can take any plane parallel to the x/y plane of the OBSERVER system, and for simplicity we choose the plane through the origin given by the equation z=0. An OBSERVED vertex is thus projected onto the view plane by the simple expedient of setting its z-coordinate to zero, and thus any two different points with OBSERVED co-ordinates (x,y,z) and (x,y,z') say (where $z \neq z'$), are projected onto the same point (x,y,0) on the view plane, and hence onto the point (x,y) in the WINDOW system.

4.3 Perspective Projection

The orthographic projection has the property that parallel lines in three-dimensional space are projected into parallel lines onto the view plane. Although they have their uses in certain scientific and architectural applications, such views do look odd. Human comprehension of spatial position is based on *perspective*. Hence our brains attempt to interpret orthographic figures as if they are perspective views. It is obviously essential to produce a projection which displays perspective phenomena -- that is, parallel lines should meet on the horizon, and an object should appear smaller as it moves away from the observer.

What is perspective vision?

To produce a perspective view, we introduce a very simple definition of what we mean by vision. We imagine every visible point in space sending out a ray which enters the eye. Naturally the eye cannot see all of space, it is limited to a cone of rays which fall on the retina, the so-called *cone of vision*, which is outlined by dashed lines of Figure 3. These rays are the lines of projection. the axis of the cone is

called the *direction of vision*. In what follows, we assume that all co-ordinates relate to the OBSERVER right-handed coordinate system, with the eye at the origin and the direction of vision identified with the negative z-axis.

We place the view plane (which we call the *perspective plane* in this special case) perpendicular to the axis of the cone of vision at a distance *d* from the eye (that is, the plane z=-*d*). In order to form the perspective projection we mark the points of intersection of each ray with this plane. Since there is an infinity of such rays, this appears to be an impossible task. Actually the problem is not that great because we need only consider the rays which emanate from the important points in the scene, in particular the corner vertices of polygonal facets. Once the projections of the vertices onto the perspective screen have been determined, the problem is reduced to that of representing the perspective plane (the view plane) on the graphics viewport. A two-dimensional coordinate system, the WINDOW system, is defined on the view plane together with a rectangular window which is identified with the viewport. The image is drawn by joining the pixels corresponding to the end-points of lines or the vertices of facets.

Calculation of the perspective projection of a point

Let the perspective plane be at a distance *d* from the eye. Consider a point **p** = (x,y,z) (with respect to the OBSERVER system) which sends a ray into the eye. We need to calculate the point of intersection, **p'** = (x',y',-*d*), where this ray cuts the view plane (the z=-*d* plane), and thus we determine the corresponding WINDOW coordinates (x',y'). First consider the value of y' by referring to Figure 3. By similar triangles we see that y'/*d* = y/|z|, that is, y' = -y*d*/z (the points in front of the eye in the OBSERVER system have negative z-coordinates).

Similarly, x' = -x*d*/z and hence **p'** = (-x*d*/z, -y*d*/z). The projection makes sense only if the point has negative z coordinate (that is, it does not lie behind the eye). We assume that the eye is positioned in such a way that this is true for all vertices.

Figure 4 illustrates the difference between orthographic and perspective projections. Perspective projections apparently diminish size and speed with depth, and so appear more natural, while orthographic views appear unrealistic and distorted.

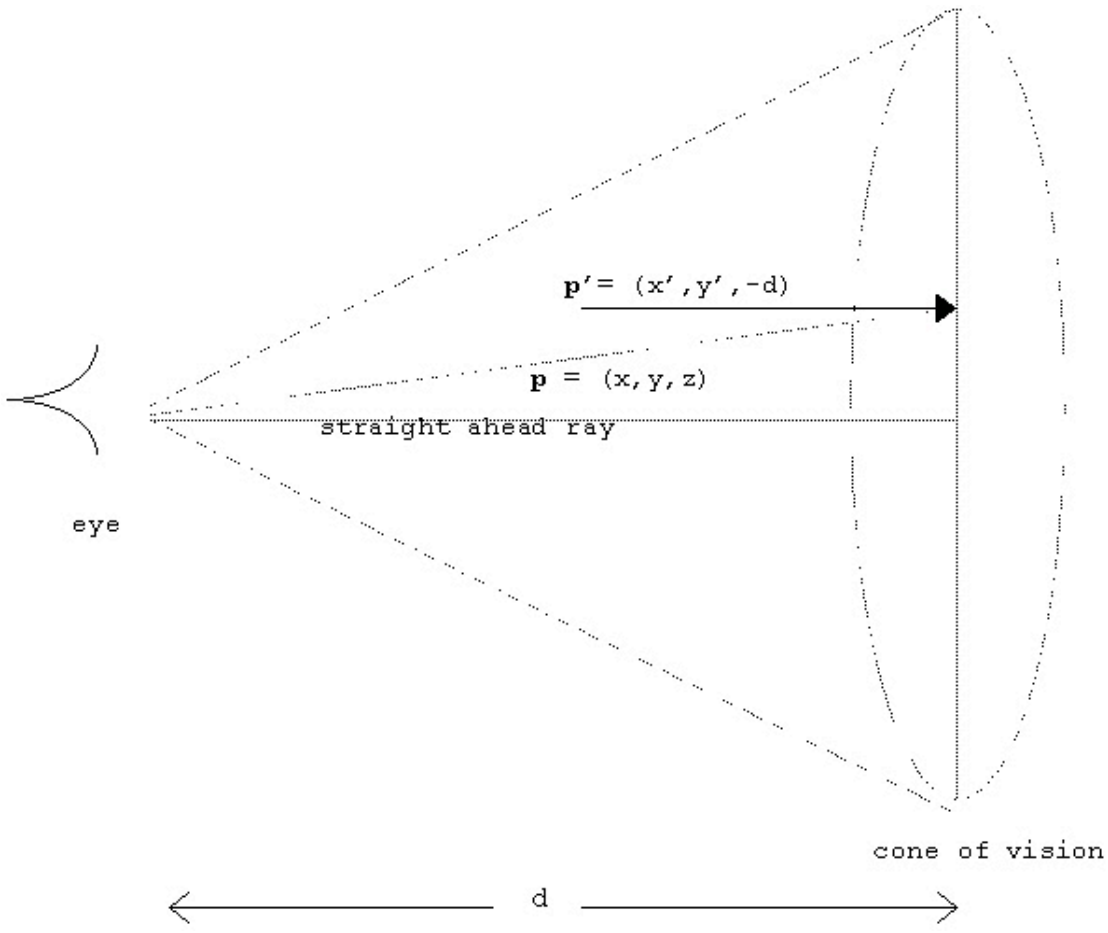

Figure 3 Perspective projection of a point

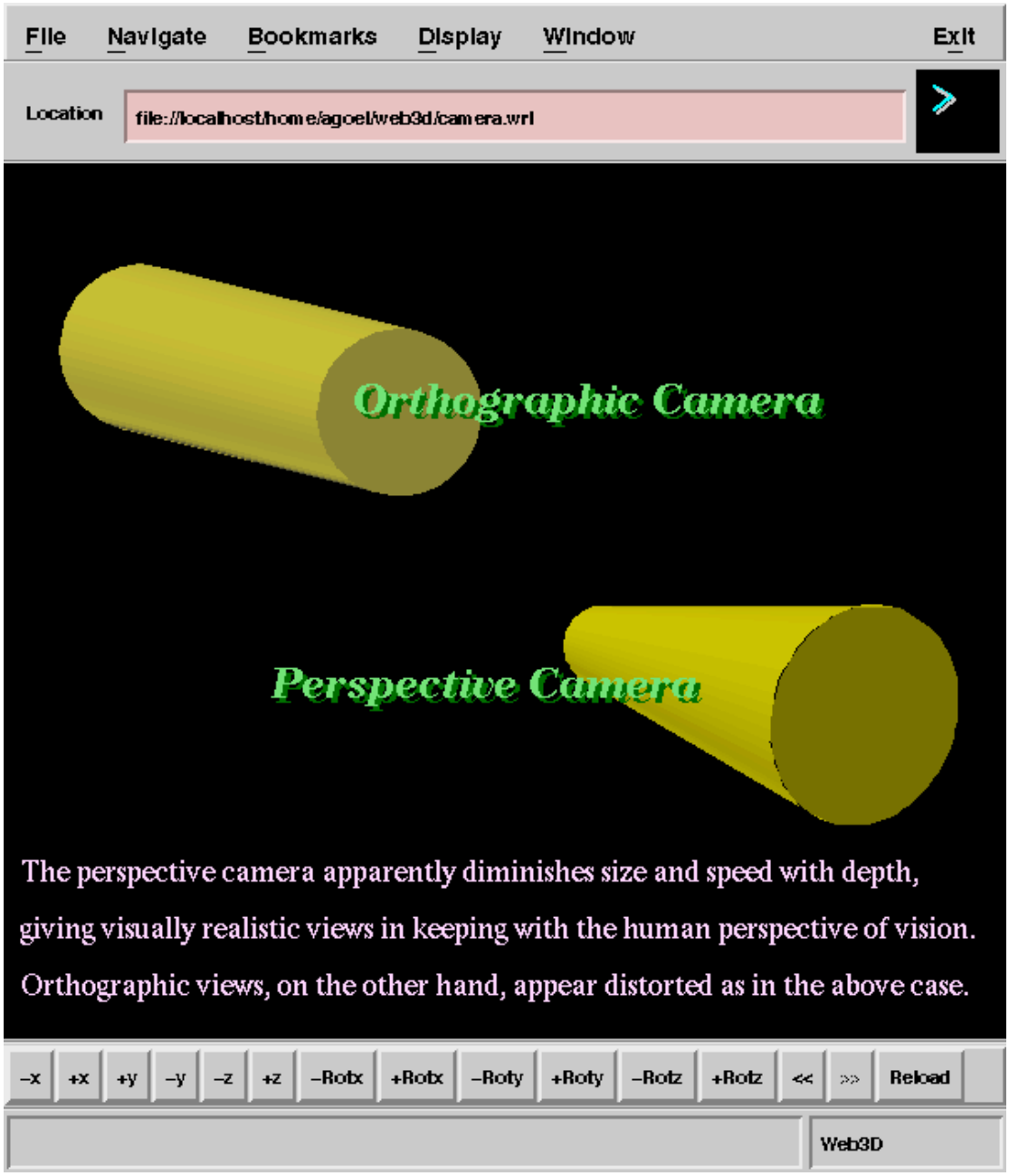

Figure 4 Orthographic and perspective views of a cylinder

4.4 Visual Realism through Shading

So far we have considered projections of three-dimensional objects on to the graphics viewport. But in order to produce visually realistic images, one needs to perform *shading* of the surfaces of these objects depending on the position and intensity of light sources, and how light interacts with the various objects in a scene. Smooth shading attains the highest form of visual realism.

Vision is a perception of light reflected onto the retinas of our eyes. Different materials reflect light in different ways, enabling us to distinguish between them, but all that we actually "see" is light. The purpose of a shading model is to calculate what light is reflected to the eye from each visible point in a scene, and then to use this information, by selecting a suitable form of display for the corresponding pixel, to create realistic images of the scene. Thus there are two distinct problems to consider. First, a mathematical model must be developed to provide the information needed about the light reflected from points in a scene; and second, this information must be interpreted for application to new facet display functions.

We assume that light consists of an infinite number of closely packed "rays" or "beams" which we may represent as vectors. There are two models which have been used for a light source in our browser. The point source model assumes that all rays emanate from a single point and may take any direction from this point. This idea corresponds to the properties of a single light bulb, or, on a larger scale, the sun.

Paradoxically, the sun may also be considered to fall into the second category -- parallel beam illumination -- which models the illumination produced by a point light source "infinitely" far from the object being illuminated or, alternatively, by a distributed light source. This model assumes that all rays

have a common direction, as with fluorescent lamps. Figure 5 shows one sphere illuminated by a point light source and another with a parallel light source.

Either a parallel beam or a point light source may be represented by a single vector specified in relation to the OBSERVER coordinate system. In the parallel case the vector is treated as a point vector from which the direction can be calculated. The position of a point light source is specified by a vector **s**, and in both cases the direction of the light illuminating a point **p** on the surface is called vector **-l**. Note the minus sign -- we adopt this notation because in most cases we use the direction vector pointing out of the surface in the opposite direction: that is, vector **l**. In order to calculate the light reflected from a point **p** on a surface we need to know the normal to the surface at **p**, which we call **n**, together with a direction vector from point **p** on the surface towards the light source. For the parallel beam model, finding this direction is easy -- it is the vector **l** for every point **p**. For the point source model, the required vector is **s**-**p**, which, for consistency, we shall also call **l**. For calculations involving specular reflection (see later) we also need to know the position of the camera, which, of course, we have placed at the origin of the OBSERVER coordinate system.

Quantifying light -- intensity and colour

Rays of light may vary in brightness or intensity. Ultimately, we wish to calculate the intensity of the light which is reflected to the eye from a point in a three-dimensional scene, and to interpret this information for display on the graphics screen. In order to do this, we must be able to map the measure of intensity onto the set of colours or shades available for display. The range of colours on any graphics display is finite -- there is a limit on brightness. We must therefore impose a maximum value on intensity, so we measure intensity of light using a floating point value between 0

(representing darkness) and 1 (representing maximum brightness).

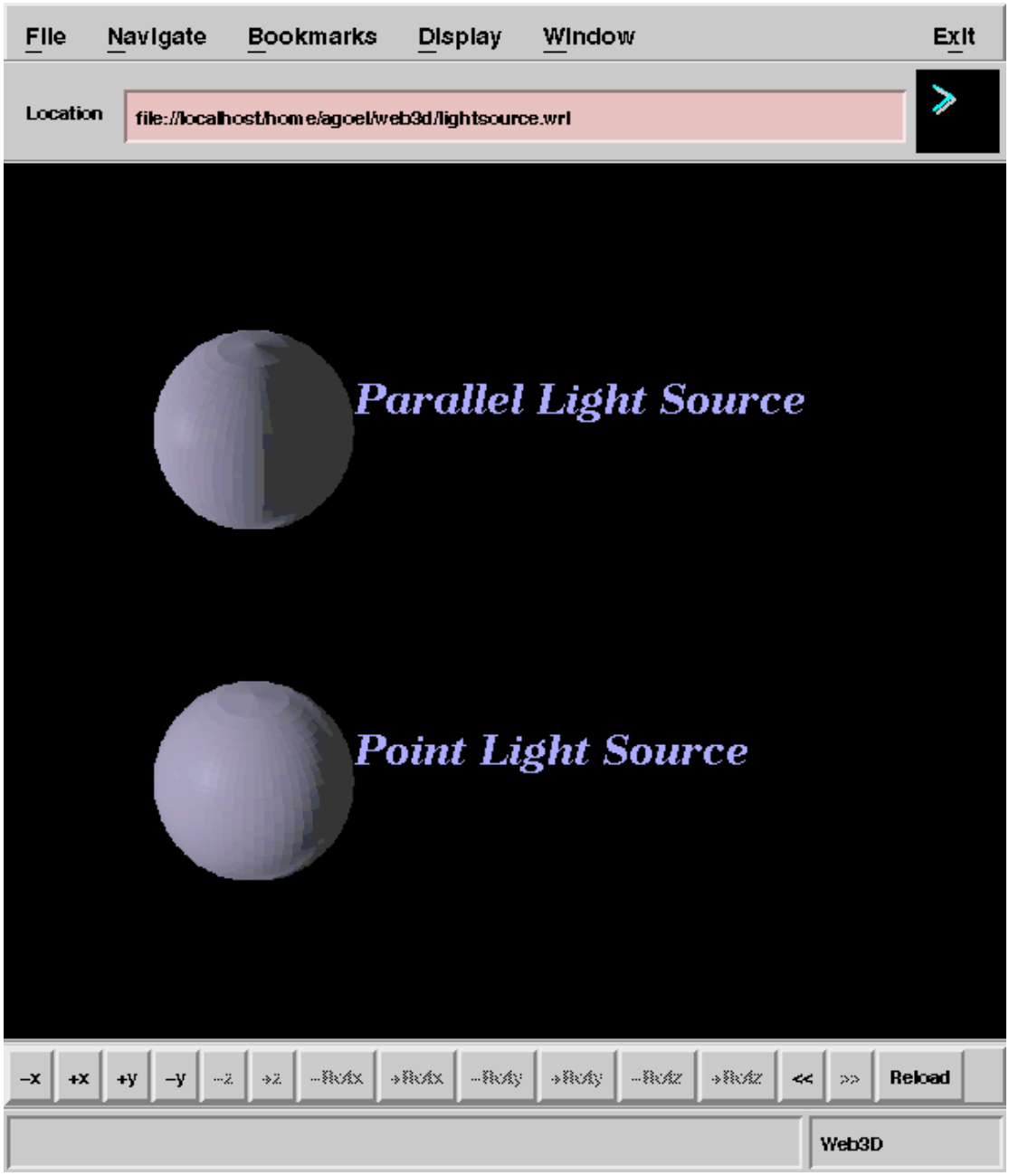

Figure 5 Parallel and point light sources

White light consists of a wide spectrum of waves of varying wavelengths, each corresponding to light of a different colour, ranging from red light at one end of the spectrum of visible wavelengths to violet at the other. In our somewhat simplistic conception of this idea we assume that light can be represented by three components -- red, green, and blue. We quantify light in terms of the intensities of each of these three components. Each of these intensities takes a floating point value between 0 and 1. In white light these components are present in equal measure. A value of 1 for I_{red}, 0 for I_{green}, and 0 for I_{blue} implies bright red light, whereas 0 for I_{blue} and 0.5 for both I_{red} and I_{green} implies a subdued yellow light. The "colour" of the light is determined by the triple (I_{red}, I_{green}, I_{blue}). A colour ($\alpha*I_{red}$, $\alpha*I_{green}$, $\alpha*I_{blue}$) for some value of α, $0 \leq \alpha \leq 1$, is said to be a "shade" of (I_{red}, I_{green}, I_{blue}) with intensity α.

The colour of a surface

All materials have properties relating the intensity of light which they reflect to that of light striking them (incident light). We call these properties the "reflective coefficients" of the material. We divide the properties into three components corresponding to the red, green, and blue components of the light. The values of the R_{red}, R_{green}, and R_{blue} coefficients represent respectively the proportion of the incident red, green, and blue light which is reflected, each taking a value between 0 and 1. A value of 1 for R_{red} implies that all incident red light is reflected, while values of 0 or 0.5 imply respectively that none or half the incident red light is reflected.

The absolute colour of a material is determined by the relative magnitudes of the R_{red}, R_{green}, and R_{blue} coefficients. For a white material all three are equal to 1, for a black material all are 0, while any material with equal R_{red}, R_{green}, and R_{blue} values between 0 and 1 is a shade of grey. A large R_{red}

coefficient combined with small R_{green} and R_{blue} gives a reddish colour and so on.

The *apparent colour* of a point on a surface is the colour of light reflected to the eye from that point on the surface. This is obviously dependent on the light illuminating the surface (including ambient light) as well as on the absolute colour and other properties of the surface (for example, gloss), but in the simple case of a dull (matt) surface illuminated by white light, the apparent colour is always a shade of the absolute colour.

Reflection of Light

There are two distinct ways in which light is reflected from a surface -- *diffuse reflection* and *specular reflection*. All surfaces exhibit diffuse reflection. When light hits a matt surface, it is scattered in all possible directions (we assume uniformly), so that the intensity of light reflected to the eye in this way is independent of the position from which the surface is viewed. We then see an apparent colour for the surface which is dependent on both the reflective colour coefficients of the surface and the colour of the incident light.

Glossy surfaces also exhibit specular reflection, the effect which produces the *highlights* clearly observed on the shiny surface of a metallic sphere illuminated by a light source. In this case, most of the light is reflected off the surface -- very little light is absorbed, and so the colour of the specularly reflected light is not dependent on the reflective coefficients of the surface.

Specular reflection is governed by two parameters which we call *m* and *s*. The parameter *m* is a measure of the *gloss* of the surface material, or the *specular reflection exponent*, and refers to the sharpness of fall-off in intensity of reflection

along directions deviating from reflection direction **r**. It takes an integer value between 1 and 400 theoretically, but for our purposes we found that a value of 10 gave sufficiently glossy surfaces. In the language specifications, the value of gloss varies between 0 and 1, which is actually scaled to a value between 0 and 10.

Not all light is reflected straight to the eye. Diffuse reflection, for instance, scatters light uniformly in all directions. This results in a low level of *ambient light* illuminating any scene. This is background light reflected equally in all directions from the ground, walls, and other exposed surfaces. We assume that ambient light illuminates all surfaces of the model equally and ensures that those surfaces which are not exposed to a genuine light source do not appear perfectly black. The colour of ambient light is, of course, dependent on the reflective coefficients of the surfaces from which it has been reflected.

In the illumination model we have used, we assume that all incident light (both source and ambient) is white light, thereby consisting of equal measures of red, green, and blue components. The intensity of ambient light is a floating point number between 0 and 1. This value can be set in the source code for describing 3D scenes. It is usually taken to be 0.2, i.e. 20% of the light in a scene comes from an ambient light source. We call this intensity I_a. The maximum intensity of light that may illuminate a scene is 1. This includes both ambient light and light emanating directly from a source. The intensity contribution of incident light from a source is therefore limited to $(1 - I_a)$. The intensity of light emitted by a source is given a value between 0 and 1, called I_s, and the incident light from this source, therefore, has the intensity value given by $I_s*(1 - I_a)$.

4.5 Developing a Shading Model

The ideal shading model calculates the precise colour of light reflected into the eye from any visible point in a scene. Therefore, such a model is required to determine the intensities of red, green, and blue components of this colour for any given point. This we call a *colour shading model*.

For our shading model, we use a number of parameters called *material properties*. These are the properties that govern the way in which a material reflects light -- its reflective coefficients, gloss, shine, etc. For our simple intensity shading model, we use just one attribute, R say, with a value between 0 and 1, which represents a general reflective coefficient. This attribute is composed of three reflective coefficients, one for each primary colour. These are the RGB reflective coefficients R_{red}, R_{green}, and R_{blue}.

Ambient light

We begin by modelling the reflection of ambient light which illuminates all surfaces equally, including those facing away from the genuine light source. Rays of ambient light strike a surface from all directions and are reflected uniformly in all directions. The intensity of light reflected to the eye (I_{amb}) is therefore independent of all but the intensity of the ambient light and the reflective coefficient of the surface with respect to this light.

For our colour shading model, we calculate all three components of ambient light as follows:

$$I_{amb(red)} = R_{red} \times I_a \quad (4.1)$$

$$I_{amb(green)} = R_{green} \times I_a \quad (4.2)$$

$$I_{amb(blue)} = R_{blue} \times I_a \quad (4.3)$$

Diffuse reflection

Diffuse reflection is modelled using *Lambert's Cosine Law*. This relates to the intensity of light striking a point on the surface to the cosine of the angle θ between the normal to the surface at that point and the vector from the point to the light source. Naturally, if the angle θ is greater than a right angle then the surface at **p** faces away from the source, and so no light reaches the surface, and consequently none is reflected. In this case I_{diff} works out to be less than 0, as we shall see. It must be set to 0 in such cases.

The model for diffuse reflection has been improved by the inclusion of a distance factor -- that is, the intensity of light from a given source falls off with increasing distance from the source. At a point that is a distance *d* from a source producing light of intensity I_s, the light has intensity proportional to I_s/d^2. But this leads to a sharp and unnatural fall-off in intensity as d increases. So a linear fall-off in intensity has been used in our programs, i.e. the light now has intensity proportional to $I_s/(d+C)$ where C is some constant value that must be set experimentally to give aesthetically pleasing results.

The three components of diffuse light are calculated as follows:

$$I_{diff(red)} = R_{red} \times I_s \times (1-I_a) \times \cos(\theta) \quad (4.4)$$

$$I_{diff(green)} = R_{green} \times I_s \times (1-I_a) \times \cos(\theta) \quad (4.5)$$

$$I_{diff(blue)} = R_{blue} \times I_s \times (1-I_a) \times \cos(\theta) \quad (4.6)$$

Specular reflection

Specular reflection, as mentioned previously, is exhibited by glossy surfaces. We have used the model developed by Bui Tong Phong. This method approximates the intensity of specular reflection at a point by using the value of cos(*m* x α), where α is the angle between the direction of perfect reflection of light from the point and the vector from that point to the eye, as shown in Figure 6, and *m* is the gloss of the surface material.

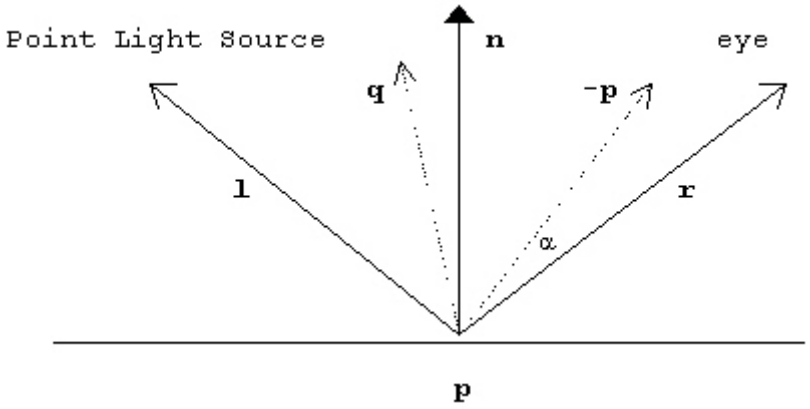

Figure 6 Specular reflection

If **r** is the direction in which light is reflected from the surface and **-p** is the vector from **p** to the eye, then the value of cos(α) is given by cos(α) = **r**.(**-p**)/(|**r**|x|**p**|). We could have taken advantage of elementary laws of trigonometry to simplify this calculation, which enables us to calculate cos(α) without first calculating **r**. But there is an alternative method that reduces the number of scalar products required, by one. And we have used that method. Here, we calculate the average of the two vectors **p** and **l**, i.e.

$$\mathbf{q} = -\mathbf{p}/|\mathbf{p}| + \mathbf{l}/|\mathbf{l}|$$

Then $\alpha/2$ is the angle between **q** and **n**, and so

$$\cos(\alpha/2) = \mathbf{n}.\mathbf{q}/(|\mathbf{n}|x|\mathbf{q}|)$$

Knowing that $\cos(\alpha) = 2\cos^2(\alpha/2)-1$, we may calculate $\cos(\alpha)$.

Specular reflection can only be used with a colour shading model, like we have used. It cannot be used with an intensity shading model. This is because the apparent colour of a point near, but not at, a highlight is neither a shade of the absolute colour of the surface nor a shade of the colour of the light source, but rather it is a mixture of the two colours. It should be pointed out that Bui Tong Phong's model does not strictly simulate the specular reflection of light, but simply produces an effect of similar appearance.

Each colour component in the complete colour shading model is calculated by summing the corresponding components of the contributions from reflected ambient light, diffuse reflection, and specular reflection. If any colour component exceeds 1, then naturally it must be set to 1. Since we have implemented polygon-shading of 3D objects, this colour component is then used to fill one out of all the polygons that constitute a primitive. Colour value calculations must be done for each visible polygon on the surface of the primitive.

Figure 7 shows the flat-shaded model of a cube rendered in the drawing area of our Web3D browser. The cube consists of six four-sided polygons in all, with only three polygons visible at any time. Figure 8, Figure 9, and Figure 10 show respectively flat-shaded models of cones, a cylinder, and a sphere. Figure 11 shows the hidden line model of a cube. Figure 12 simulates the solar eclipse, with the light source placed directly behind a yellow sphere. Figure 13 and Figure 14 show some 3D worlds rendered with the help of the four 3D geometric primitives, mainly cuboids.

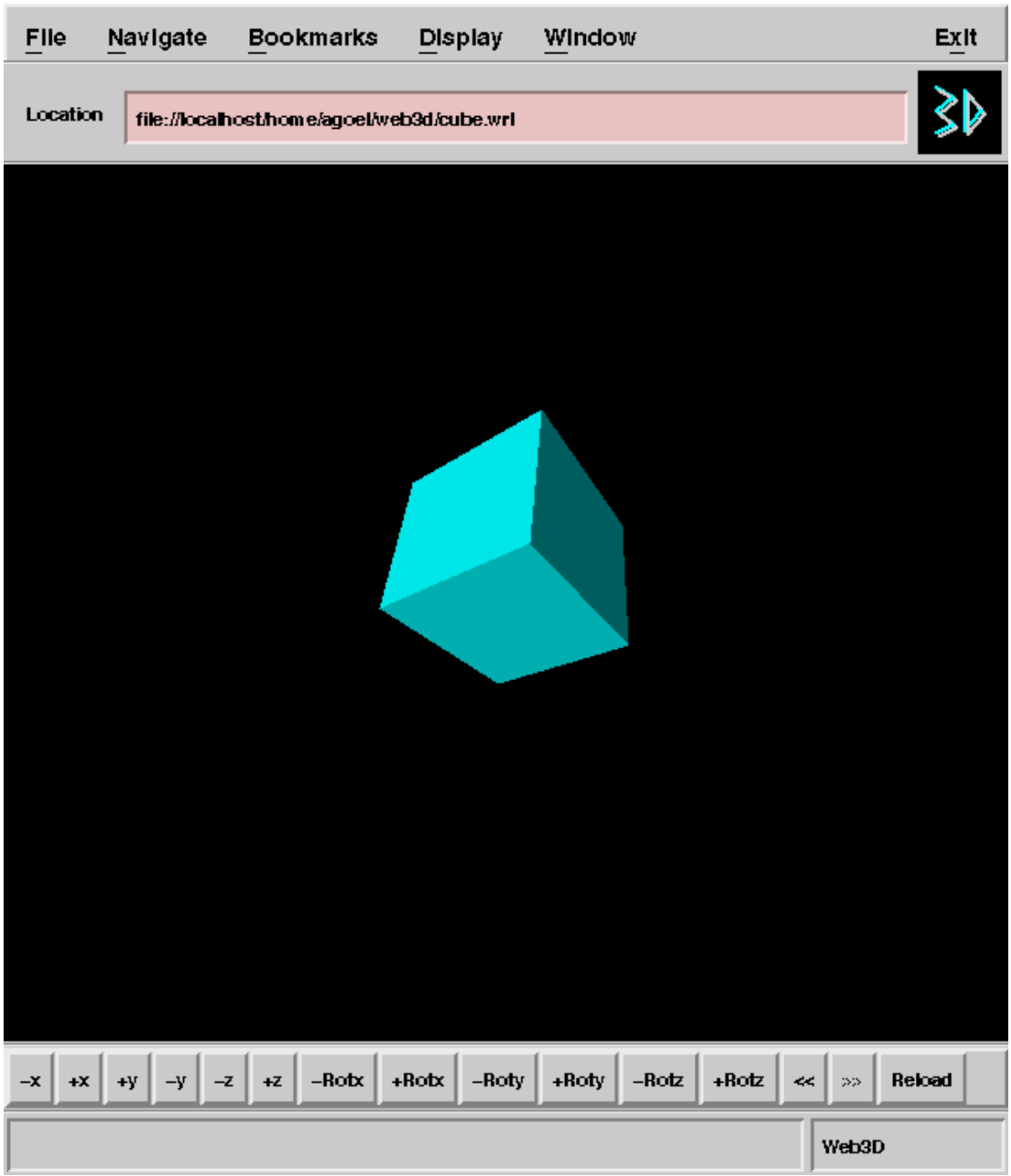

Figure 7 Rendering of a cube

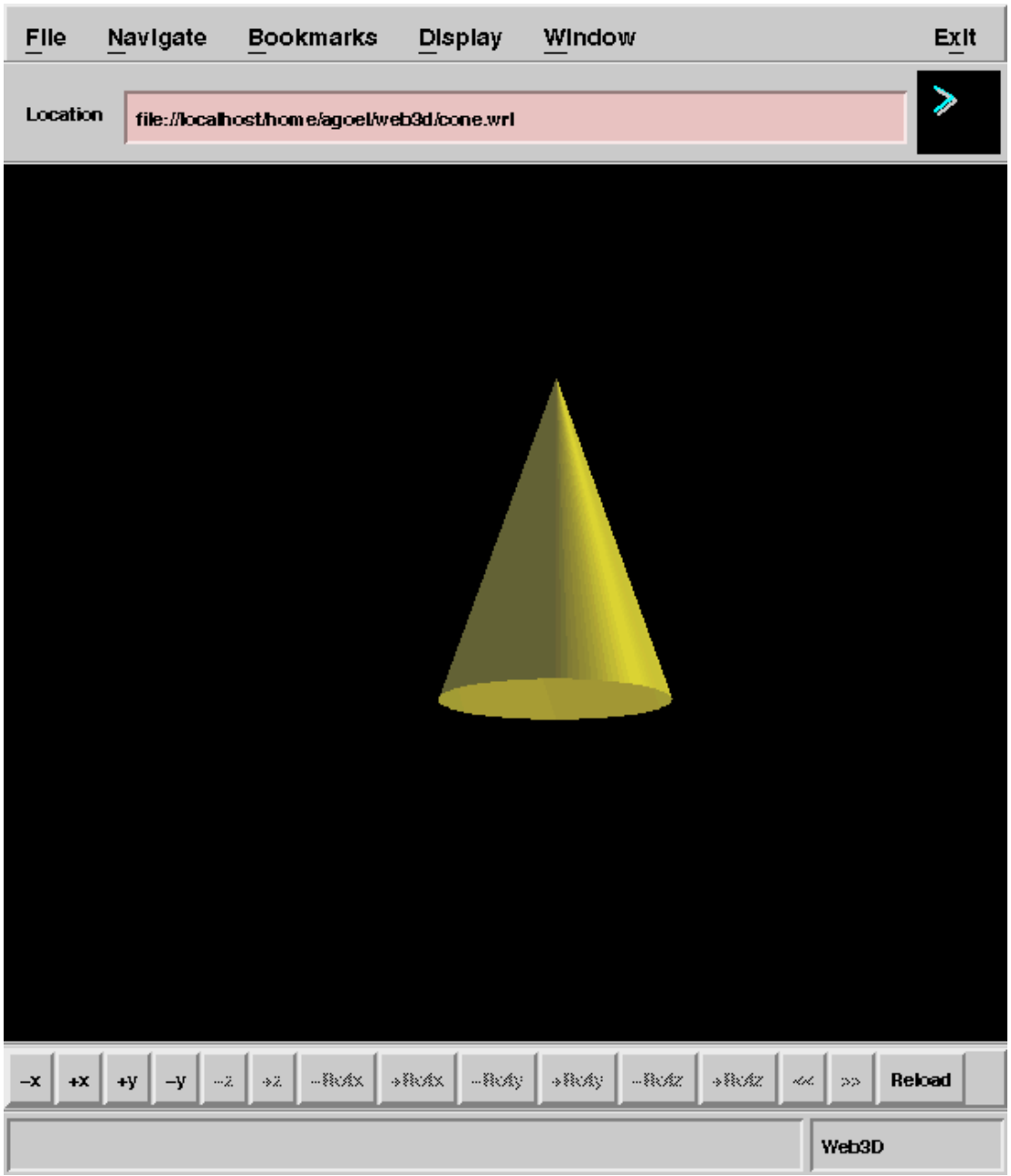

Figure 8 Polygonal rendering of a cone

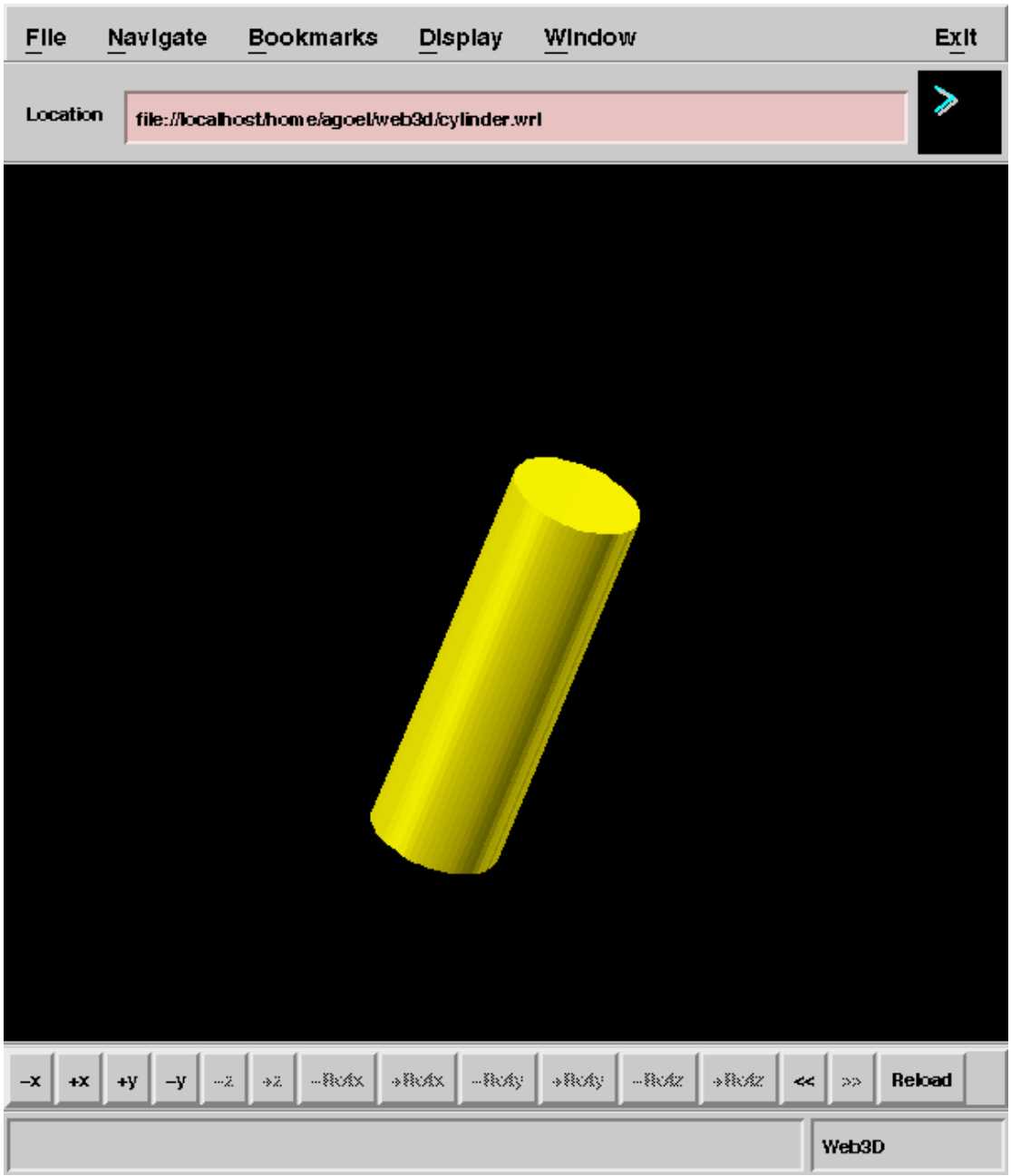

Figure 9 Polygonal rendering of a cylinder

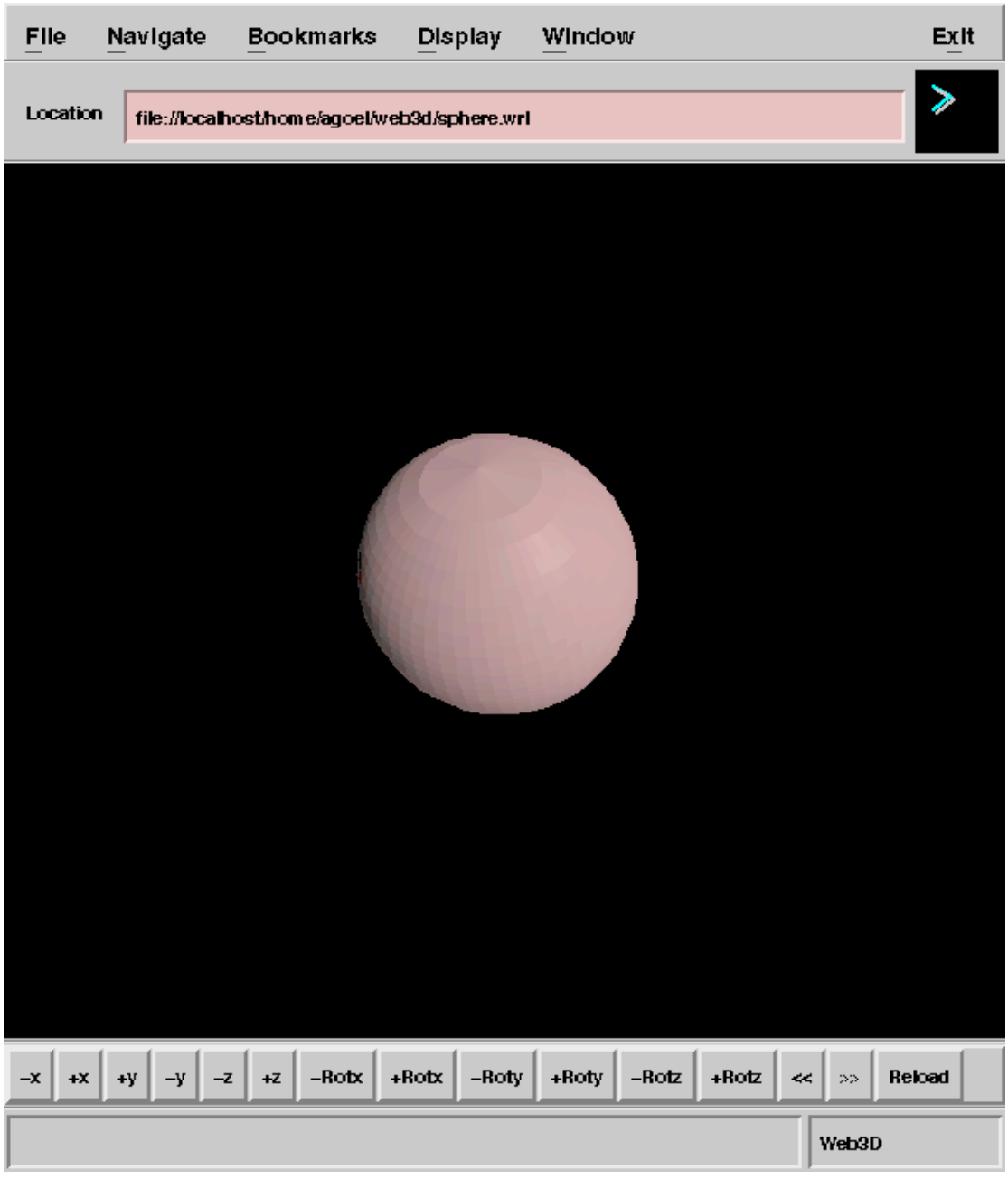

Figure 10 Polygonal rendering of a sphere

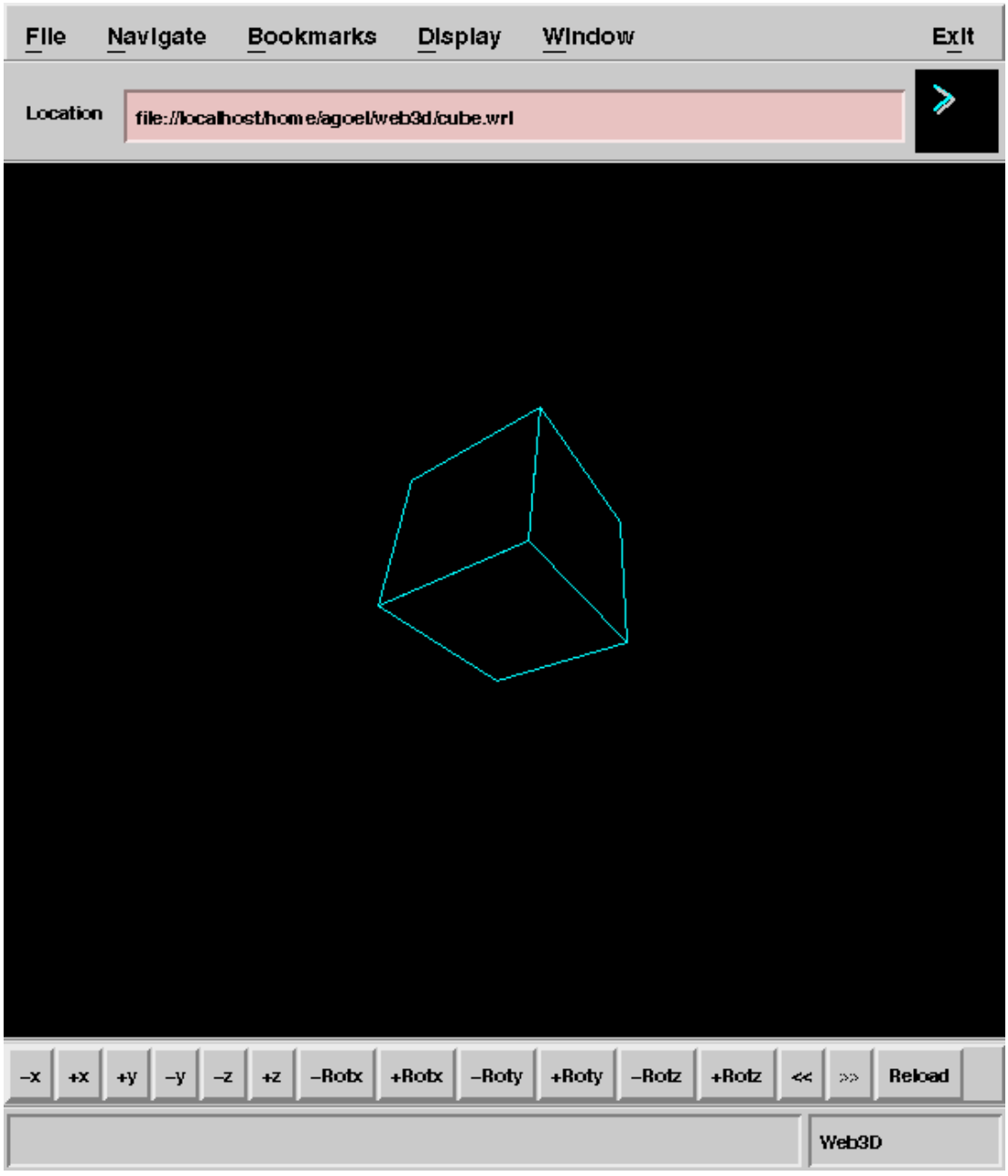

Figure 11 Hidden line rendering of a cube

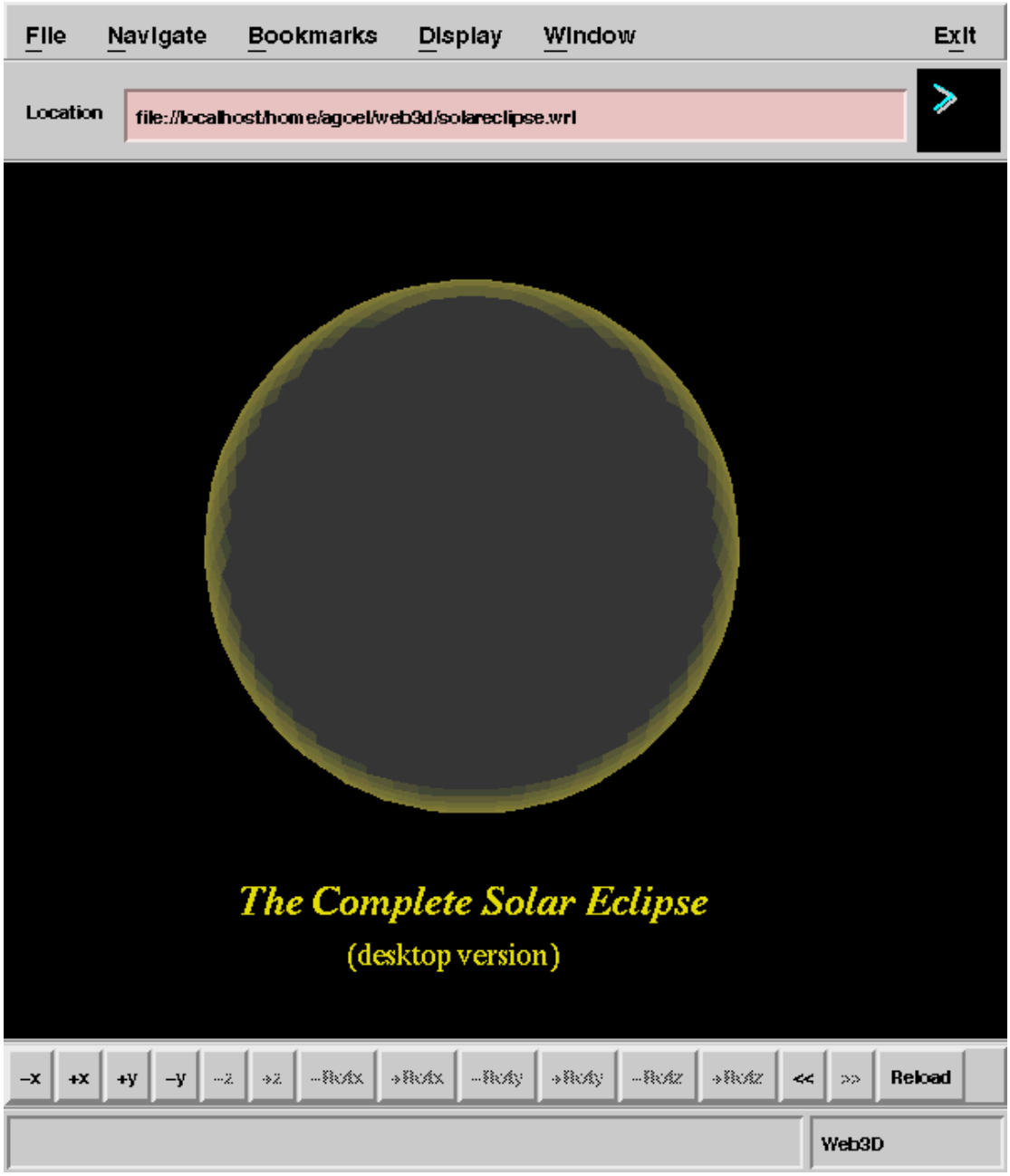

Figure 12 The solar eclipse

Figure 13 Flatshaded rendering of a billiard table

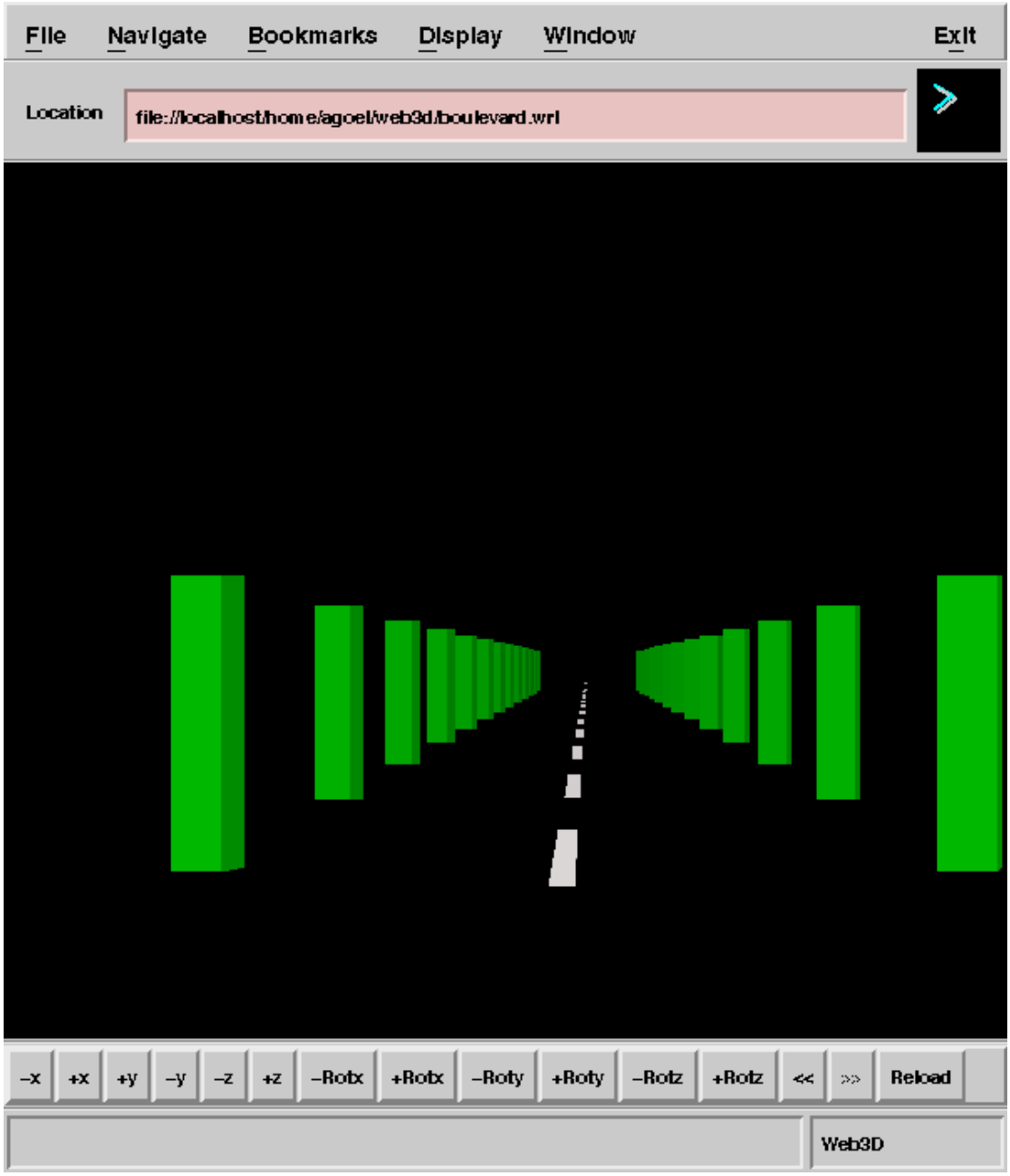

Figure 14 Flatshaded rendering of a boulevard

CHAPTER 5

3D ANIMATION

Web3D features both static as well as animated worlds. For static scene-descriptions, the size and position of objects in space is fixed. However, with the help of time-variant functions, Web3D is capable of displaying simple real-time animations.

5.1 Basic Implementation

Web3D is capable of interpreting time-varying values, giving rise to animated scenes. These kind of real-time animations belong to a class of animations called *procedural animations* where mathematical models define the geometry, shape, size, and path of objects as explicit functions of time. For example, we can have an object making periodic movements about an axis by defining its coordinates as trigonometric functions of time. Trigonometric functions supported by Web3D are sin() and cos(), and all expressions derived from them. Evaluation of expressions is recursive, as defined by the grammar of Web3D.

For example, a sphere revolving in an elliptical orbit can be defined by keeping its x-coordinate *a*cos(t), its z-coordinate *b*sin(t), its y-coordinate remaining constant. This animation routine can be implemented with the following source code:

```
# The following code describes a red sphere revolving in an
# elliptical orbit about the y-axis, as viewed by a stationary
# camera.  major axis = 500, minor axis = 300

Separator {

# light the scene
DirectionalLight
{     on  1                 # light source is switched on
      direction  1, 0, -1   # shining from viewer into scene
}

# set up the camera
PerspectiveCamera
{     focalDistance  200
      position  0 0 1000
      orientation  0, 0, -1
}

# define animation parameters
Animation
{     duration  30     # seconds
      velocity  1      # min = 0; max = 1
}

# render a sphere
Sphere
{     radius  100
      centre  500cos(t) 0 300sin(t)    # revolving about y-axis
      color  1 0 0                     # red sphere
}
} # end
```

In the above code, the x-coordinate and the z-coordinate of the sphere's position are time-variant, so that as time progresses, the sphere appears to revolve in an elliptical orbit about the y-axis. The time-variant expressions need to be parsed, converted to postfix, and then calculated for increasing values of t. The scene needs to be redrawn every time the value of t changes. The rate at which the scenes are redrawn is determined by the "velocity" parameter defined for the "Animation" construct. This floating point parameter takes 0 as its minimum value and 1 as its maximum value. Its value is mapped inversely to the time duration after which the rendering routine will be called again. This recursive (callback) feature is handled automatically by the *Timeout* functions provided by the X11 library. When the rendering function is called again, it redraws the scene with freshly calculated values of time and other necessary variables. The *duration* parameter belonging to *Animation* node sets the time (in seconds) for which the animation must proceed.

While parsing an input file, the interpreter typically checks for any time-variant properties. In case there are no time-variant properties, the scene is drawn straightaway. Otherwise the expressions are stored in a buffer, and after the first parse, a loop is executed, which evaluates the function with values of time-variable 't' interpolated from 0 to the desired time, in steps of say 0.5. These values are all stored in a temporary buffer, and then the animation process begins. The buffer is used to obtain the values of the position, orientation etc. of objects at any point of time while the animations are in progress. This greatly speeds up the animation process, since the browser has to then only concentrate on placing the objects correctly in space during animation, rather than having to evaluate the expressions during the animation process. Since the implementation of animations is an object-oriented approach, the user is free to carry on his or her normal operations with the browser while the animation proceeds in the background.

5.2 Introduction to Quaternions

Quaternions provide an effective and convenient means for representing the orientation of an observer with good behaviour during interpolation. Quaternions are complex numbers with one real component and three imaginary components. The imaginary units are i, j, and k, and have the following properties:

$$i^2 = j^2 = k^2 = -1$$
$$ij = k \quad \text{and} \quad ji = -k$$

with the cyclic permutation $i \rightarrow j \rightarrow k \rightarrow i$.

Quaternions can be represented as:

$$q = a + bi + cj + dk.$$

For our purposes we shall use the condensed notation:

$$q = (s, \mathbf{v})$$

where

$$(s, \mathbf{v}) = s + \mathbf{v}_x i + \mathbf{v}_y j + \mathbf{v}_z k$$

s is thought of as the scalar part of the quaternion and v the vector part with axes i, j, and k. Using the above rules it is easy to derive the following properties. The multiplication of two quaternions:

$$q1 = (s1, \mathbf{v1}) \quad \text{and} \quad q2 = (s2, \mathbf{v2})$$

is given by:

$q1q2 = (s1s2 - \mathbf{v1.v2},\ s1\mathbf{v2} + s2\mathbf{v1} + \mathbf{v1}*\mathbf{v2})$

The multiplication of two quaternions is thus a quaternion. Mathematically, we have defined a group. Stated simply, a group is just a set of elements with a rule defining their multiplication such that the result of this multiplication is itself an element of that group. Groups can be constructed completely arbitrarily, though a surprising number of groups are relevant to the physical world. We shall see that a subgroup of the quaternion group is closely related to the group of rotation matrices.

Note that except for the cross product term at the end of the previous equation, it bears a strong similarity to the law of complex multiplication:

$(a1 + ib1)(a2 + ib2) = (a1a2 - b1b2) + i(a1b2 + a2b1)$

The cross product term has the effect of making quaternion multiplication noncommutative.

We define the *conjugate* of the quaternion:

$q = (s,\mathbf{v})$ to be $\overline{q} = (s,-\mathbf{v})$

The product of the quaternion with its conjugate defines its magnitude:

$q\overline{q} = s^2 + |\mathbf{v}|^2 = |q|^2$

Finally, take a pure quaternion (one that has no scalar part):

$p = (0,\mathbf{r})$

and a unit quaternion

$q = (s,\mathbf{v})$ where $q\overline{q} = 1$

and define

$R_q(p) = qpq^{-1}$

Using our multiplication rule, and the fact that $q^{-1} = q$ for q of unit magnitude, this expands to:

$$R_q(p) = (0, (s^2 - \boldsymbol{v}.\boldsymbol{v})\boldsymbol{r} + 2\boldsymbol{v}(\boldsymbol{v}.\boldsymbol{r}) + 2s\boldsymbol{v}\times\boldsymbol{r}) \qquad (5.1)$$

This can be simplified further since q is of unit magnitude, and we can write:

$q = (\cos\theta, \sin\theta\boldsymbol{n}) \qquad |\boldsymbol{n}| = 1$

Substituting into Equation (5.1) gives:

$$R_q(p) = (0, (\cos^2\theta - \sin^2\theta)\boldsymbol{r} + 2\sin^2\theta\boldsymbol{n}(\boldsymbol{n}.\boldsymbol{r}) + 2\cos\theta\sin\theta\boldsymbol{n}\times\boldsymbol{r})$$
$$= (0, \cos2\theta\boldsymbol{r} + (1-\cos2\theta)\ \boldsymbol{n}(\boldsymbol{n}.\boldsymbol{r}) + \sin2\theta\boldsymbol{n}\times\boldsymbol{r}) \qquad (5.2)$$

From this, we can say that the act of rotating a vector $\boldsymbol{r}$ by an angular displacement $(\theta,\boldsymbol{n})$ is the same as taking this angular displacement, "lifting" it into quaternion space, by representing it as the unit quaternion $(\cos(\theta/2),\sin(\theta/2)\boldsymbol{n})$ and performing the operation $q()q^{-1}$ on the quaternion $(0,\boldsymbol{r})$. We could therefore parametrize orientation in terms of the four parameters:

$\cos(\theta/2),\ \sin(\theta/2)\boldsymbol{n}_x,\ \sin(\theta/2)\boldsymbol{n}_y,\ \sin(\theta/2)\boldsymbol{n}_z$

using quaternion algebra to manipulate the components.

In practice this would seem an extremely perverse way of going about things were it not for one very important advantage afforded by the quaternion parametrization. Two quaternions multiplied together, each of unit magnitude, will result in a

single quaternion of unit magnitude. If we use quaternions to represent rotations, then this translates to two successive rotations producing a single rotation.
Now let us see how quaternions work in practice. One single x-roll of π is represented by the quaternion:

$$(\cos(\pi/2), \sin(\pi/2)\ (1,0,0)) = (0, (1,0,0))$$

Similarly, a y-roll of π and a z-roll of π are given by $(0,(0,1,0))$ and $(0,(0,0,1))$ respectively. Now the effect of a y-roll of π followed by a z-roll of π can be represented by the single quaternion formed by multiplying these two quaternions together:

$$\begin{aligned}(0,(0,1,0))\ (0,(0,0,1)) &= (0,(0,1,0) \times (0,0,1)) \\ &= (0,(1,0,0))\end{aligned}$$

which is a single x-roll of π. From this we can see that the cross product term in Equation (5.2) can be thought of as correcting for the interdependence of the separate axes that is ignored by *Euler's angle* notation. Euler angles refer to a family of methods by which an object is rotated successively three times around three fixed axes to produce a composite rotation. An additional advantage afforded by using quaternions is that the gimbal lock singularity, which is a consequence of using three parameters to parametrize orientation, disappears.

5.3 Interpolating using Quaternions

Given the superiority of quaternion parametrization over other ways of parametrization, this section covers the issue of interpolating rotation in quaternion space. Consider an animator sitting at a workstation and interactively setting up a sequence of key orientations by whatever method is appropriate in the Web3D environment. This is usually done with the principal rotation operations, but now the restrictions that were placed on the animator when using Euler angles, namely using a fixed number of principal rotations in a fixed order for each key, can be removed. In general, each key will be represented as a single rotation matrix. This sequence of matrices will then be converted into a sequence of quaternions. Interpolation between key quaternions is performed and this produces a sequence of in-between quaternions, which are then converted back into rotation matrices. The matrices are then applied to the object. The fact that a quaternion interpolation is being used is transparent to the animator.

Moving into and out of quaternion space

The implementation of such a scheme requires us to move into and out of quaternion space, that is, to go from a general rotation matrix to a quaternion and vice versa. It can be shown that the effect of taking a unit quaternion:

$$q = (\cos(\theta/2), \sin(\theta/2)\boldsymbol{n})$$

and performing the operation $q()q^{-1}$ on a vector is the same as applying the following rotation matrix to that vector:

$$\begin{bmatrix} 1-2Y^2-2Z^2 & 2XY-2WZ & 2XZ+2WY & 0 \\ 2XY+2WZ & 1-2X^2-2Z^2 & 2YZ-2WX & 0 \\ 2XZ-2WY & 2YZ-2WX & 1-2X^2-2Y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the quaternion $(\cos(\theta/2), \sin(\theta/2)\boldsymbol{n})$ is written as $(W,(X,Y,Z))$. By these means then, we can move from quaternion space to rotation matrices.

The inverse mapping from a rotation matrix to a quaternion is only slightly more involved. All that is required is to convert a general rotation matrix:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

into the matrix format directly above. The resulting quaternion is trivially $(W,(X,Y,Z))$. Given a general rotation matrix, the first thing to do is to examine the sum of its diagonal components M_{ii} where $0 \le i \le 3$. This is called the trace of the matrix. From the above format we know:

$$\begin{aligned} \text{trace} &= 1-2Y^2-2Z^2+1-2X^2-2Z^2+1-2X^2-2Y^2+1 \\ &= 4-4(X^2+Y^2+Z^2) \end{aligned}$$

Since the matrix represents a rotation, we know that the corresponding quaternion must be of unit magnitude, that is:

$X^2+Y^2+Z^2+W^2 = 1$

and so the trace reduces to $4W^2$. Thus for a 4x4 homogeneous matrix we have:

$W = (\text{trace})^{1/2}$

The remaining components of the quaternion (X,Y,Z) which is the axis of rotation scaled by half the sine of the angle of rotation, are obtained by combining diagonally opposite elements of the matrix M_{ij} and M_{ji}, where $0 \leq i,j \leq 2$. We have:

$X = (M_{21}-M_{12})/4W$
$Y = (M_{02}-M_{20})/4W$
$Z = (M_{10}-M_{01})/4W$

For zero W these equations are undefined and so other combinations of the matrix components, along with the fact that the quaternion is of unit magnitude, are used to determine the axis of rotation.

Having outlined our scheme, we now discuss how to interpolate in quaternion space. Since a rotation maps onto a quaternion of unit magnitude, the entire group of rotation maps onto the surface of the four-dimensional unit hypersphere in quaternion space. Curves interpolating through key orientations should therefore lie on the surface of this sphere. Consider the simplest case of interpolating between just two key quaternions. A naive, straightforward, linear interpolation between the two keys results in a motion that speeds up in the middle. This is because we are not moving along the surface of the hypersphere but cutting across it. In order to ensure a steady rotation we must employ *spherical linear interpolation*, where we move along an arc of the geodesic that passed through the two keys.

Spherical linear interpolation

The formula for spherical linear interpolation is easy to derive geometrically. Consider the two-dimensional case of two vectors **A** and **B** separated by an angle Ω and vector **P** which makes an angle θ with **A** as shown in Figure 15. **P** is derived from spherical interpolation between **A** and **B** and we write:

$$\mathbf{P} = \alpha\mathbf{A} + \beta\mathbf{B}$$

Trivially, we can solve for α and β given:

$$|\mathbf{P}| = 1$$
$$\mathbf{A}.\mathbf{B} = \cos(\Omega)$$
$$\mathbf{A}.\mathbf{P} = \cos(\theta)$$

to give:

$$\mathbf{P} = \mathbf{A}\ \sin(\Omega-\theta)/\sin(\Omega) + \mathbf{B}\ \sin(\theta)/\sin(\Omega)$$

Spherical linear interpolation between two unit quaternions *q1* and *q2*, where:

$$q1.q2 = \cos(\Omega)$$

is obtained by generalising the above to four dimensions and replacing θ by Ωu where $0 \le u \le 1$. We write:

$$\mathrm{slerp}(q1,q2,u) = q1\ \sin(1-u)\Omega/\sin(\Omega) + q2\ \sin(\Omega u)/\sin(\Omega)$$

Function slerp(*p*,*q*,*t*,*qt*) returns the interpolated quaternion *qt*, for *t* between *p* and *q*. It caters to special cases where the keys are very close together, in which case we approximate using the more economical linear interpolation and avoid divisions by very small numbers since

$\sin(\Omega) \rightarrow 0 \qquad \text{as} \quad \Omega \rightarrow 0$

The case where p and q are diametrically opposite, or nearly so, requires special attention.

Now, given any two key quaternions, p and q, there exist two possible arcs along which one can move, corresponding to alternative starting directions on the geodesic that connects them. One of them goes around the long way, and this is the one we wish to avoid. Naively, one might assume that this reduces to either spherically interpolating between p and q by angle Ω, where

$p.q = \cos(\Omega)$,

or interpolating in the opposite direction by the angle $(2\pi - \Omega)$. This, however, will not produce the desired effect. The reason is that the topology of the hypersphere of orientation is not just a straightforward extension of the three-dimensional Euclidean sphere. To appreciate this, it is sufficient to consider the fact that every rotation has two representations in quaternion space, namely q and $-q$, that is, the effect of q and $-q$ is the same. That this is so is because algebraically the operator $q()q^{-1}$ has exactly the same effect as $(-q)()(-q)^{-1}$. Thus, points diametrically opposed represent the same rotation. Because of this topological oddity, care must be taken when determining the shorter arc. A strategy that works is to choose interpolating between either the quaternion pairs p and q or p and $-q$. Given two key orientations p and q, find the magnitude of their difference, that is $(p-q).(p+q)$, and compare this to the magnitude of the difference when the second key is negated, that is $(p+q).(p+q)$. If the former is smaller, then we are already moving along the smaller arc and nothing needs to be done. These considerations are shown schematically in Figure 16.

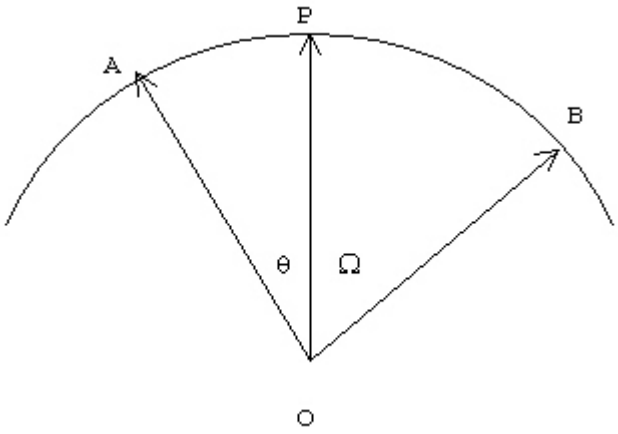

Figure 15 Spherical linear interpolation

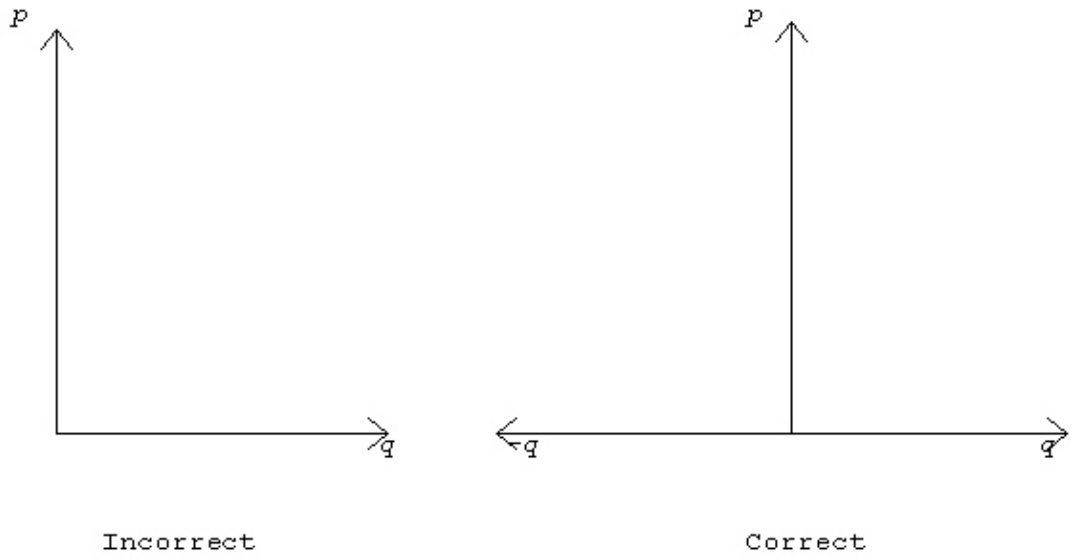

Figure 16 Shortest arc determination

CHAPTER 6

BROWSER DESCRIPTION

Based on the specifications discussed so far, a browser has been developed in C using X Window Motif user interface toolkit, and it has been named Web3D. Web3D uses the standard X11 library for graphical renderings, and Motif widget library for the interface layout and functionality.

6.1 The Web3D User Interface

The Web3D window is divided vertically into five areas: menu bar, file name display area, drawing area, tool bar, and status bar, as shown in Figure 17. The menu bar provides access to the full functionality of Web3D. The toolbar and additional accelerator keys and mnemonics provide quick access to commonly used functions like movement of camera, and document navigation.

The major functions included in the menu bar are as follows:

New: This function fires up the editor where one can create a new input source file in the file format recognised by Web3D. The standard extension for Web3D files is ".wrl" for world.

Figure 17 Web3D user interface

Open: One can open an existing *wrl* file by selecting from a file-selection box that is a one of Motif's widgets.

Edit Source: Edit the source code of any input file being browsed currently. Here, too, the editor is called, and one can view/edit the source code and then browse the output at the same time in the same Web3D environment.

Back/Forward: While following hyperlinks, users can go "Back" in the sequence of worlds they have browsed previously, or go "forward" in the sequence by one step (if any).

Add Bookmark: As in standard WWW browsers, this feature allows users to mark important worlds in a bookmarks file, which they can come back to later on.

Scene Information Window: Clicking on this menu item fires a small independent window that displays information regarding the scene currently being browsed in the display area. More specifically, it gives information about the position/orientation of the camera and light sources, the number of primitives used to render the scene, and the centre coordinates of these primitives in three-dimensional space.

The menu bar can also be used to select the rendering mode for a particular scene. Web3D supports four rendering modes: wireframe, hidden line, flat shading, and texturing. The rendering mode specified in the input file can be overridden. Besides, three levels of detail are provided: *low*, *medium*, and *high*. These modes determine the number of polygons used to render a primitive. *Low* detail uses very few polygons to speed up rendering and animation, whereas *high* detail option uses a considerably large number of polygons to render the primitives. On a display without hardware graphics acceleration, where frame rates are slow for shaded models, it might be advisable to navigate in wireframe under *low* level of detail, and see the shaded version in *high* detail only in the pauses between the animations.

The Web3D user interface is in the form of a window (the drawing area) that looks out into 3D space. Users (synthetic cameras) can enter this space, manipulate 3D objects placed there, throw light upon them, and experience the same sights as any observer would in real space. Navigation in 3D space is complicated by the need to control (at least) six degrees of freedom at the same time. Web3D 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. The mouse can be used to control the motion of the observer by clicking on push-buttons available on the browser's window. Web3D gives the observer six degrees of freedom, and allows him or her to control only one degree at a time. Should users get lost in a scene, a "Reload" function is available to restore the initial view of the scene.

Besides describing graphical objects and their attributes, Web3D is capable of embedding graphical objects within text documents, and linking these documents by means of hypertexts. Hyperlinks can be to other virtual worlds, or to HTML documents, in which case Web3D invokes a standard WWW browser (Mosaic or Netscape). Web3D, too, can be invoked by a properly configured WWW browser. Web3D has so far been known to interface with both *NCSA Mosaic* for X Windows and *Netscape Navigator* for X Windows.

6.2 Web3D's Software Architecture

Figure 18 shows the basic software architecture of Web3D. Users communicate with the browser via the Interface Layer which is the client-end user interface of Web3D. The arrows indicate the flow of data and control signals. The input stream in parsed and an internal stack of tokens is constructed. The rendering component (X11 library) then visualizes the contents of this stack and renders the scene correctly in the display area of the browser's window.

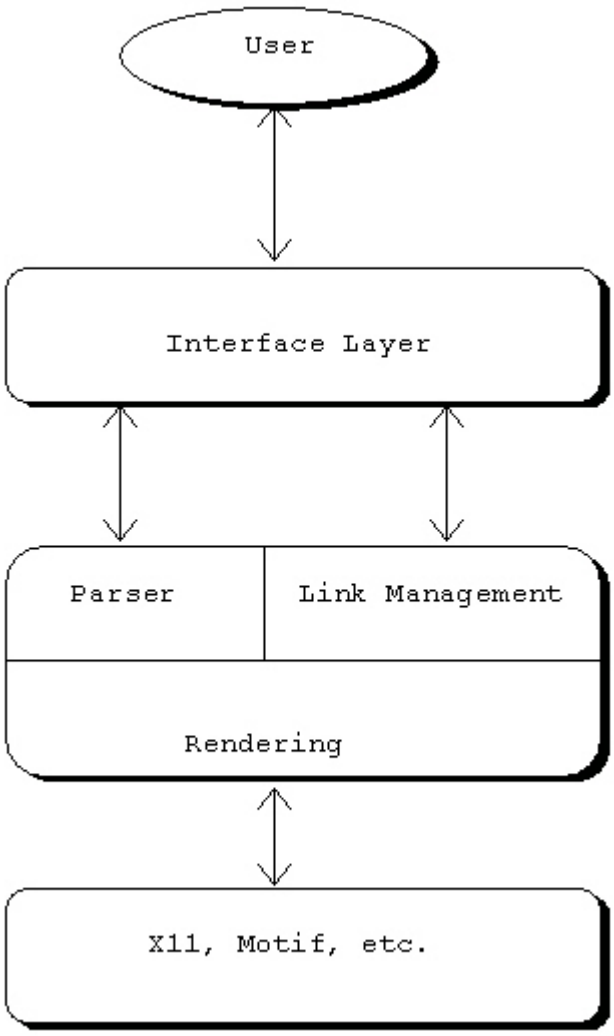

Figure 18 Web3D architecture

Thus the source code of Web3D is independent of any particular 'graphics' library for its graphical renderings, and is based on freely available software components except that it requires Motif, a commercial library, for building its 'interface'.

Every node has a separate function associated with it. This function alone is responsible for interpreting/rendering itself. For example, provided global variables and other environment settings are correct, the function cube() when called, draws a cube all by itself. The main routine that calls all the drawing functions is as follows:

```
while (array_of_tokens[token_count])
{
      /* scan the array of tokens created from input stream */

      switch (array_of_tokens[token_count++])
      {
            case  AMBIENTLIGHT         :  ambientlight();
            case  ANIMATION            :  animation();
            case  ASCIITEXT            :  asciitext();
            case  CONE                 :  cone();
            case  CUBE                 :  cube();
            case  CYLINDER             :  cylinder();
            case  DIRECTIONALLIGHT     :  directionallight();
            case  FOG                  :  fog();
            case  MATERIAL             :  material();
            case  ORTHOGRAPHICCAMERA   :  orthographiccamera();
            case  PERSPECTIVECAMERA    :  perspectivecamera();
            case  POINTLIGHT           :  pointlight();
            case  SPHERE               :  sphere();
            case  SPOTLIGHT            :  spotlight();
            case  TEXTPROPERTIES       :  textproperties();
      }

      /*  Flush the display in the drawing area  */

      XFlush(XtDisplay(wDraw));

}
```

As is evident from the code above, Web3D treats cameras and light sources as any other geometric primitive.

CHAPTER 7

CONCLUDING REMARKS

This thesis presented the specifications of a browser capable of presenting graphical renderings of a virtual 3D world within a computer. This 3D world may be a CAD model, a scientific simulation, a view into a database, or a walkthrough into a virtual world. Needless to say, such applications can add exciting new dimensions to the power and usefulness of the Web, adding new ways of interfacing with the user and presenting information.

In order to firmly establish Web3D as *the* standard for describing three-dimensional scenes and animations on the Internet, it is not sufficient to have a good language specification. Browsers capable of many protocols, and implemented on many platforms are necessary. Another crucial point is the free availability of source code for non-commercial organisations to allow research and experiment with this new type of media. These were the design goals of Web3D. We hope that the availability of an open development platform will contribute significantly to the evolution of the Web3D standard.

Web3D is still only a “viewer”, i.e. it has to depend on other applications like Mosaic for file retrievals. The ability to retrieve files across a network needs to be added. A number of animation features like animation of articulated structures, collision detection, and soft-object animations can be added to the visual realism of the rendered scenes. Real-time multi-user interaction over a network can be implemented, which will enable multi-participant simulations of the same 3D worlds simultaneously.

Web3D's 3D-scene-description language is not yet object-oriented, i.e. it lacks the capability of declaring prototypes for nodes, and the ability to use those nodes lower in the scene-graph hierarchy. The language can also be made more extensible by being able to declare nodes that are not part of the standard language specifications without having to recompile the code. Besides, a number of features can be added to the user interface of Web3D to make it closer in functionality to a standard Web browser.

REFERENCES

1. Najork and M.H. Brown, "Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System", IEEE Transactions on Visualization and Computer Graphics, Vol. 1, No. 2, June 1995.

2. Michael Pichler, Gerbert Orasche, Keith Andrews, Ed Grossman, and Mark McCahill, "VRweb: A Multi-System VRML Viewer", The First Annual Symposium on the Virtual Reality Modeling Language (VRML 95), December 1995.

3. Allen I. Holub, "Compiler Design in C".

4. Foley van Dam, "Fundamentals of Interactive Computer Graphics".

5. Ian O. Angell and Dimitrios Tsoubelis, "Advanced Graphics in Borland C++".

6. Nabajyoti Barkakati, "X Window Programming System".

7. Douglas A. Young, "The X Window System, Programming and Applications with Xt".

8. Andrew S. Glassner, Graphics Gems.

9. James Arvo, Graphics Gems II.

10. Marshall Brain, "Motif Programming: The Essentials and More".