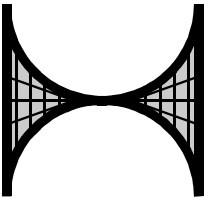


The Hypercosm 3D Graphics System

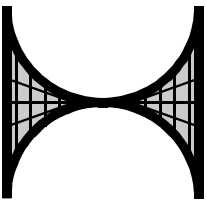
A Guide to the Hypercosm System & OMAR's Graphics Extensions

Revision: September 1999

Hypercosm, Inc.
(608) 821-0500
www.hypercosm.com

For technical support, please contact Hypercosm at the phone number above or send email to support@hypercosm.com.

To see support options, frequently asked questions, and information on the email discussion group, go to http://www.hypercosm.com/support/index.html.



Contents

Introduction 1

The 3D Graphics Process 1
How It All Works 2
Hypercosm Features 5
Sources for Additional Information 6

Getting Started 7

What You Need 7
OMAR Files. 8
Hypercosm's Standard Resource Files 8
Hypercosm's Sample Source Files 9
OMAR's Graphics Extensions 9
3D Coordinates. 10
A Note About Units 12
The Basic OMAR Graphics File. 12
A Simple Example 13
A More Interesting Example 14

Modeling 15

Elements of Modeling 15
Primitive Shapes 16

Using the Primitives 18
A Closer Look at the Primitives 26
Sweeps or Partial Surfaces 26
The Mesh Primitive 28
The Shaded Triangle & Shaded Polygon 31
The Volume Primitive 32
Lighting 33
Lighting Primitives 33
Ambient Light 34
Hierarchical Modeling 36
Relative Transformations 37
The Transformation Stack 43
The Current Transformation State 43
Transforming a Series of Shapes 43
Nesting Transformations 43
Absolute Transformations 45
Specifying Absolute Transformations 45
Mixing Relative and Absolute Transformations 45
Implementation of Absolute Transformations 45
Color 48
Assigning Color to Shapes 48
Predefined Colors 49
Materials 50
Coloring Precedence 51
Textures 51
Procedural Modeling 55
Parametric Procedural Models 55
Fractals in Nature 56
Implementing Fractals 57

Viewing 61

Camera Placement 61
Camera Orientation 62
Field of View 63
Projection 64
The Orthographic Projection 64
The Perspective Projection 65
The Fisheye Projection 67
The Panoramic Projection 68

Stereoscopic Pictures. 69
Stereo Glasses 70
How to Use the Stereo Feature. 70
Changing Stereo Colors 71
Producing Stereo Pairs 71

Rendering 73

Window Dimensions & Position 73
Screen Dimensions 74
Background Color 75
Rendering Mode 75
The Pointplot Rendering Mode 75
The Wireframe Rendering Mode 76
The Hidden Line Rendering Mode 77
The Shaded Rendering Mode 77
The Shaded Line Rendering Mode 78
Edges. 78
All Edges 79
Silhouette Edges 79
Outline Edges. 80
Tessellation 80
Ray Tracing 82
Scanning 82
Linear Scanning 83
Ordered Scanning 83
Random Scanning 83
A Note About Ray Tracing: Voxels. 83
Shading 83
Face Shading 84
Vertex Shading. 84
Pixel Shading 85
Feature Abstraction 86
Coarse Ray Tracing 86
Antialiasing 87
Supersampling. 88
Shadows 90
Reflections 90
Refractions 90

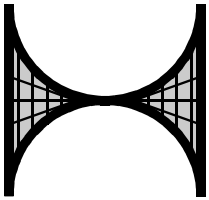
Fog 92

Animation 95

The Principles of Animation 95
Acceptable Frame Rates 95
Animation Speed 96
Animation with the Hypercosm System 96
Anims 97
Double Buffering 97
Parameter-Based Animation 98
Animation Examples 99
Mouse-Controlled Anims 102
Customizing Mouse Controls 106
Cursor Location 106
The Mouse Button 107
Mouse Clicks 109
Customizing Keyboard Controls 112
Converting Between Characters and Keycodes 113
The Time Procedure 116

Glossary 119

Index 133



CHAPTER 1
Introduction

The Hypercosm 3D Graphics System is a complete modeling, rendering, and animation system that lets you create sophisticated three-dimensional graphics with an unlimited range of behaviors. The power and flexibility of Hypercosm are made possible through the use of the OMAR programming language and its graphics extensions.

This manual provides an overview of OMAR's graphics extensions and describes how they can be used to create compelling 3D images and animations. This is neither an introduction nor a complete guide to the OMAR programming language. Such information is provided in our other available manual: *The OMAR Programming Language Reference Manual and Programming Guide*. While this manual can get you started in creating Hypercosm graphics, you cannot make full use of Hypercosm's capabilities without first gaining the basic understanding of OMAR that our other manual provides.

The 3D Graphics Process

The Hypercosm system is, in the simplest of terms, a means of producing three-dimensional graphics with a computer. Of course, images presented on a two-dimensional computer monitor can only ever truly be two-dimensional. To a computer, the difference between a typical 2D image and a so-called '3D' image is not so much in how an image *appears* as in how an image is *defined*.

A 2D image definition only contains information about a particular image: what colors appear where on a two-dimensional grid. 3D graphics, on the other hand, are derived from information about a complete three-dimensional scene. The difference is analogous to the difference between a painting and a photograph. Both appear on a two-dimensional surface, but a painting is created by drawing

directly to a 2D surface while a photograph is created by projecting a 3D scene onto a 2D surface.

The 3D graphics process is in fact very similar to photography. To produce a photograph, you must first select a subject to shoot, then decide how and where the photo should be taken, and then press a button and let the camera do the work of setting the scene to film. To produce 3D graphics, you must first define a 3D scene, then choose how the scene will be viewed, and then let the computer do the work of displaying the scene on a 2D computer screen. As in photography, no drawing ability is required. You need only define what appears in the scene—the shapes, the lights, etc.—and the computer figures out how to draw the scene by itself.

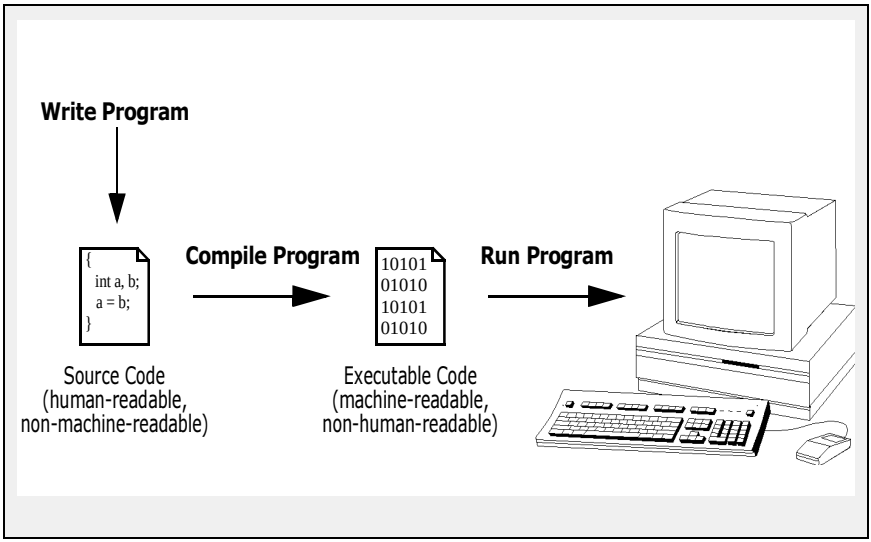
When you use the Hypercosm 3D Graphics System, you use the OMAR programming language to define 3D scenes and tell the computer how they should be viewed, then the Hypercosm system takes care of drawing the scenes for you.

How It All Works

To create 3D graphics with the Hypercosm system, you need to write programs in OMAR, then run them with a Hypercosm *interpreter*. This process is made very easy by Hypercosm's development kits, either Hypercosm Sojourner or Hypercosm Studio, which both have a built-in interpreter.

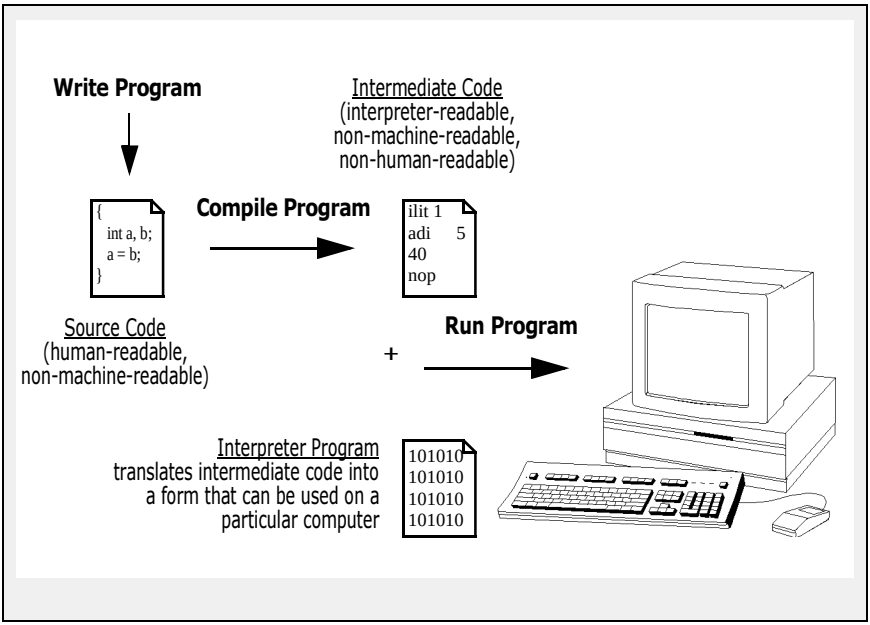
Interpreted programming languages, such as OMAR and Java, are generally similar to conventional programming languages, such as C, Pascal, and FORTRAN, but differ in their implementation. In order to run programs on a computer, all programming languages must somehow be converted, or *compiled*, into *machine language*—the 1's and 0's that computers actually understand. C, Pascal, and FORTRAN programs are all compiled directly into machine language *executable code* that runs by itself (see Figure 1-1). Because the resulting executable file is written in one particular machine language, it cannot be run on all kinds of computers, but only on computers that understand that particular machine language.

Figure 1-1: Creating and Running a Program with a Conventional Compiler



When you use OMAR or Java, however, your code is first compiled into a generic intermediate code that is not specific to any particular type of machine. In order to run this intermediate code on your computer, it must be run through a special program called an *interpreter,* which translates the generic intermediate code into machine language instructions that can be executed on your particular computer, no matter what machine language your computer understands (see Figure 1-2).

Figure 1-2: Creating and Running a Program with an Interpreter



It may at first seem like a drawback that your OMAR graphics code cannot run without the Hypercosm graphics interpreter. There are several important benefits to this approach, however.

Using the conventional approach, you must compile your program into an executable format that is particular to each kind of computer that you would like to run the program on. If you have ten programs that you would like to run on five different kinds of computers, then you would need to make a total of fifty different executables. Using the interpreter as a universal translator, however, you would only need the ten programs that have been compiled into the intermediate code and then the five different interpreters for each of the computer types that you intend to run the programs on.

As the recent explosion in Java's popularity shows, interpreted languages are extraordinarily useful in web development. Once you've created a Hypercosm graphics applet, you can put that single applet on your web page and be assured that it will run on any machine that can view the web page, provided that that machine has a Hypercosm interpreter. Currently, the Hypercosm interpreter is available to all web users in the Hypercosm 3D Player, which you can download from the Hypercosm web site (http://www.hypercosm.com) for free.

One valid criticism that can be made of interpreted languages is that interpreting code is always slower than running machine code directly, which could be a

problem for computationally-intensive graphics applications. In our case, however, this is a negligible difference because almost all graphics applications spend at least 95% of their time in redrawing the graphics and usually less than 5% of their time actually animating objects. Since all the code to compute the graphics is already compiled and optimized internally, this means that your animations would only be around 5% faster if they were completely compiled as they would be if you were writing them in C, Pascal, or Fortran.

Hypercosm Features

The Hypercosm 3D Graphics System has many features that distinguish it from other commonly used 3D graphics systems, including the following:

- **The OMAR programming language**
  Hypercosm's graphics are created using the OMAR (**O**bject-oriented **M**odeling **A**nd **R**endering) programming language. Designed as a general-purpose language like Java or C++, OMAR improves on both as a 3D graphics tool. It's generally simpler and more readable, and it has many built-in extensions designed specifically for 3D development. OMAR is also ideally suited for web development because, like Java, it is an interpreted language, resulting in platform independence and remarkably small file sizes.

- **Hierarchical geometry**
  Using OMAR, you can group surface primitives together and manipulate them as a single object. Since the hierarchy data is also used by the renderer, databases with millions or even billions of primitives can be rendered on modest machines.

- **Multiple rendering modes**
  Hypercosm provides a variety of rendering modes, from simple wireframe to full ray tracing. Included are two line rendering modes that find silhouette edges and produce clean, simple images instead of the cluttered wireframe meshes that are usually used.

- **A choice of projections**
  Several different types of camera lenses are available for special wide angle effects that cannot be achieved with standard projections.

- **Real-time/interactive shadows, reflections and transparency**
  You can produce approximate soft shadows, reflections and transparency/refraction effects at interactive rates.

- **Extensibility**
  OMAR files can import shapes, utilities, etc., from other files.

Sources for Additional Information

Using some of the advanced features of the OMAR programming language requires at least a cursory understanding of programming fundamentals, and of computer graphics. The Hypercosm documentation could never hope to cover in detail all that you need to know to write good, clean programs and compelling graphic images. However, there are a number of good reference books available if you are interested in reading more about graphics programming. Here is a small list of some that we like:

- ***Computer Graphics: Principles and Practic****e, Second Edition*
  James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes
  Addison-Wesley Publishing Company

  For most graphics issues, this may be the only book that you really need as it has good, but not overly detailed, coverage of just about everything.

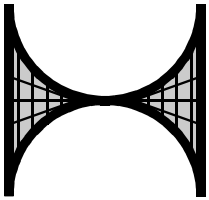
- ***Procedural Elements For Computer Graphics***
  David F. Rogers
  McGraw-Hill Book Company

  This is an older and much less extensive book than the one above. Although it focuses mainly on rendering technology, it still has a good coverage of most important graphics topics.

- ***Fundamentals of Three-Dimensional Computer Graphics***
  Alan Watt
  Addison Wesley Publishing Company

  This is a good overview of many topics in computer graphics but at a less detailed and mathematical level than others.

In addition, until a complete tutorial for the OMAR programming language is created, users with no programming experience could look into any number of introductory books or courses for other object-oriented languages such as Java or possibly C++.



CHAPTER 2
Getting Started

This chapter outlines some of the fundamentals of using the Hypercosm system, including a few guidelines for OMAR file organization. For a more complete guide to OMAR programming principles, consult *The OMAR Programming Language Reference Manual and Programming Guide*.

What You Need

To create your own graphics with the Hypercosm system, you must write your own OMAR files. Hypercosm Sojourner and Hypercosm Studio are two Hypercosm products designed specifically for OMAR file creation. You'll need one of these products before making and running new OMAR files.

Hypercosm Sojourner is Hypercosm's basic OMAR development kit, designed to demonstrate the power and ease-of-use of the Hypercosm 3D programming environment and the OMAR programming language. Sojourner has the built-in capability to interpret your OMAR code and run the program you've created. Hypercosm Sojourner is free and can be downloaded from the Hypercosm web site at http://www.hypercosm.com.

Files that you create with Hypercosm Sojourner can be exchanged with other Sojourner users. If, however, you want to create Hypercosm graphics for use in web development, you should purchase Hypercosm Studio. Like Sojourner, Studio lets you create and run your OMAR programs. However, unlike Sojourner, Studio lets you compile your OMAR programs into applets that can be uploaded to your web site.

OMAR Files

OMAR files, composed of OMAR programming code, contain all of the definitions, instructions, and data that are needed to create 3D graphics with the Hypercosm system. There are two different OMAR file types:

- **OMAR source files** are runnable files that contain essential instructions for producing OMAR applets. Source file names should carry the extension **.omar**.
- **OMAR resource files** are not runnable, but contain 'pieces' of OMAR code that may be imported for use in source files. Resource file names carry the extension **.ores**.

Hypercosm's Standard Resource Files

When you install Hypercosm Sojourner or Hypercosm Studio, you also receive Hypercosm's standard library of resource files (found in a directory titled **Ores**.) Many of the provided resources are essential for the creation of even the most basic graphics, and thus making changes to them is generally discouraged. However, all of the resource files contain readable OMAR code, and perusing them is a good way to find out about the various resources Hypercosm has to offer. There are many useful features available in the resource files that are not fully covered in this manual, including the following:

- **Shaders**—Shaders are a sophisticated way of controlling the exact characteristics of objects' surfaces. Hypercosm provides shaders that can produce transparent, cloudy, wooden, or bumpy surfaces, to name just a few.
- **Program argument handler—**The `args` type, defined in **args.ores**, is Hypercosm's standard program argument handler. Including an `args check` in your code allows you to set a variety of attributes—including window dimensions, camera orientation, rendering modes, and background color—in command line arguments or in HTML code. In Hypercosm Studio, `args check` is run *automatically* at the beginning of *every* program, so if you're using Studio, you may want to examine **args.ores** to find out more about what can be controlled with program arguments.
- **Sounds**—Hypercosm's `sound` type is defined in **native_sounds.ores**. The `sound` type can be used to import .wav files and play them in your OMAR programs.
- **Mouse cursor setting—**With the `set_cursor` procedure in **native_display.ores**, you can change the mouse cursor style to any of several common standard cursor types.
- **Web utilities—**Procedures in **native_links.ores** allow you to set the url and the status line in web browsers.
- **Collision detection and closest-point detection—**Built-in procedures defined in **native_collision.ores** can tell you whether two shapes are inter-

secting, whether a ray intersects a shape, or what part of a shape is closest to a point or plane.

- **Actors**—Actors allow you to describe high level behavior of an object and let the animation system control the details of moving the object for each frame of an animation.
- **Advanced shapes**—In addition to the primitive shapes described in this manual, Hypercosm provides code that defines hulls, extrusions, lathes, lattices, pipes, pyramids, point clouds, and several other useful shapes.
- **Random, noise, and turbulence functions**—The functions in **random.ores**, **native_noise.ores**, and **turbulence.ores** are a great way to add variety to an image or make an object's behavior more realistic.

There are a few special files, such as **native_math.ores** and **native_model.ores**, that contain `native` declarations. These declarations describe features that are built into the Hypercosm system and are therefore not written out completely in OMAR code. Since these files provide descriptions of all the native features that you can access, they can be a useful reference guide. You can actually change the default values for these features by modifying their declarations, but you should never change a native declaration in any other way. A native declaration must perfectly match the interpreter's precompiled code or else the runtime system may behave erratically.

Hypercosm's Sample Source Files

Hypercosm also provides numerous sample OMAR source files located in a directory titled **Omar**. These files have been created to demonstrate Hypercosm's many capabilities, and range in complexity from very simple to very sophisticated. Running through the example files is an excellent way to learn how to write OMAR files, especially if you already have some programming background. You can copy and paste lines of code from the sample files to your own files to experiment with different behaviors or make modifications to the code to learn how different changes affect appearance and behavior.

OMAR's Graphics Extensions

Using OMAR for graphics development would not be possible without some way of expressing graphical information in the language. For this reason, Hypercosm has added four fundamental graphics procedures to OMAR. They are syntactically identical to other OMAR procedures but have been given different names to reflect the different jobs they do.

Figure 2-1: Fundamental Graphics Extensions to OMAR

Procedure	Action
shape	Creates a shape
picture	Draws a picture
anim	Animates a sequence of pictures
shader	Returns color values for shading objects

The shape, picture, and anim procedures are like verb procedures because they carry out actions but do not return any values. The shader procedure is like a question procedure because it returns a value. Shaders are a feature beyond the scope of this manual's coverage. The other graphics procedures will be covered in greater detail in later sections.

3D Coordinates

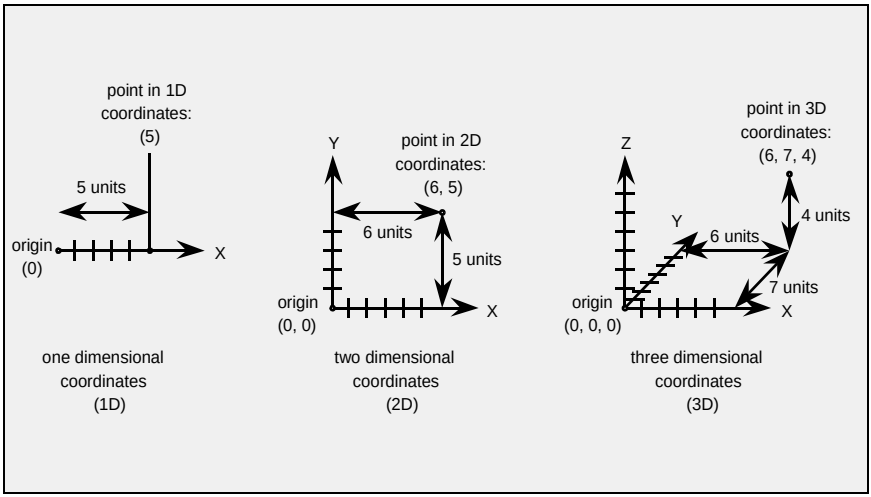
Although it is not necessary to know any algebra or trigonometry to use the Hypercosm system, it is necessary to be comfortable with the three dimensional coordinate system that is used to describe the geometry of the forms you wish to create.

While Hypercosm graphics do appear on a 2D computer screen, it's important to remember that the Hypercosm world is truly three-dimensional. The computer screen in the Hypercosm world is not so much a canvas on which images are drawn as it is a camera's view into a vast open space. Objects in that space can be moved up or down, left or right, forward or backwards, and how such changes appear on screen depends completely upon how the camera is positioned. Objects appear large on the screen if the camera is placed relatively close to them, and objects appear to be very small if the camera is relatively far away. Any objects outside the camera's field of view won't appear on the screen at all.

In order to describe various shapes and positions in the 'vast Hypercosm space,' we need some means of specifying locations in it. To do this, Hypercosm uses the standard mathematical 3D coordinate system. The system works by setting a single point in space as the *origin*. Three orthogonal (perpendicular) axes—X, Y, and Z—run through the origin and can be used to measure distance from the origin. All other points in space can then be specified by their X, Y, and Z coordinates. The origin itself has the coordinates <0 0 0>, because it lies at

the center of each of the axes. Other points may have positive or negative coordinates depending on where they're positioned with respect to the axes.

Figure 2-2: Coordinate Systems



Here's an illustration of using 3D coordinates to specify a location: If all the streets in your neighborhood were orthogonal, you could tell someone how to get to your apartment by telling them to go 3 blocks east, 5 blocks north, and up 2 floors. This would put your apartment coordinates at <3 5 2> (assuming you gave the directions at the origin). Note that the order of the coordinates is important, because if the person went 2 blocks east, 5 blocks north, and 3 stories up, they would end up in a different location.

Using the Hypercosm system, you may generally assume that the origin will appear at about the center of your screen; the X-axis will run from left to right (right being the positive direction); the Y-axis will run from you into the screen (forward being the positive direction); and the Z-axis will run up and down (up being the positive direction). However, remember that how directions appear on screen depends on where the 'camera' is placed. If, for example, the camera is placed somewhere up the Z-axis and is pointed towards the origin, then the Y-axis could run up and down.

A Note About Units

You might be wondering exactly what size the units are in the 3D coordinate system. If you place an object one unit away from the origin, how far will that be on-screen? If you want to place a sphere 3/4 of the way across the graphics window, how many units should you move it?

Units in the Hypercosm world are in fact completely relative, and how they appear on screen depends completely upon camera placement and perspective. Just as in real life, a unit that is close appears larger than a unit that is far away. You can imagine the standard unit to be any length you choose—a millimeter, a yard, or a light year—but how big a unit appears on screen still depends on how close it is to the 'camera.'

Camera placement is determined in OMAR code by the variable called `eye`. As a default, the eye is placed at the coordinates <10 -30 20> and is pointed towards the origin. If you create a shape at the origin using default viewing parameters, the shape should probably be between one and thirty units in width in order to fit in the graphics window and not be too small. However, changing any of the viewing parameters (as discussed in Chapter 4: *Viewing*) or moving the shapes you create can change how big a single unit appears.

In the end, the best method of handling 3D units is to follow, once again, the practice of photography. When you want to determine how a scene will look in a photograph, you don't calculate beforehand what objects will appear where in the photo. Instead, you simply look in the camera's view finder, and if you don't like what you see, you can move items in the scene, move the camera, zoom in or out, focus, etc. Similarly, when you want to know how a Hypercosm 3D scene will look on-screen, you simply *run the program*. If you don't like what you see, you can move the 'virtual camera,' zoom in or out, or change the shapes in the scene.

The Basic OMAR Graphics File

To create even the most basic OMAR graphics source file, you must at least include the following:

Header Statement

At the beginning of every OMAR source file, you must write a *header statement* that tells the computer which procedure to run first. A header statement is composed of the keyword `do`, followed by a procedure name, and ends with a semicolon, like this: `do example;`. In a graphics source file, the procedure named is either a `picture` or `anim`. Running a `picture` produces a still image. Running an `anim` produces an animation, which is essentially a series of `picture`s being continuously redrawn.

Note: the major difference between source files and resource files is that a resource file does not contain a header statement, and therefore cannot be compiled and run by itself.

Include Statements

An *include statement* is used to import OMAR code from other OMAR files. Producing even the most basic of graphics requires that OMAR code is imported from some of Hypercosm's standard resource files. All of the essential graphics resource files can be included by including the single file **3d.ores**. Therefore, almost all OMAR graphics files should contain the statement include "3d.ores"; after the header statement. Other include statements may be used to include any of Hypercosm's OMAR files or any of your own, whether they are regular source files or resource files.

Declarations

At a bare minimum, an OMAR graphics file requires a picture declaration. Running a picture is the only way to display something on the screen using the Hypercosm system. Even if the header statement runs an anim, the anim must in turn run a picture in order to produce any image. Inside a picture are shape statements that indicate what objects are to appear in the image. Hypercosm can only display a shape if it is called inside a picture declaration.

Figure 2-3: Basic Picture Declaration

```
picture <name> is
      <shapes>
end;
```

If an OMAR source file is to run an animation, it must contain an anim declaration as well. For more discussion about how sophisticated animations can be written, see Chapter 6: *Animation*.

A Simple Example

The file below contains the bare minimum requirements to produce graphics. It simply draws a shaded sphere. Text that appears after a pair of slashes // is considered a *comment* and is ignored by the computer.

Listing 2-1: A Simple Picture

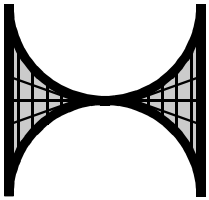
```
do sphere_picture;          // The header statement: instructs the computer to run 'sphere_picture.'

include "3d.ores";          // An include statement: imports OMAR code from '3d.ores.'

picture sphere_picture is   // A picture declaration: specifies what appears in the picture.
      distant_light;        // Calls for a distant light to be cast on objects in the picture.
      sphere;               // Calls for a sphere to be drawn in the picture.
end;
```

A More Interesting Example

Only a little more code is needed to produce much more interesting graphics. The following example file creates an interactive animation that allows you to spin a cube around, zoom in and out, and pan the camera around by pressing mouse buttons and dragging. More information about how this works is given in Chapter 6: *Animation*.

Listing 2-2: An Interactive Cube

```
do cube_anim;

include "3d.ores";
include "anims.ores";              // 'anims.ores' is needed to use 'mouse_controlled_picture.'

picture cube_picture is
        distant_light from <-1 -2 3>;
        block;
end;

anim cube_anim is
        mouse_controlled_picture cube_picture;  //'mouse_controlled_picture' is a special procedure
                                                // that takes a picture and makes it an interactive
                                                // animation.
end;
```



CHAPTER 3
Modeling

In computer graphics, the term *modeling* refers to the process of defining, or building all of the shapes in a scene. Once you have defined a scene, it may be turned into a picture through the process of *rendering*, which is covered in Chapter 5: *Rendering*.

Objects in the real world are composed of many different shapes and surface types, and it is challenging to try to model such variety on the computer. Some shapes are surprisingly easy to represent with a computer, while some are remarkably difficult. A molecule, for example, may be modeled as a group of spheres. This is easy to do on a computer because a sphere is a basic geometric shape and is therefore easily defined. Something like a person, however, is a much more difficult shape to model because people come in all sorts of shapes with lots of slightly differing curves. Computer graphics is still not capable of realistically modeling many shapes, such as clouds, fire, or hair. However, while challenging, it's usually possible to figure out an adequate way of modeling most objects with basic easily-modeled shapes.

Elements of Modeling

The three basic components of modeling that this chapter covers are:

- **Geometry**. The precise size, shape, and location of each shape in a scene must all be specified.
- **Lighting**. The lighting of a scene determines how bright objects appear and how they are shaded. You must define where light is coming from, how bright it is and what color it is.
- **Surface Attributes**. In a lighted, three-dimensional scene, the color of an object varies across its surface. How the color varies depends on the

material the object is made of. Metal, plastic, and paper all reflect light differently. Thus, modeling also involves specifying an object's material or texture.

Primitive Shapes

The first important aspect of modeling is defining a scene's geometry. All the shapes you create, no matter how complex, must be built up from basic geometric shapes that the computer knows how to draw. These basic shapes are known as *primitive* shapes.

The primitive shapes are divided into four groups:

- Quadrics
- Planar primitives
- Non-planar primitives
- Non-surface primitives

In the Hypercosm system, lights are also considered to be a special class of primitive shapes, but will be discussed later.

Quadrics

There are six different types of quadrics:

- Sphere
- Cylinder
- Cone
- Paraboloid
- Hyperboloid1
- Hyperboloid2

These quadrics are all surfaces of revolution. Their surfaces are defined by the types of curves known as conic sections. The sphere, cylinder, and cone shapes should be very familiar to you.

The hyperboloid1 is the shape of a cooling tower of a nuclear reactor and is also roughly the shape of the bell of a musical instrument such as a trumpet. The paraboloid and hyperboloid2 are gumdrop-shaped surfaces that you can use to model such things as domes or satellite dishes.

Planar Primitives

There are six different types of planar primitives:

- Plane
- Disk
- Ring
- Triangle
- Parallelogram
- Polygon

As their name implies, these primitives are all flat, as though cut out of an infinitely thin sheet of paper. The plane is most useful for representing the ground (computer graphics people believe in a flat earth). In the raytracing render mode, the plane is rendered as an infinite plane. In the other rendering modes, the plane is represented by a finite planar grid.

The disk and ring are very useful for capping the ends of quadric primitives like the cylinder or cone to make them look solid. The triangle, parallelogram, and polygon are all useful when constructing shapes with a lot of flat faces, so they are commonly used in architectural and mechanical design.

Non-Planar Primitives

There are six different types of non-planar primitives:

- Torus
- Block
- Shaded triangle
- Shaded polygon
- Mesh
- Blob

The torus is a doughnut shape. The block is a simple block. The shaded_triangle and shaded_polygon are like tiny pieces of a curved surface. If you put a large number of them together, with their curvature matching at the edges, they can be used to approximate curved shapes. The mesh is an efficient way of representing any general shape or surface as a collection of tiny facets. Shapes with unusual curved surfaces such as automobiles, aircraft, or human figures are often represented this way. The blob can be used to model such shapes as water droplets, molecules, and organic shapes that flow together and would be hard to construct otherwise.

Non-Surface Primitives

There are three different types of non-surface primitives, **points**, **lines**, and a **volume** primitive. The points and lines are called non-surface primitives because they are infinitely tiny and therefore do not define a surface. Since they are infinitely tiny, points are represented on the computer screen as individual pixels, and lines are represented as contiguous, one-pixel-wide groups of pixels. The volume primitive does not have an explicitly defined surface but is instead specified as a density field.

Using the Primitives

The following pages illustrate the declarations of all the geometric primitives as they appear in the file, native_model.ores. Shapes are actually called in picture declarations or in other shape declarations. To reset a shape's parameters, you must use the keyword with when you call the shape. To learn more about the syntax of procedure calls, consult *The OMAR Programming Language Reference Manual and Programming Guide*.

Example: Using Primitive Shapes

```
picture example is
      sphere;              // This sphere uses the sphere's default parameters

      cone with            // This cone uses default parameters, except for the parameters reset below.
            end1 = <-2 0 0>;
            end2 = <2 0 0>;
            umax = 180;
      end;
end;
```

Figure 3-1: The Sphere

```
shape sphere with
      vector center = <0 0 0>;
      scalar radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
      scalar vmin = -90;
      scalar  vmax = 90;
end;
```

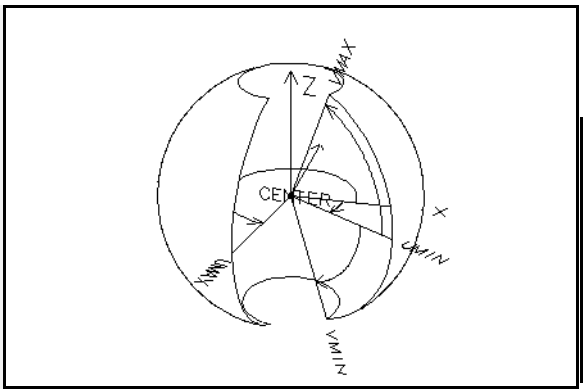


Figure 3-2: The Cylinder

```
shape cylinder with
      vector end1 = <0 0 1>;
      vector end2 = <0 0 -1>;
      scalar radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

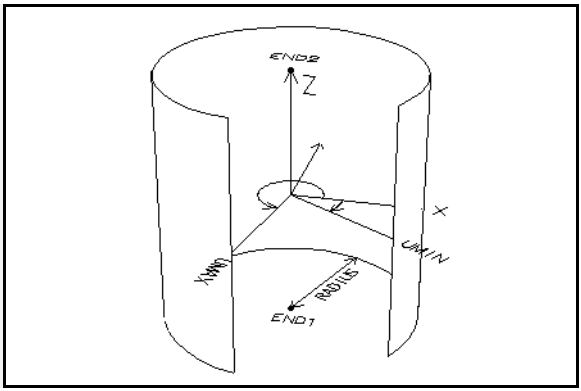


Figure 3-3: The Cone

```
shape cone with
      vector end1 = <0 0 1>;
      vector end2 = <0 0 -1>;
      scalar radius1 = 0;
      scalar radius2 = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

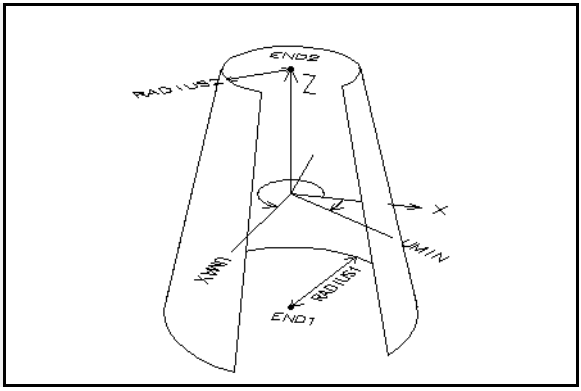


Figure 3-4: The Paraboloid

```
shape paraboloid with
      vector top = <0 0 1>;
      vector base = <0 0 -1>
      scalar radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

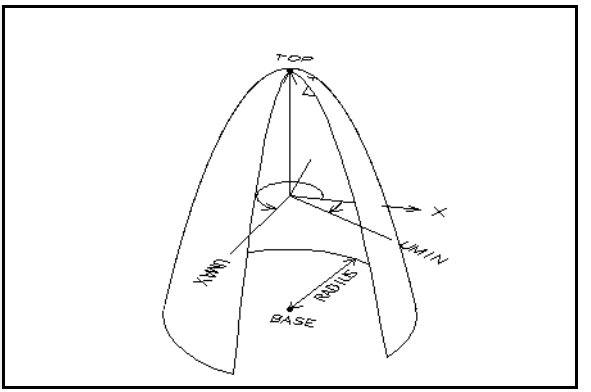


Figure 3-5: The Hyperboloid1

```
shape hyperboloid1 with
      vector end1 = <0 0 1>;
      vector end2 = <0 0 -1>;
      scalar radius1 = .5;
      scalar radius2 = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

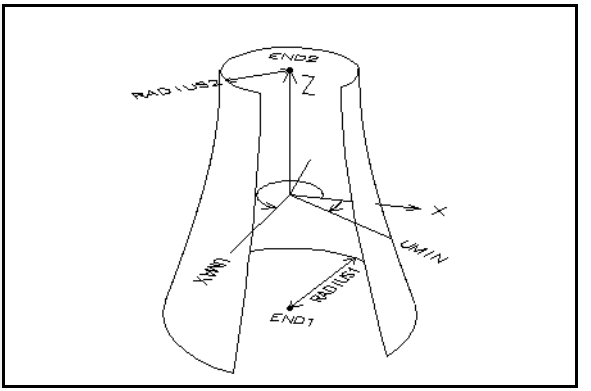


Figure 3-6: The Hyperboloid2

```
shape hyperboloid2 with
      vector top = <0 0 1>;
      vector base = <0 0 -1>;
      scalar radius = 1;
      scalar eccentricity = .5;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

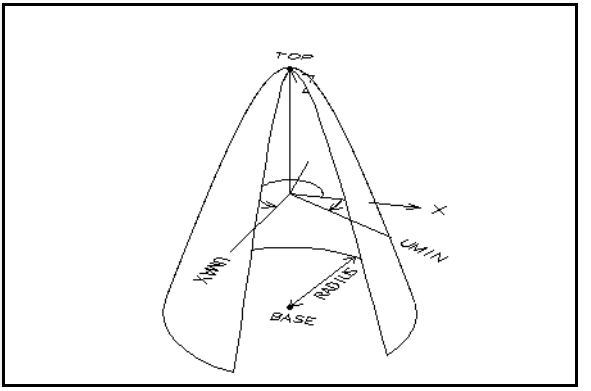


Figure 3-7: The Plane

```
shape plane with
      vector origin = <0 0 0>;
      vector normal = <0 0 1>;
end;
```

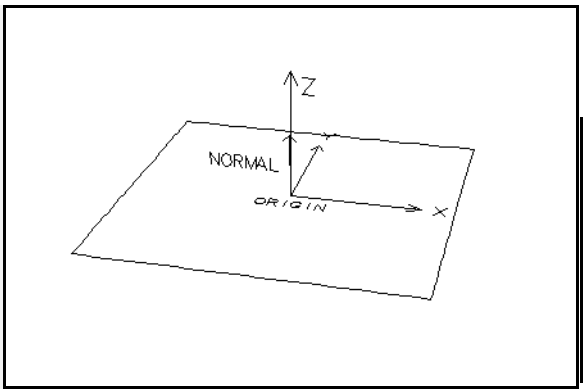


Figure 3-8: The Disk

```
shape disk with
      vector center = <0 0 0>;
      vector normal = <0 0 1>;
      scalar radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

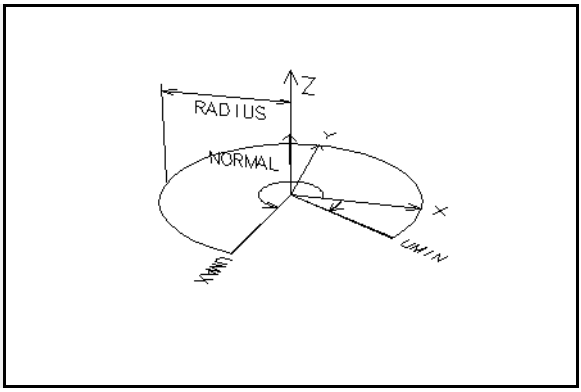


Figure 3-9: The Ring

```
shape ring with
      vector center = <0 0 0>;
      vector normal = <0 0 1>;
      scalar inner_radius = .5;
      scalar outer_radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
end;
```

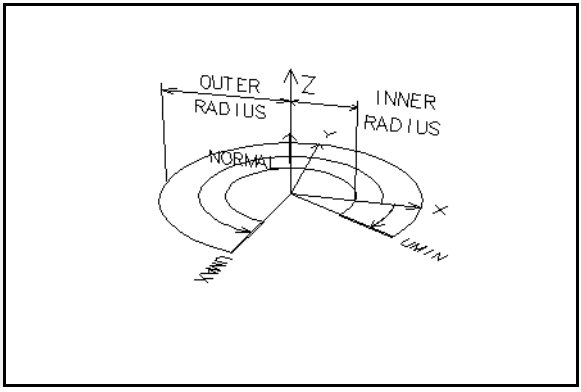


Figure 3-10: The Triangle

```
shape triangle
        vector vertex1;
        vector vertex2;
        vector vertex3;
end;
```

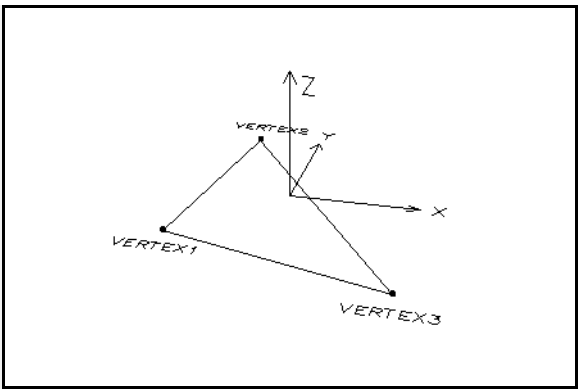


Figure 3-11: The Parallelogram

```
shape parallelogram with
        vector vertex =<-1 -1 0>;
        vector side1 = <2 0 0>;
        vector side2 = <0 2 0>;
end;
```

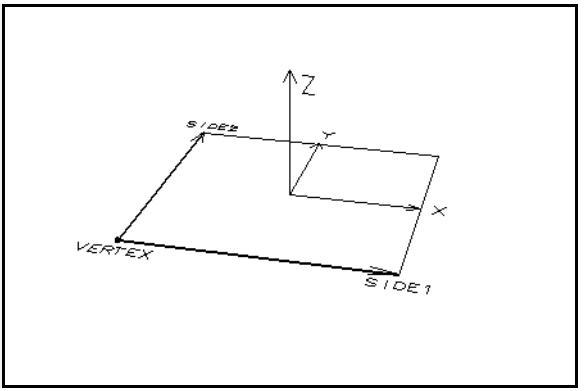


Figure 3-12: The Polygon

```
shape polygon
        vector vertices[];
with
        vector texture[];
end;
```

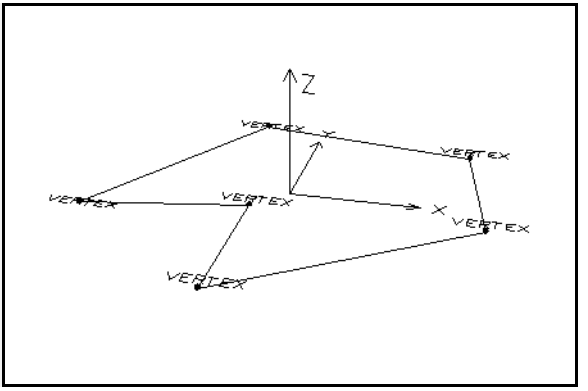


Figure 3-13: The Torus

```
shape torus with
      vector center = <0 0 0>;
      vector normal = <0 0 1>;
      scalar inner_radius = .5;
      scalar outer_radius = 1;

      // Sweep parameters
      scalar umin = 0;
      scalar umax = 360;
      scalar vmin = 0;
      scalar vmax = 360;
end;
```

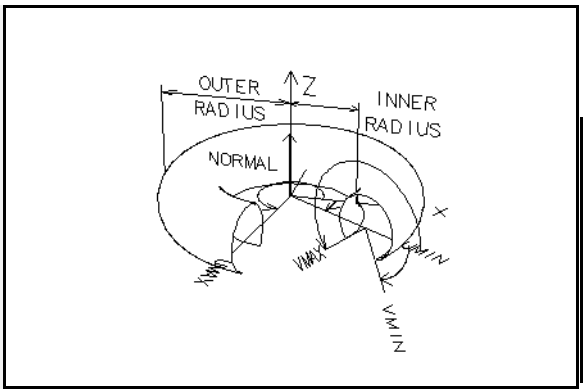


Figure 3-14: The Block

```
shape block with
      vector vertex=<-1 -1 -1>;
      vector side1 = <2 0 0>;
      vector side2 = <0 2 0>;
      vector side3 = <0 0 2>;
end;
```

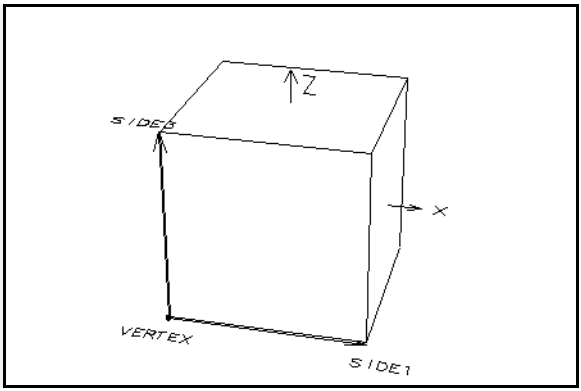


Figure 3-15: The Shaded_Triangle

```
shape shaded_triangle
      vector vertex1;
      vector vertex2;
      vector vertex3;

      vector normal1;
      vector normal2;
      vector normal3;
 end;
```

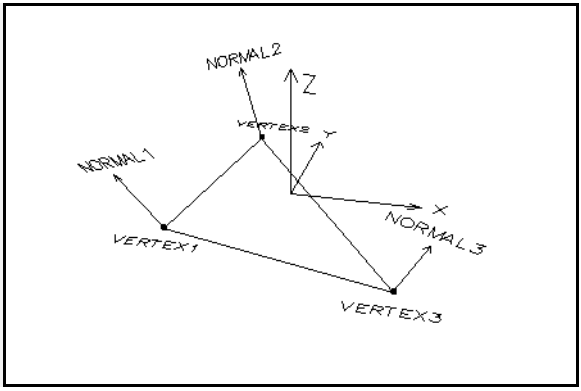


Figure 3-16: The Shaded_Polygon

```
shape shaded_polygon
        vector vertices[];
        vector normals[];
with
        vector textures[];
end;
```

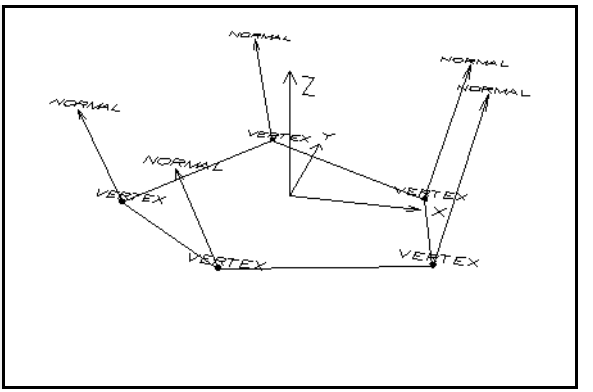


Figure 3-17: The Mesh

```
shape mesh
        vector vertices[];
        integer edges[,];
        integer faces[];
with
        vector normals[];
        vector textures[];
        boolean smoothing is true;
        boolean mending is false;
        boolean closed is false;
end;
```

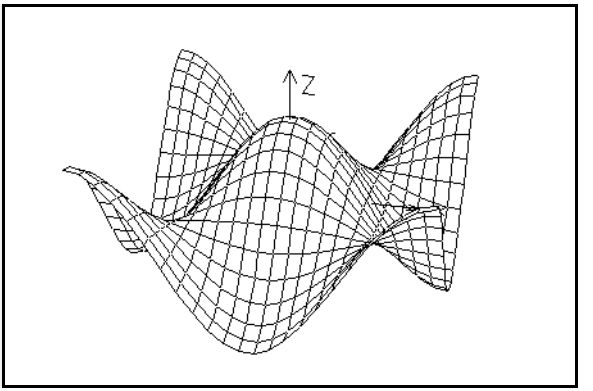


Figure 3-18: The Blob

```
shape blob
        vector centers[];
with
        scalar radii[];
        scalar strengths[];
        scalar threshold = .5;
end;
```

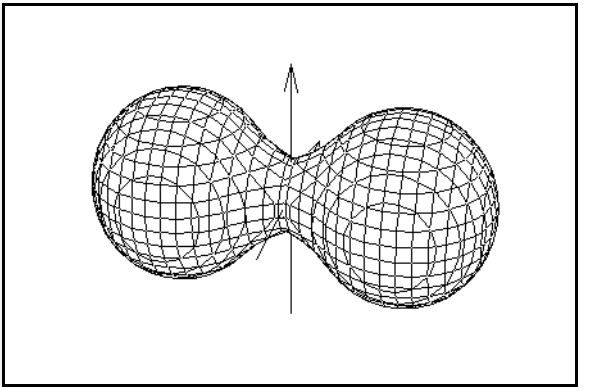


Figure 3-19: Points

```
shape points
      vector vertices[];
end;
```

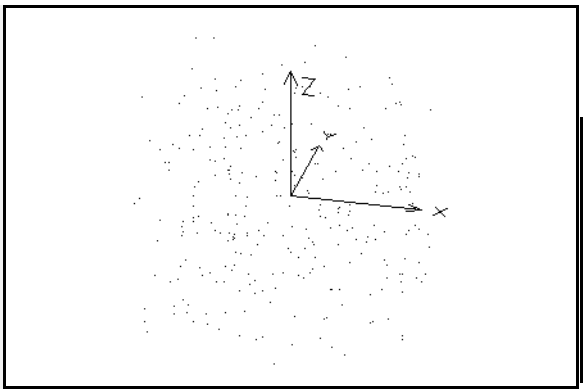


Figure 3-20: Line

```
shape line
      vector vertices[];
end;
```

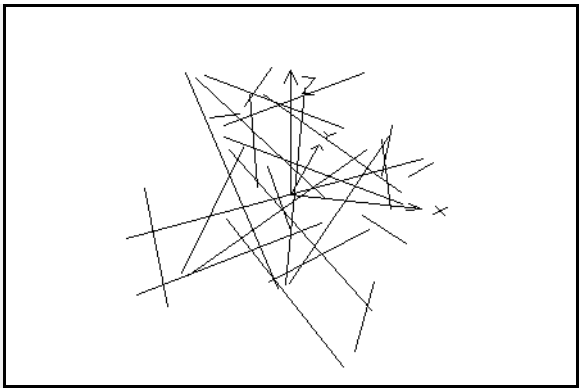
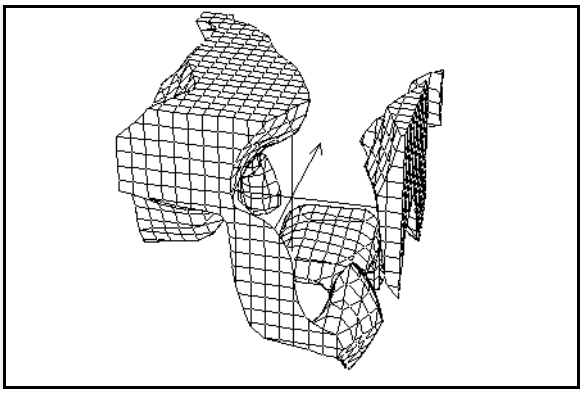


Figure 3-21: Volume

```
shape volume
      scalar densities[,,];
      in form of vector vertices[,,]
            is none;
with
      vector vertex[,,];
      scalar threshold = .5;
      boolean capping is true;
      boolean smoothing is true;
end;
```



A Closer Look at the Primitives

Some aspects of the primitives are complicated and require further explanation.

Sweeps or Partial Surfaces

Primitives that are derived from circles can be cut along longitude lines and sometimes along latitude lines. This results in a partial surface called a *sweep*. The surfaces that allow this operation are the quadrics, the disk and ring, and the torus.

Longitude and Latitude

On a globe, the longitude lines are the lines that run from the north pole to the south pole. The latitude lines are the ones that circle the globe parallel to the equator. You can describe a sweep in terms of minimum and maximum longitude and latitude.

The longitude restraints are the parameters `umin` and `umax`. Longitude is measured in degrees and usually ranges between 0 and 360. Angles greater than 360 or less than 0 will wrap around automatically by adding or subtracting 360 degrees.

The latitude restraints are `vmin` and `vmax`. Latitude usually ranges from -90 to 90 with 0 degrees at the equator. Latitudinal angles that are out of range will automatically wrap around by adding or subtracting 180 degrees. The longitude and latitude are measured in the local coordinates of the primitive.

Figure 3-22: Longitude and Latitude

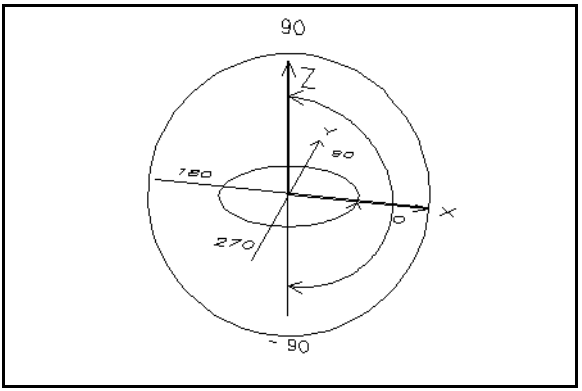
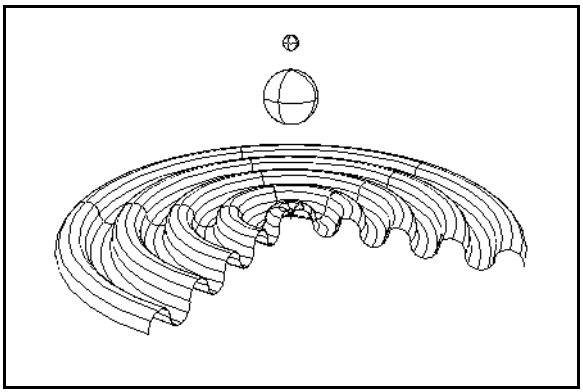


Figure 3-23: A Circular Wave Using Sweeps



Listing 3-1: Using Sweeps of Torii to Create a Circular Wave

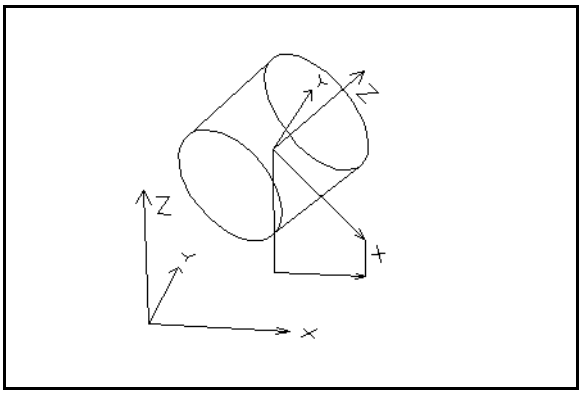
```
do circle_wave_picture;

include "3d.ores";

shape circle_wave with
      integer steps = 4;
      scalar radius = 1, ucut = 360;
is
      scalar stepsize = radius / steps;

      sphere with
            radius = stepsize; umax = ucut;
            vmin = 0; vmax = 90;
      end;

      for integer i = 1 .. (steps - 1) do
            if even i then
                  torus with
                        inner_radius = i * stepsize; outer_radius = (i + 1) * stepsize;
                        umax = ucut; vmin = 0; vmax = 180;
                  end;        // crests
            else
                  torus with
                        inner_radius = i * stepsize; outer_radius = (i + 1) * stepsize;
                        umax = ucut; vmin = 180; vmax = 360;
                  end;        // troughs
            end;
      end;
end;        // circle_wave

picture circle_wave_picture with
      eye = <1 -2 1>;
is
      sphere with center = <0 0 .5>; radius = .1, color = aqua; end;                  // big droplet
      sphere with center = <0 0 .7>; radius = .025; color = aqua; end;                // little droplet
      circle_wave with steps = 10; ucut = 270; end;
end;        // circle_wave_picture
```

Measuring Longitude

Longitude is always measured as degrees of counterclockwise rotation around the Z-axis, starting in the positive X direction. This means that if you want to create a quarter-cylinder that sweeps from the positive X direction to the positive Y direction, you should use a cylinder primitive and set its umin to 0 and its umax to 90.

However, if a primitive is rotated so that its axis of rotation does not correspond to the real Z-axis (see Figure 3-24) then the primitive's longitude measurement is determined by its *local* coordinates. The *local* Z-axis of a primitive is always simply aligned with its axis of rotation. The local X-axis of a primitive is computed by finding the vector that is closest to the real X direction and is still perpendicular to the local Z-axis of the primitive. The local Y-axis of the primitive is the direction that is perpendicular to the local Z-axis and the local X-axis, which are both defined above.

Figure 3-24: Local Coordinates of a Surface of Rotation



The Mesh Primitive

A mesh is a group of polygonal facets that can be automatically shaded in a way that makes it appear to be smoothly curved. The computer shades a typical polygon by using the polygon's *normal*—the direction perpendicular to the polygon's surface—to determine how much light is reflected off the polygon's surface. The normals of a mesh's facets, however, are automatically adjusted in such a way that when shading is done, the facets appear to be curved instead of flat.

You specify a mesh by defining an array of vertices, an array of edges, and an array of faces. The vertices are vectors. The edges are pairs of indices to the vertices. The face array contains indices to edges. The index, 0, is used to indicate the end of a face. The face array: [4 6 2 0 2 8 5 3 0] defines two faces because there are two lists of integers delimited by zeros.

You must specify the edges of a face in a consistent order, meaning that they must all go clockwise, or all go counterclockwise. The positive direction of an edge is defined by the order in which the vertices are given. To specify that you want an edge that goes in the opposite direction, put a negative sign in front of the index to the edge. For example, if you have the face [4 -3 2 1 0], that means that edge 4 is followed by the reverse of edge 3, followed by edges 2 and 1. The format is illustrated by the following example:

Figure 3-25: Cube Vertices

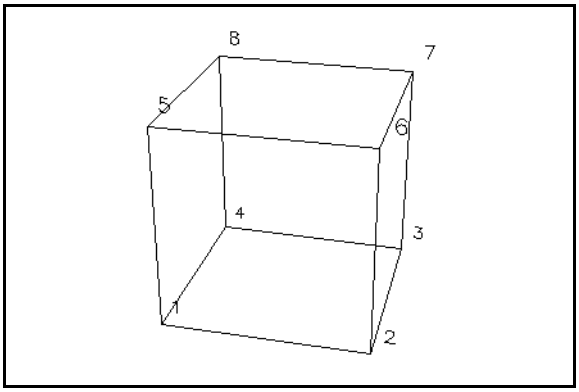


Figure 3-26: Cube Edges

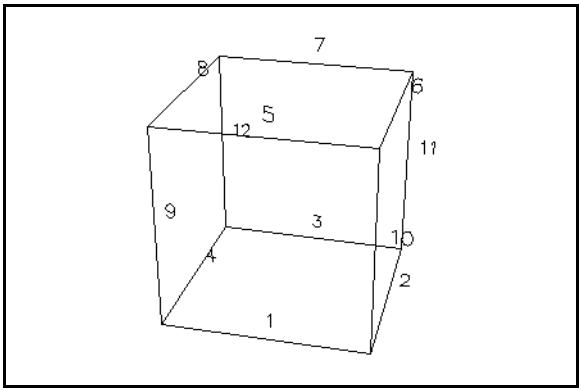
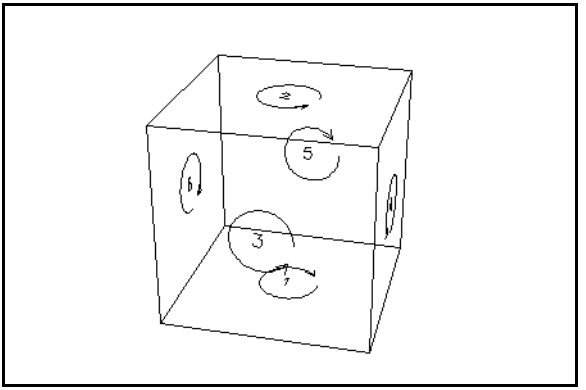


Figure 3-27: Cube Faces



Example: Using a Mesh

```
shape cube is
        mesh
                // vertices
                [<-1 -1 -1> <1 -1 -1> <1 1 -1> <-1 1 -1>
                 <-1 -1 1> <1 -1 1> <1 1 1> <-1 1 1>]

                // edges
                [[1 2][2 3][3 4][4 1][5 6][6 7]
                [7 8][8 5][1 5][2 6][3 7][4 8]]

                // faces
                [-4 -3 -2 -1 0 5 6 7 8 0 1 10 -5 -9 0
                2 11 -6 -10 0 3 12 -7 -11 0 4 9 -8 -12 0];
end; // cube
```

Figure 3-28: The Cube Mesh

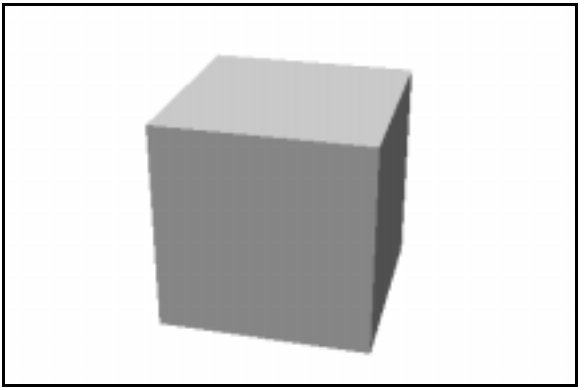
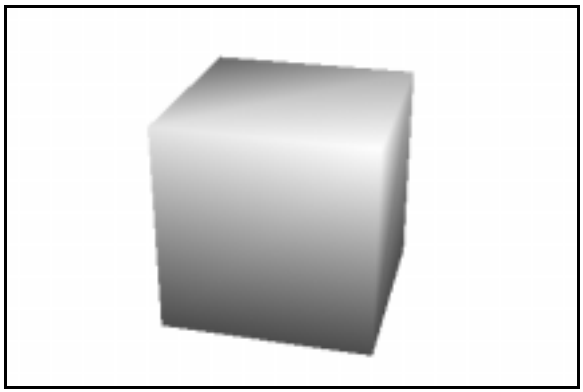


Figure 3-29: The Cube Mesh with Smoothing



The Shaded Triangle & Shaded Polygon

The shaded triangle and shaded polygon are like pieces of a mesh in that they are flat shapes that can be shaded as if they were curved. The mesh, however, knows which facets are adjacent to which other facets, so it can automatically compute the normals to perform smooth shading. With the shaded polygon and shaded triangle, however, you must specify the normals at the vertices yourself.

The normals must all be specified to point in the same direction relative to the plane of the polygon. If they don't point in the same direction, very strange shading effects occur.

Figure 3-30: Normals in the Same Direction

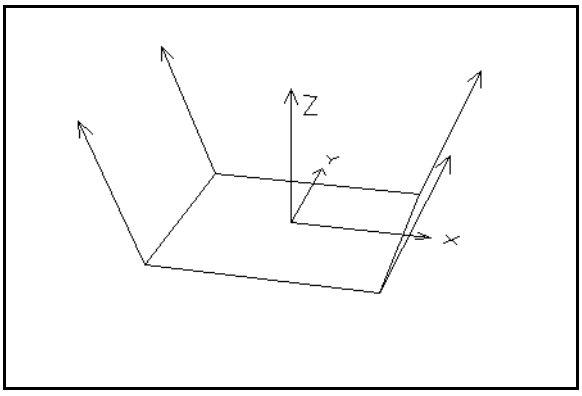
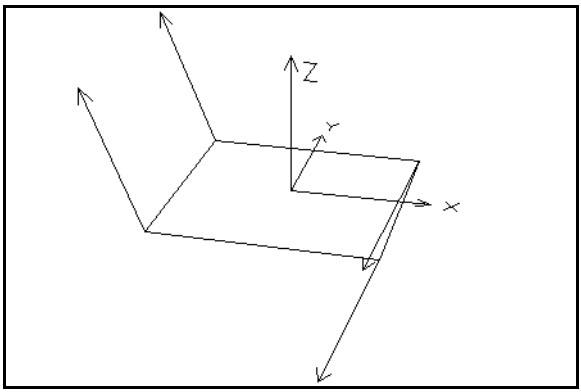


Figure 3-31: Normals Not in the Same Direction



Remember that the mesh, shaded triangle, and shaded polygon are all means of *approximating* curved surfaces. They will not yield great results if the number of polygons used to approximate a surface is very low. If the shading looks strange, you might need to use more polygons to approximate the curve.

The Volume Primitive

The volume primitive is useful for scientific visualization and for modeling shapes that would be difficult to define explicitly in terms of their vertices, edges and faces. A volume primitive contains information about a volume's density at regular points throughout the volume. What appears on-screen is the portion of the volume that has density values greater than the volume's threshold parameter.

The volume primitive normally takes the shape of a unit cube. To make a volume of some other rectangular shape, you can simply scale the cube along its axes to attain the desired shape. For non-rectangular volumes, you have to specify the shape of the volume by an optional array of vertices. For example, if you want to visualize the humidity in the atmosphere of the earth, you would have to warp the volume to a spherical region. If the densities are meant to denote the temperature in an engine cylinder, you would warp the volume to a cylinder. If you are interested in seeing an example of how this is done, you

can look at Hypercosm's example file, **volume.omar**, found in the **Physics** directory.

Figure 3-32: Volume Warped to a Sphere

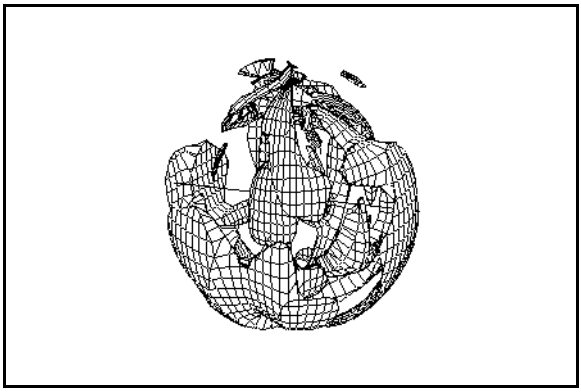
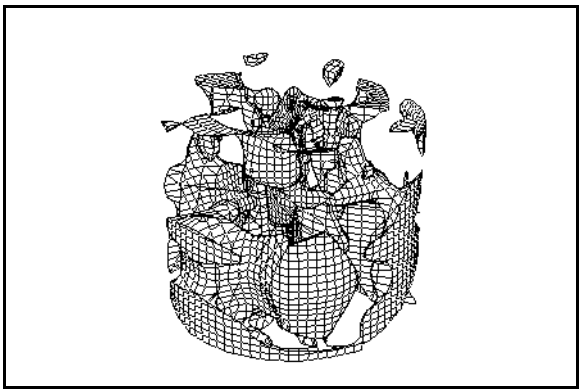


Figure 3-33: Volume Warped to a Cylinder



Lighting

In order to model a shaded scene, you must not only specify the geometry of the scene, but the lighting conditions as well. Hypercosm's renderer then computes how the light interacts with the geometry of the scene.

Lighting Primitives

Lights are treated in many respects as shapes. They may be scaled, moved, and rotated just like ordinary shapes. Lights are different, however, because the lights themselves can't be seen. Only their influence on the scene around them gives away their presence. It is as if you can make light appear spontaneously out of empty space.

There are three types of lights: **distant**, **point**, and **spot** lights.

- **Distant lights** are used most often because they're easy to set and similar to sunlight.
- **Point lights** are like a light bulb and only illuminate an area around themselves.
- **Spot lights** emit cones of light that dim at the edges like flashlights or a car's headlights.

Lights may be any color (even negative colors—darkon emitters!) and there may be any number of lights, although beware that many lights will cause the program to render very slowly.

The definitions of the lighting primitives are illustrated in the following sections. Their definitions can also be found in **native_lights.ores**.

Ambient Light

If you walk outside on a sunny day and look at the 'dark side' of an object such as a building which is not receiving any light from the sun, you notice that the surface is not completely dark, like the dark side of the moon. Why is this? It's true the surface receives no direct light from the sun, but it receives indirect sunlight from all around—from the lit up sky and from light reflected off of other objects.

Hypercosm approximates this kind of light by saying that an entire scene is bathed in a constant uniform light coming from all directions. This is what is known as ambient light.

Figure 3-34: The Ambient Light Variable & Default

```
vector ambient = <.4 .4 .4>;
```

The default ambient light is set at <.4 .4 .4> (in **native_render.ores**). You could give sunny outdoor scenes a more realistic appearance by setting the ambient term to a slightly bluer color such as <.3 .3 .4> because blue sky emits a diffuse bluish glow. If you're modeling space pictures, then it is a good idea to set the ambient term to something like <.1 .1 .1> or even <0 0 0> since there is very little scattered light in space.

Figure 3-35: The Distant Light

```
shape distant_light
      from vector direction
                  = <0 0 1>;
with
      scalar brightness = 1;
      color type color = white;
      boolean shadows is true;
end;
```

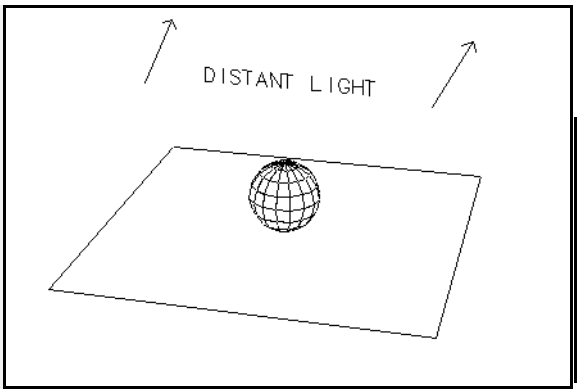


Figure 3-36: The Point Light

```
shape point_light with
      scalar brightness = 1;
      color type color = white;
      boolean shadows is true;
end;


// Note: a point_light is by default
// located at the origin.
```

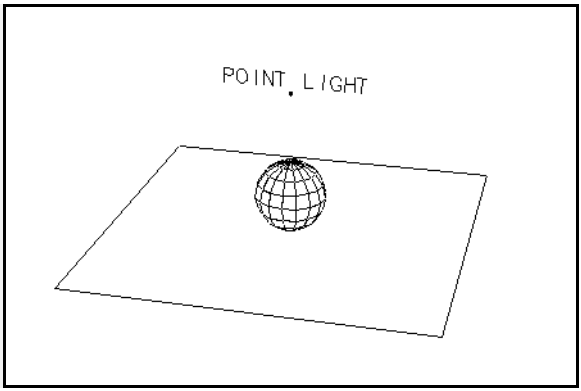
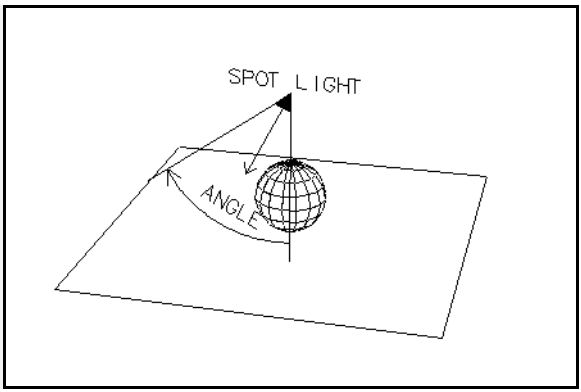


Figure 3-37: The Spot Light

```
shape spot_light
      towards vector direction
                  = <0 0 1>;
with
      scalar brightness = 1;
      scalar angle = 90;
      color type color = white;
      boolean shadows is true;
 end;

// Note: a spot_light is by default
// located at the origin.
```



The following example program draws a swinging spot light. It demonstrates the usage of distant lighting, spot lighting, and ambient lighting.

Listing 3-2: A Swinging Spot Light

```
do spot_anim;

include "3d.ores";

picture swinging_spot with
      scalar time = 0;
      eye = <0 -15 10>;
      ambient = <.3 .3 .4>;                    // Ambient light is slightly bluish.
is
      scalar s = cos(time * 5);                // Swing light back and forth.
      vector spot_location = <0 0 2>;
      vector spot_direction = <s 0 -1>;

      distant_light from <.3 -.5 1>with        // overhead light
            brightness = .5;
      end;

      spot_light towards spot_direction with   // spot light inside of cone
            move to spot_location;             // use 'move to' to place spot light
            brightness = 6;
      end;

      cone with                                // shade for spot light
            end1 = spot_location - (normalize spot_direction) * .5;
            end2 = spot_location + (normalize spot_direction) * .5;
            radius1 = 0; radius2 = .5; color = yellow;
      end;

      plane with                               // floor
            magnify by 5; color = white;
      end;
end;        // swinging_spot

anim spot_anim with
      double_buffer is on;
is
      scalar s = 0;
      while true do
            rotate by 1 around <0 0 1>;        // This rotates the entire scene.
            swinging_spot with time = s; end;
            s = itself + 1;                    // These numbers can be changed to adjust
                                               // animation speed.
      end;
end;        // spot_anim
```

Hierarchical Modeling

You could conceivably build any shape that you wanted just by creating a long list of primitives. However, a more practical and intuitive approach would be to define simple shapes out of primitives and then combine these simple shapes into more complex shapes. This is what is known as hierarchical modeling.

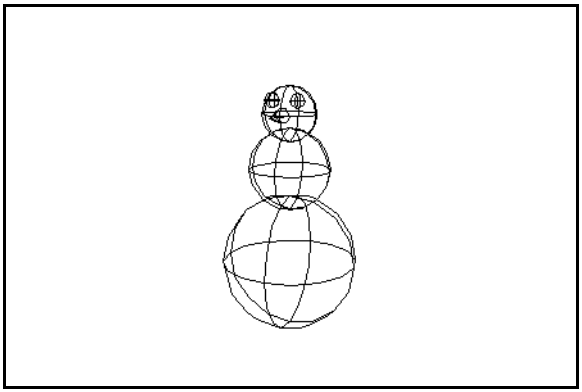
A group of shapes, or an *aggregate shape*, can be treated just like a primitive shape. It can be moved, rotated, magnified, or manipulated with any other operation that you can perform on a primitive shape. Once an aggregate shape is defined, it will not appear anywhere in a picture unless you specifically create an instance of the shape in either a picture or another shape that appears in the picture. This is known as *instantiation*. Aggregate shapes are declared as follows:

Figure 3-38: Aggregate (Hierarchical) Shape Declaration

```
shape <name> is
      <shapes>
end;
```


All aggregate shape definitions must have a name so they can be uniquely identified. Sometimes the hardest thing about writing good OMAR description files is thinking up good names for all the shapes that are described.

Figure 3-39: A Hierarchically Defined Snowman



Relative Transformations

Relative transformations include actions such as *moving*, *rotating*, *magnifying*, *stretching*, and *scaling* a shape. Once you define a shape, you can create several instances of the shapes, all with different locations, orientations, and dimensions. More than one transformation may be applied to a single instance. For example, one shape may be rotated, then stretched, then moved. Note that the order of the transformations is important. An object that has been stretched, then rotated looks different from an object that has been rotated, then stretched.

Relative transformations specify the size or position of a shape relative to the way that it was originally defined in its declaration. For instance, if a shape is set to 2 units high in its declaration, you can make it 6 units high by magnifying by 3. To make the shape 6 units high regardless of how it was defined, you'd

Listing 3-3: Description of a Snowman Using Hierarchy

```
do mr_snow;

include "3d.ores";

shape snowman is
      // Body:
      color = white;
      sphere with center =<0 0 1>; end;
      sphere with center = <0 0 2.4>; radius = .6; end;
      sphere with center = <0 0 3.2>; radius = .4; end;

      // Nose:
      cone with
            color = orange;
            end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>;
            radius1 = .1; radius2 = 0;
      end;

      // Eyes:
      color = charcoal;
      sphere with center = <-.2 -.2 3.4>; radius = .1; end;
      sphere with center = <.2 -.2 3.4>; radius = .1; end;
end;        // snowman

picture mr_snow with
      eye = <3 -8 4>;
      lookat = <0 0 2>;
is
      snowman;              // Here, instead of instantiating many primitive shapes,
                            // we need only instantiate 'snowman.'
end;        // mr_snow
```

need to use an absolute transformation, which is covered later. Relative transformations are defined in **transforms.ores**.

Figure 3-40: Using Relative Transformations

```
<shape name> with
      <relative transformations>
end;
```

Figure 3-41: The Move Transformation

```
verb move
      to vector location;
end;
```

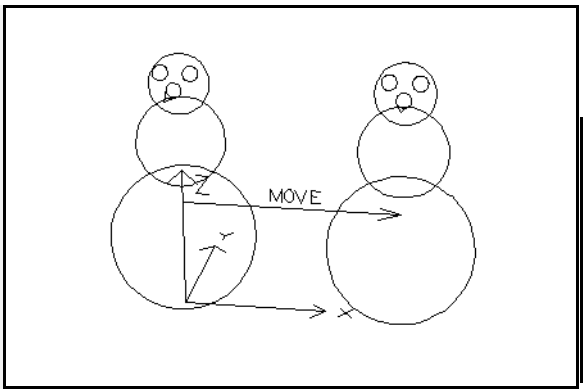


Figure 3-42: The Magnify Transformation

```
verb magnify
      by scalar s;
      about vector point = <0 0 0>;
end;
```

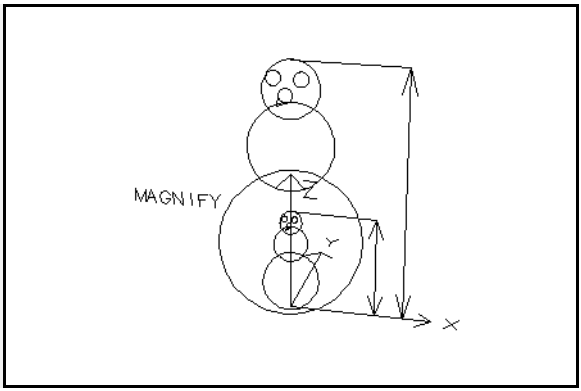


Figure 3-43: The Rotate Transformation

```
verb rotate
      by scalar angle;
      around vector axis;
      about vector point = <0 0 0>;
 end;
```

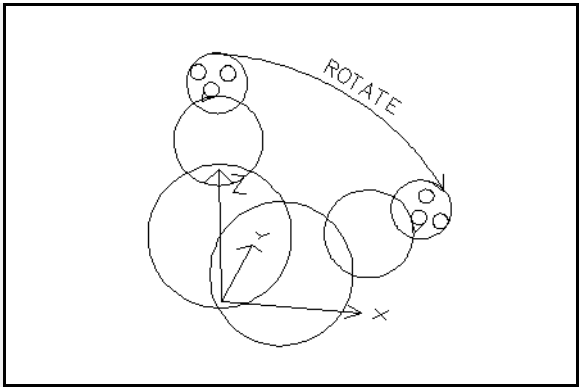


Figure 3-44: The Scale Transformation

```
verb scale
        by scalar s;
        along vector v;
        about vector point = <0 0 0>;
end;
```

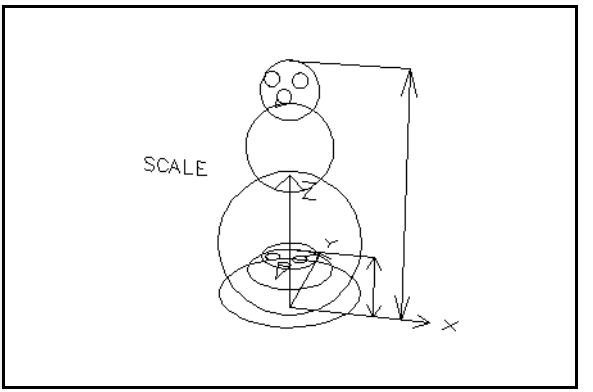


Figure 3-45: The Stretch Transformation

```
verb stretch
        by scalar s;
        along vector v;
        about vector point = <0 0 0>;
end;
```

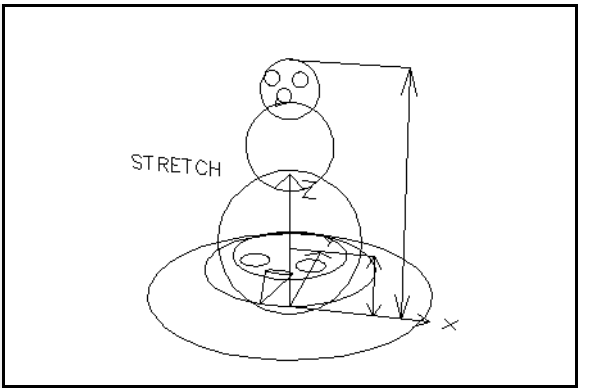


Figure 3-46: The Skew Transformation

```
verb skew
        from vector point1;
        to vector point2;
        about vector point = <0 0 0>;
end;
```

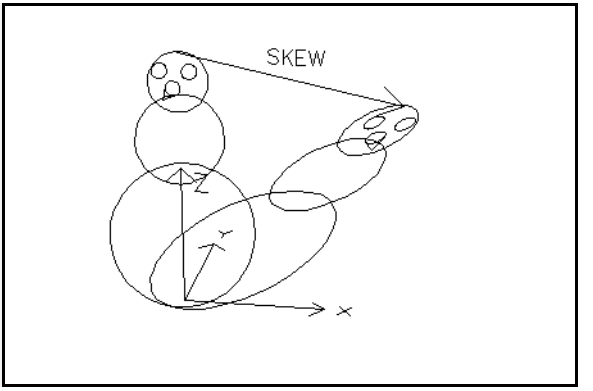


Figure 3-47: The Slant Transformation

```
verb slant
        by scalar angle;
        about vector point = <0 0 0>;
with
        vector x_axis = <1 0 0>;
        vector y_axis = <0 1 0>;
end;
```

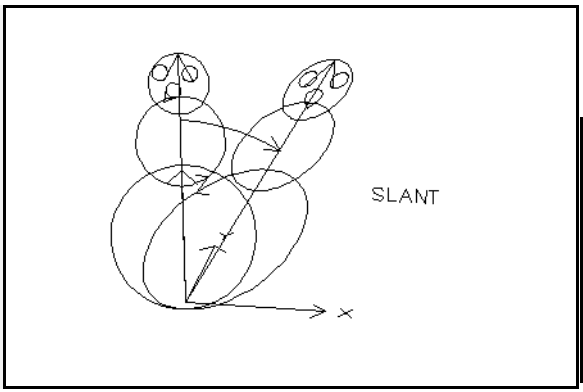


Figure 3-48: The Direct Transformation

```
verb direct
        from vector v1;
        to vector v2;
        about vector point = <0 0 0>;
end;
```

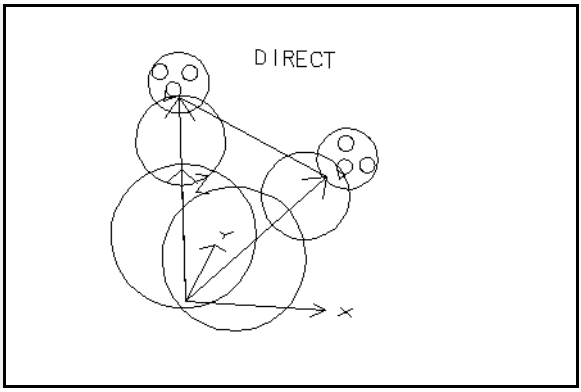


Figure 3-49: The Orient Transformation

```
verb orient
        from vector v1;
        to vector v2;
        about vector point = <0 0 0>;
end;
```

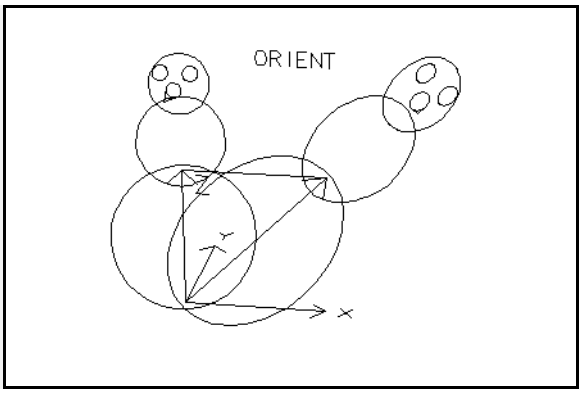
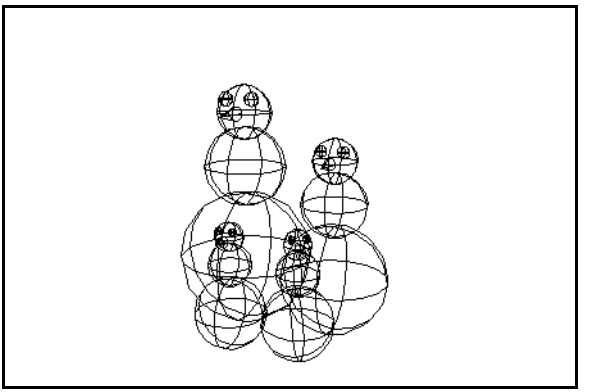


Figure 3-50: A Hierarchically Defined Snow Family



Listing 3-4: A Family of Snowmen Using Hierarchy and Transformations

```
do snow_family;

include "3d.ores";

shape snowman is
        // Body:
        color = white;
        sphere with center = <0 0 1>; end;
        sphere with center = <0 0 2.4>; radius = .6; end;
        sphere with center = <0 0 3.2>; radius = .4; end;

        // Nose:
        cone with
                color = orange;
                end1 = <0 -.3 3.2>; end2 = <0 -.8 3.2>;
                radius1 = .1; radius2 = 0;
        end;

        // Eyes:
        color = charcoal;
        sphere with center = <-.2 -.2 3.4>; radius = .1; end;
        sphere with center = <.2 -.2 3.4>; radius = .1; end;
end;        // snowman

picture snow_family with
        eye = <3 -8 4>;
        lookat = <0 0 2>;
is
        snowman with move to <-.7 0 0>; end;                                    // Dad
        snowman with magnify by .8; move to <.7 0 0>; end;                      // Mom
        snowman with magnify by .5; move to <-.5 -1 0>; end;                    // Kids
        snowman with  magnify by .5; rotate by 30 around <0 0 1>; move to <.5 -1 0>; end;
end; // snow_family
```

The Transformation Stack

The previous examples showed how to use transformations to modify a particular instance of an object relative to its local frame of coordinates. There may be cases where you wish to do a more complicated series of transformations, such as a series of objects each of which is offset or rotated in relation to the previous object, instead of in relation to the local coordinates. If you understand the way the transformation stack works, you can do these types of operations more easily.

The Current Transformation State

Each time you make a new shape, it is positioned relative to the current transformation. Each time you begin to define a new object or a new instance, a new state is created and when you are finished, the state is returned to the state that existed before.

You can think of this as a stack of transformation states with the current transformation on top of the stack. When you begin a new object, you push a new state onto the stack and when you are through defining that object, you pop its state off the stack to return to the previous one.

Transforming a Series of Shapes

Usually, transformations are placed only within the scope created by a shape instance (i.e. after the with keyword in an instantiation). In that case the transformations affect only that particular instance of an shape. However, transformations can be applied anywhere in a list of shape instances. If you place a transformation inside of a shape declaration, but outside of a single instance, then all the instances that follow the transformation will be affected because you have changed the state in which they are declared.

Example: Transforming a Series of Objects

```
shape thing is
      <instances1>     // These shapes are unaffected by the transformations

      <transformation1>

      <instances2>     // These shapes are affected by transformation1

      <transformation2>

      <instances3>     // These shapes are affected by both transformations
end;
```

Nesting Transformations

Sometimes, you want to have the transformations work in a hierarchical manner, with each transformation relative to the previous one. You can do this by nesting transformations.

For example, say you want to define a finger, with each joint bent by a certain angle relative to the previous joint. Also, you want each successive joint to be smaller. To accomplish this, the second joint is instantiated within the first joint, after the first joint's transformations, so it is moved and scaled relative to the first joint.

Listing 3-5: Nested Transformations - The Finger

```
do finger_picture;

include "3d.ores";

shape joint with
      scalar radius = .5, length = 1;
is
      color = pink;
      sphere;
      color = flesh;
      cone with end1 = <0 0 0>; end2 = <0 0 length>; radius1 = 1; radius2 = radius; end;
end;        // joint

shape finger with
      scalar curl = 0, joint_radius = .9, joint_length = 2.3;
is
      joint with                                                    //First knuckle
            radius = joint_radius; length = joint_length;
            rotate by curl around <1 0 0>;

            joint with                                              //Second knuckle
                  radius = joint_radius; length = joint_length;
                  magnify by joint_radius;  move to <0 0 joint_length>;
                  rotate by curl around <1 0 0>;

                  joint with                                        //Third knuckle
                        radius = joint_radius; length = joint_length;
                        magnify by joint_radius; move to <0 0 joint_length>;
                        rotate by curl around <1 0 0>;

                        sphere with                                 //Fingertip
                              center = <0 0 joint_length>;
                              radius = joint_radius;
                              color = pink;
                        end;
                  end;
            end;
      end;
end;        // finger

picture finger_picture is
      distant_light from <1 -1 1>;
      rotate by 90 around <0 0 1>;
      finger with curl = 0,  with curl = 20,   with curl = 40; end;
end;        // finger_picture
```

Absolute Transformations

The main difficulty with relative transformations is that sometimes you may want to manipulate an object without having to look at the dimensions that it was defined with.

For example, say that you include a model of a car in your picture and want to set it onto a road. Assume that the road is one unit wide and you want the car to fit nicely onto the road. If you were to use the relative transformations, you would need to examine the model of the car to see how big it is and then magnify it by the (road width / car width). However, if you use the absolute transformations, you can just specify that the car should be a certain width, no matter how big or small the car was originally defined to be.

Specifying Absolute Transformations

Absolute transformations are specified a little differently than relative transformations. Since the absolute transformations need to know the dimensions of the object in order to work, they must take place *after* the object has been created. The relative transformations are listed in a block of statements beginning at the keyword with which is executed before the object is actually created by the computer. The absolute transformations must occur after the object has been built, so they are listed in a later block of statements beginning with the keywords return with.

Figure 3-51: Specifying Absolute Transformations

```
<object name> with
      <relative transformations (optional)>
return with
      <absolute transformations>
end;
```

Mixing Relative and Absolute Transformations

It is also possible to mix relative transformations and absolute transformations by listing them both in the same block after the return keyword. For example, you may want to scale the size of the model car to fit onto the road using an absolute transformation and then follow this with a rotation which is a relative transformation to make the car appear to be turning off the road.

Implementation of Absolute Transformations

The absolute transformations are implemented by having the program return the size of the object through the state variables, **origin, x_axis, y_axis,** and **z_axis**.

The size of the object is actually reported as a bounding box, which may be slightly bigger than the actual object because a box must have extra room inside to encompass a non-square object. Therefore, if you specify the dimensions of

an object using an absolute transformation, the object is always guaranteed to lie inside of the dimensions, but may be slightly smaller than the bounds that you specify. If the object fits nicely inside of a box, then the absolute transformations will more closely reflect the actual dimensions of the object itself.

Absolute transformations are defined in the file **abs_trans.ores**.

Figure 3-52: The Dimensions Transformation

```
verb dimensions
        of vector v;
        about vector point
                        = <0 0 0>;
end;

{dimensions resizes a shape to fill
the bounding box specified by the
coordinates in v.}
```

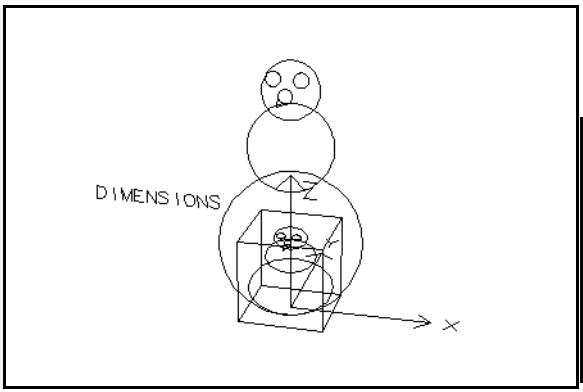


Figure 3-53: The Size Transformation

```
verb size
        of scalar s;
        enum axis is x_axis, y_axis,
                        z_axis;
        along axis type axis;
        about vector point
                        = <0 0 0>;
end;

{size resizes a shape, preserving
relative proportions, to fit in a
bounding box with length s
along axis.}
```

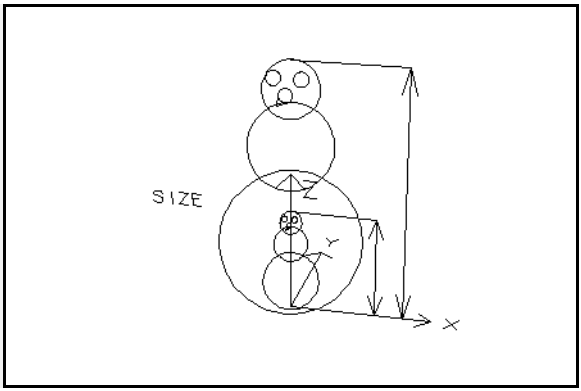
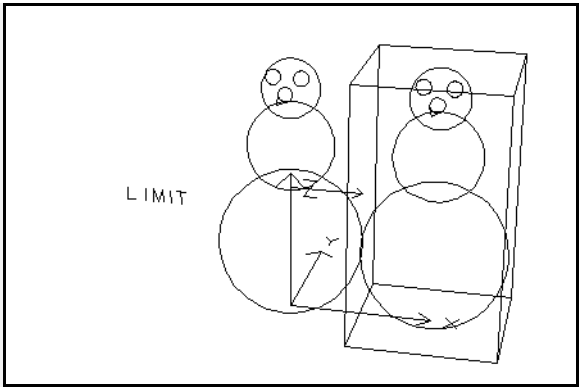


Figure 3-54: The Limit Transformation

```
verb limit
        enum limit is x_min, x_max,
                y_min, y_max,
                z_min, z_max;
        limit type limit;
        to scalar s;
end;

{limit moves a shape, preserving its
size, to the limit set by s.}
```



Color

Because of certain characteristics of the human eye, almost all imaginable colors can be created by mixing red, green, and blue in various proportions. While computer monitors may appear to display a full spectrum of colors, they actually use only three: red, green, and blue.

In the Hypercosm system, all colors are specified using RGB values, which indicate exact amounts of red, green, and blue in ranges from 0 to 1.

Assigning Color to Shapes

To assign a color to a shape, you must use the global color variable (which is declared in **native_render.ores**).

Figure 3-55: The Global Color Variable

```
color type color;
```

The global color variable is both called color and is of the color type. The color type works the same as the vector type because colors, like vectors, are specified by three numbers. For example, to create a red sphere, enter:

Example: Assigning Color to a Shape

```
sphere with
    color = <1 0 0>;        // Sets the RGB values so that R = 1, G = 0, B = 0
end;
```

The system for assigning colors to shapes works in a hierarchical manner similar to the way that the transformation stack works. Each time you begin a new shape definition or instance, you enter a new context. The color that gets assigned to a primitive is the color that is closest to the shape in the context of the hierarchy. This makes it easy to specify the colors of special portions of the shape and leave the rest of the shape 'unpainted.' Then, when you assign a color to an instance of the shape, that color only applies to the 'unpainted' parts of the shape that don't already have a color previously assigned to them. For the parts of the shape that have already been assigned a color, the previously assigned color takes precedence.

Predefined Colors

For your convenience, Hypercosm provides a number of predefined colors. The following chart shows the names of the available colors and the RGB values that each color represents, as defined in **colors.ores**.

Table 3-1: The RGB Values of Hypercosm's Predefined Colors

Name	RGB Values	Name	RGB Values
white	1 1 1	raspberry	1 0 .5
black	0 0 0	pink	1 .6 .7
grey	.5 .5 .5	flesh	.9 .7 .6
red	1 0 0	beige	1 .9 .85
green	0 1 0	lime_green	.5 .8 0
blue	0 0 1	olive	.4 .5 0
cyan	0 1 1	evergreen	0 .4 .25
magenta	1 0 1	teal	0 .75 .6
yellow	1 1 0	aqua	0 .75 .75
orange	1 .5 0	turquoise	0 .7 .9
brown	.35 .2 0	sky_blue	.6 .75 1
gold	.9 .8 .3	azure	.35 .3 .75
maize	.8 .7 0	lavender	.8 .6 .9
brick	.5 .15 0	purple	.6 .15 .75
rust	.7 .3 .1	violet	.5 0 .9
charcoal	.2 .2 .2	eggplant	.3 0 .2

In addition to these colors, Hypercosm provides two procedures, light and dark, that make colors lighter or darker by mixing them with white or black. Using these procedures, you can make colors such as light yellow or dark green. The procedures may also be called repeatedly so you can create colors such as light light yellow or dark dark green.

Example: Using Predefined Colors & Color Modifiers

```
cone with color = red; end;
plane with color = light teal; end;
torus with color = dark dark blue; end;
```

Materials

In the line-drawing modes, the colors that appear in a picture are simply the colors that were assigned to the respective shapes. In the shaded modes, however, the apparent surface color varies because of lighting, shadows, and other factors. How the color varies across an object's surface also depends on the material the object is made of. Hypercosm allows you to specify different material types for each shape you make.

The three main predefined materials are chalk, plastic, and metal. If a material is not explicitly assigned, shapes are shaded as though they were chalk; that is, they appear to have a non-glossy, non-reflective surface such as chalk or paper. A plastic shape is shinier than chalk, and a metal shape is also shiny, but much more reflective, so it is darker where it does not directly reflect light.

To assign a material to an object, change the global variable material when you make an instance of the object. For example, to create a metal sphere, write:

Example: Assigning a Material to a Shape

```
sphere with
      material is metal;
end;
```

To set the color of a material, use this format:

Example: Assigning a Colored Material to a Shape

```
sphere with
      material is metal colored light blue;
end;
```

Another useful material type is constant_color, which effectively nullifies shading effects. This is very useful when you are modeling a shape that should be luminous, like a light bulb or a TV screen. When you use a constant color material, you *must* also specify what color the material should be.

Example: Assigning a Constant Color to a Shape

```
sphere with
      material is constant_color orange;
end;
```

Coloring Precedence

The global material variable takes precedence over the color variable when objects are drawn. If neither a color nor a material is assigned, the default color for each primitive will be used.

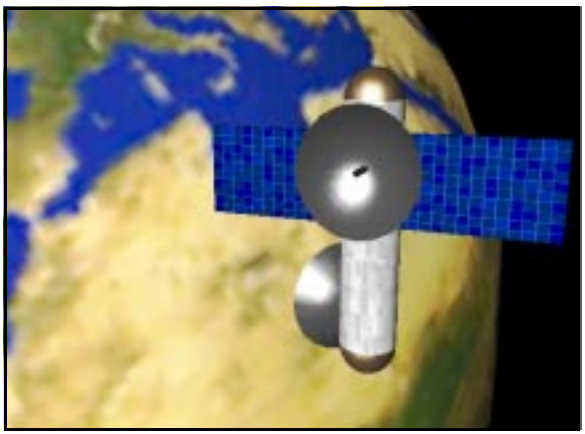
Table 3-2: Order of precedence of colors and materials from highest to lowest

material of object
material of parent object
color of object
color of parent object
default color for that primitive

Textures

The easiest way to add interesting, complex features to the surfaces of your shapes is to use *textures*. When a shape is textured, an image from an image file is mapped onto it. Using textures, you can map an image of the earth onto a sphere, or map an image of a crowd across the seats of a virtual stadium. You can create a virtual room inside of a wallpapered box, or move a scene 'outside' by surrounding everything with a sky-textured sphere.

Figure 3-56: A Scene Using Textures



To create a texture in OMAR, you create a new material type that incorporates an image file. There are a few different ways to do this, but the easiest way is to use the `poster` keyword. The following listing demonstrates how to use the `poster` keyword to create a textured image of a satellite.

Listing 3-6: A Textured Satellite Scene Using the poster *Keyword*

```
do satellite_anim;

include "3d.ores";
include "anims.ores";          // "anims.ores" is needed for the mouse_controlled_picture procedure.

material type solar_cell_material is metal poster "satellite.jpg";  // Creates a texture shaded like metal.
material type panel_material is chalk poster "panels.jpg";          // Creates a texture shaded like chalk.

shape earth is
      sphere with
            magnify by 200;
            material is chalk poster "earth.jpg";       // Sets the material to a texture shaded like chalk.
      end;
end; // earth

shape dish is
      cylinder with radius = .1; end1 = <0 0 0>; end2 = <0 -2 0>; end;
      paraboloid with
            top = <0 -1 0>; base = <0 -2 0>; radius = 2;
            material is plastic colored charcoal;
      end;
end; // dish

shape satellite is
      cylinder with
            scale by 4 along <0 0 1>;
            material is panel_material;                 // Sets the material to the panel texture.
      end;
      block with
            vertex = <-6 0 0>; side1 = <12 0 0>; side2 = <0 .2 0>; side3 = <0 0 3>;
            material is solar_cell_material;            // Sets the material to the solar cell material.
      end;

      // capping bulges:
      sphere with center = <0 0 4>; vmin = 0; end;
      sphere with center = <0 0 -4>; vmax = 0; end;

      // radio dishes:
      dish with move to <0 0 2>; end;
      dish with move to <0 0 -3>; rotate by -135 around <0 0 1>; end;
end; // satellite

picture satellite_picture is
      distant_light from <0 -10 3>;
      satellite with material is metal colored light orange; end;  // Note: because of coloring precedence
                                                                   // rules, this line will only change the material of the
                                                                   // "capping bulges" and the cylinder in the "dish."
      earth with move to <-200 250 -100>; end;
end; // satellite_picture

anim satellite_anim with
      facets = 10;                  // facets, eye, and lookat are parameters discussed in later chapters.
      eye = <5 -20 5>;
      lookat = <-3 0 0>;
      ambient = <.1 .1 .1>;         // Ambient lighting should be low because this is a space scene.
is
      mouse_controlled_picture satellite_picture;  // This lets you use mouse controls to change the view.
end; // satellite_anim
```

Note that a poster must use one of the standard materials described in the *Materials* section above: chalk, plastic, metal, or constant_color. When you make a textured material, the image you specify is *blended* with the material's color. Thus, color and shading affect textured materials as well as normal materials. For this reason, when you use a constant_color together with a texture, you'll probably want to use white, so that all the colors of the texture are pure. However, you could just as easily blend an image with a non-white color to produce a differently hued or darker texture.

In addition to the poster keyword, there is also a painted keyword, and a textured keyword. The painted keyword takes an image type parameter and is useful if you want to use the same image for multiple textures, but only wish to load it once.

Example: Using the Image Type

```
image type star_image named "stars.jpg";      // Now, "stars.jpg" has been loaded into star_image.
material type white_stars is constant_color white painted star_image;
material type red_stars is constant_color red painted star_image;
```

The textured keyword takes a texture type parameter. The texture type is an OMAR subject defined in **native_textures.ores**. To understand how to use the texture type properly, you may need to review the sections on object-oriented programming in *The OMAR Programming Language Reference Manual and Programming Guide*. Using a texture type allows you to turn on or off special texturing features, such as mipmapping and interpolation, whose properties are beyond the scope of this manual.

Figure 3-57: The Texture Type and its Methods

```
enum texture_status is invalid, disabled, unloaded, loaded;

subject texture does
        native verb new
                using image type image;
        with
                boolean interpolation is on;
                boolean mipmapping is on;
                boolean wraparound is on;
        end; // new

        native texture_status type question status;
        native verb finish_loading;
end; // texture
```

The texture type's most useful feature is that it allows you to check the current status of a particular texture. The Hypercosm system uses *deferred texturing*, which means that it draws all graphics immediately, even if textures haven't been loaded yet, and then includes the textures in the graphics as soon as they're loaded. Sometimes textures take a while to load, especially if they use large image files.

For this reason, you may want to delay drawing a textured shape, or color it differently, until its texture is 'ready' to be drawn.

You can use the status method to determine whether a particular texture has been loaded, or is still unloaded. You can also use the finish_loading method to prevent deferred texturing, and force a texture to load before your program proceeds with drawing.

In some cases, a texture may never be rendered at all. The image file may be invalid, which means it doesn't exist, cannot be found, or is not a recognized image file type. Texturing might also be disabled. This occurs on computers that do not support texturing, and many computers still do not. You can use the status method to determine if a texture is invalid or disabled, and then adjust your graphics accordingly.

Example: Using the Texture Type

```
image type lightwood named "lightwood.jpg";
texture type lightwood_texture using lightwood;
texture type scratched_texture using (new image type named "scratched.jpg") with
        interpolation is on;
        mipmapping is off;
end;

material type wood_veneer is (plastic textured lightwood_texture colored brown);
material type brushed_metal is (metal textured scratched_texture);

if scratched_texture status is disabled then            // Check whether this machine supports
        block with material is metal colored grey;      // texturing or not.
else
        scratched_texture finish_loading;               // Prevent deferred texturing
        if scratched_texture status is invalid then
                block with material is metal colored grey;
        else
                block with material is wood_veneer;
        end;
end;
```

The Hypercosm system can create textures from GIF, JPEG, or Targa image files. To be able to view textures, your machine must have 3D graphics hardware support. If you're producing a web applet, remember that even if your computer can render textures, other computers running the same OMAR applet may not be able to render textures.

Procedural Modeling

Most graphics systems describe each object as a simple list of the component parts. Since Hypercosm uses a complete programming language, you can define shapes at a much higher level than this, in a process known as *procedural modeling.*

Procedural modeling allows you to describe objects by giving the instructions necessary to generate the component parts of the object instead of explicitly listing the parts. Procedural modeling is especially effective for defining objects which have a lot of repetition or self-similarity. For example, to describe a picket fence, instead of simply listing the locations of each picket, you could instead describe the fence by writing a snippet of code to repeatedly create the pickets a certain number of times while moving each picket over by a certain amount each time.

Another example where procedural modeling is very effective is in describing natural phenomena. Consider a complicated object such as a tree. If you were to describe how a tree looks by describing each and every twig, leaf and branch, you would have a very large description which doesn't really get at the essence of the tree. Instead, you could describe a tree by saying that it has a trunk which divides into branches and each branch divides into smaller and smaller branches until the little twigs end in leaves. This description simply relates *how* the tree is created, rather than describing the entire tree itself, and that's where the power of procedural modeling lies.

Parametric Procedural Models

To create procedural models, you have to learn how to use the OMAR programming language, which is described in the *OMAR Programming Language Reference Manual and Programming Guide*. The rest of this section gives a brief overview of the kinds of things you can do with procedural modeling, along with some code examples.

One technique for creating flexible models that can easily be varied is to provide the models with parameters. The idea behind parameters is to use the same object description with different values to create different objects. The picket fence from the example above could have the number of pickets as a parameter. Then the same object description could be used to create fences with 100 pickets or 1000 pickets.

The following two examples illustrate how a shape description, crawler, can be given extra flexibility by using a variable parameter, treads.

Figure 3-58: Crawler with Treads = 10

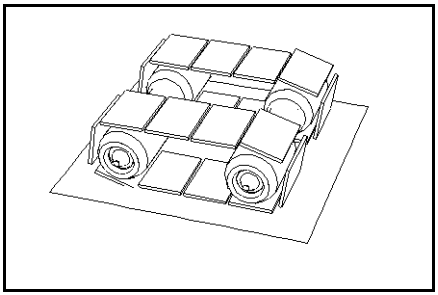
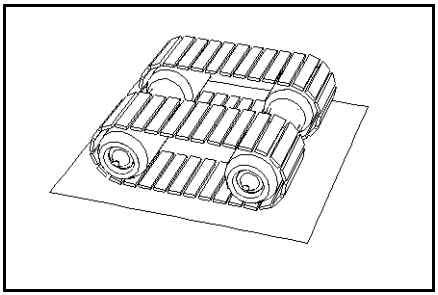


Figure 3-59: Crawler with Treads = 30



Fractals in Nature

Many shapes in nature have the characteristic that if you examine them on a small scale, you see the same structure that you see on a large scale. If you look at a mountain, you see that it has many rocky outcroppings which in turn each look like little mountains. Each individual rocky outcropping is made up of large boulders which in turn look like even littler mountains, and so forth. The same thing goes for trees. A tree has a few large branches, which are each similar to smaller trees. The large branches have smaller ones, which have even smaller branches until you get to the tiny branches, which have leaves attached. This property is called *self-similarity*.

Figure 3-60: Tree with Branching = 2

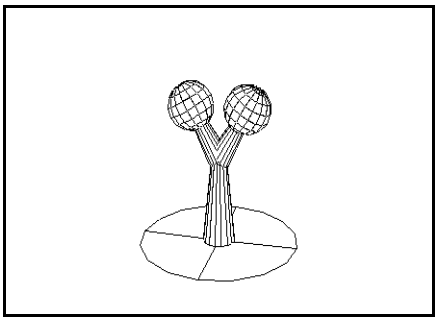
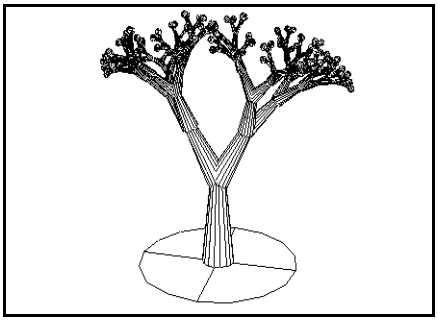


Figure 3-61: Tree with Branching = 8



Implementing Fractals

To implement fractals on the computer, you express the self-similar property of objects by having the object contain smaller copies of itself. The definition of the objects is therefore, recursive. All recursive objects must have a point at which the recursion stops or else the computer will try to build an infinitely recursive object and eventually fail. For the tree example, the recursion stops when the tiny branches turn into leaves. A stopping point can be specified by having the object take an integer parameter that indicates the recursion level. When the object creates one of its recursive parts, it increments the recursion level one step closer to the stopping level so that eventually, after so many steps, the recursion stops.

In the tree example below, the recursion level is indicated by the integer named branching.

Listing 3-7: A Fractal (Recursive) Tree

```
do tree_picture;

include "3d.ores";

shape tree with
      integer branching = 10;              // the recursion level
is
      shape leaf is
            sphere with radius = .4;  color = green;  end;
      end;        // leaf

      // trunk
      cone with
            end1 = <0 0 0>; end2 = <0 0 1>;
            radius1 = .2; radius2 = .1; color = brown;
      end;
      if branching > 1 then
            // branches
            tree with
                  branching = static branching - 1;
                  rotate by 60 around <0 0 1>;
                  rotate by 30 around <0 1 0>;
                  magnify by .7; move to <0 0 1>;
            end;
            tree with
                  branching = static branching - 1;
                  rotate by -60 around <0 0 1>;
                  rotate by -30 around <0 1 0>;
                  magnify by .7; move to <0 0 1>;
            end;
      else
            leaf with move to <0 0 1>; end;                        // leaf
      end;
end;        // tree

picture tree_picture with
      eye = <2 -8 4>;
      lookat = <0 0 1.25>;
      field_of_view = 40;
is
      distant_light from <1 -3 2>;
      tree;
      disk with color = dark green; end;
end;        // tree_picture
```

In the following 'sphereflake' example, the level of recursion is indicated by the integer variable named level.

Figure 3-62: Sphereflake with Level = 2

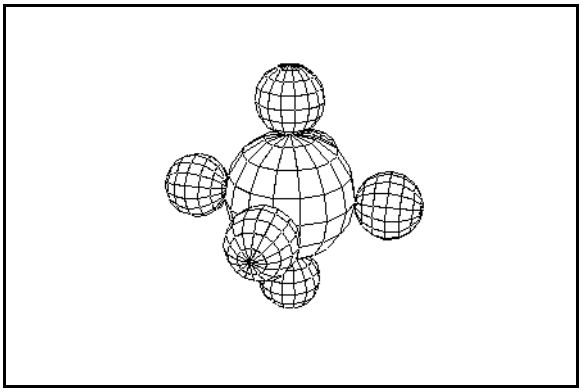
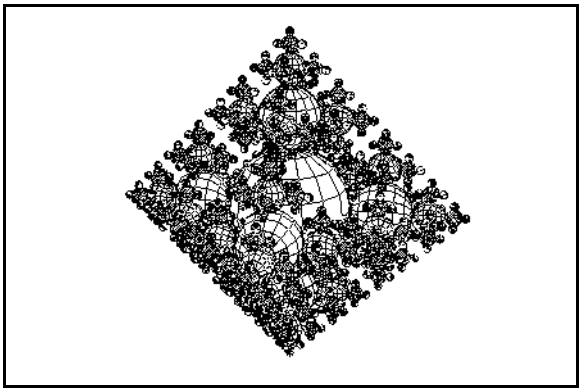
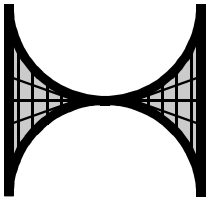


Figure 3-63: Sphereflake with Level = 5



Listing 3-8: A Fractal (Recursive) Sphereflake

```
do sphereflake_picture;

include "3d.ores";

shape sphereflake with
        integer level = 8;              //the level of recursion
        scalar factor = 0.5;            //determines the relative size of each level of spheres
is
        shape miniflake is
                sphereflake with
                        level = static level - 1;
                        magnify by factor;
                        move to <0 0 (1 + factor)>;
                end;
        end;        // miniflake

        if level mod 3 = 0 then
                color = orange;
        elseif level mod 2 = 1 then
                color = dark raspberry;
        else
                color = light teal;
        end;

        // Center sphere
        sphere with material is metal colored color;

        // Surrounding sphereflakes
        if level > 1 then
                miniflake;
                miniflake with rotate by 180 around <1 0 0>; end;
                miniflake with rotate by 90 around <1 0 0>; end;
                miniflake with rotate by -90 around <1 0 0>; end;
                miniflake with rotate by 90 around <0 1 0>; end;
                miniflake with rotate by -90 around <0 1 0>; end;
        end;
end;        // sphereflake

picture sphereflake_picture with
        eye = <2 -8 4>;
        field_of_view = 70;
        reflections is on;
        shadows is on;
        facets = 0;                             //sets the render_mode to ray tracing
is
        distant_light from <.3 -.5 1>;
        sphereflake with
                level = 7;
        end;
end;        // sphereflake_picture
```



CHAPTER 4
Viewing

As has been noted, the computer graphics process is a lot like photography. In both systems, a good picture depends as much upon the camera placement as it does on the subject matter and lighting of the scene. Using the Hypercosm system, once you've modeled a scene, you can easily generate several different renderings of the same scene with little effort, just by changing the position of the 'virtual camera.' An architectural rendering, for example, has a very different feel to it depending upon whether you choose to place the camera in front of a building, inside of it or above it, as in an aerial shot.

Hypercosm also has an assortment of different camera lenses from which to choose. Wide angle lenses may be used to emphasize the wide open expanse of a fractal mountain landscape or the towering height of an imaginary skyscraper. Telephoto lenses may be used to minimize or even eliminate perspective to make an engineering design more understandable.

This chapter describes Hypercosm's numerous viewing options.

Camera Placement

Any camera placement can be described to the computer by two parameters. These are:

- eye—The location of the virtual camera.
- lookat—The location towards which the camera points.

If the camera positioning variables are not explicitly set in your OMAR files, the defaults, found in **native_render.ores**, will be used.

Figure 4-1: The Camera Positioning Variables & Defaults

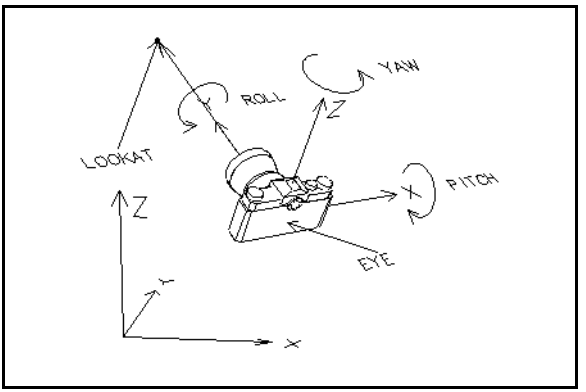
```
vector eye = <10 -30 20>;
vector lookat = <0 0 0>;
```

If you want to set these parameters (or any of those described later in this chapter) yourself, you can do so in the with section of a picture or anim.

Example: Setting the Camera Parameters

```
picture scene with
      eye = <-2 -4 3>;
      lookat = <-1 0 1>;
is
      {...}
end;
```

Figure 4-2: Camera Geometry



Camera Orientation

Once you specify where the camera is and in what direction it is pointed, you can tilt the camera around in the frame of reference of the camera using the parameters roll, yaw and pitch.

- roll—The angle of the camera's tilt around the line of sight. A positive roll tilts the camera in a clockwise direction, which spins the screen image counterclockwise.
- yaw—Use this angle to turn the camera to the left or right. A positive yaw turns the camera to the right, which causes the image to swing to the left.

- pitch—Use this angle to tilt the camera up and down. A positive pitch tilts the camera upwards (as in an airplane).

Figure 4-3: The Camera Orientation Variables & Defaults

```
scalar roll = 0;
scalar yaw = 0;
scalar pitch = 0;
```

Field of View

Another important viewing parameter is the field of view. Imagine the volume of space that is taken in by the camera lens to be a pyramid, narrowing to a point at the camera and extending outwards infinitely in the direction that you are looking. This is known as the *viewing frustum*. The field of view is specified as the angle between the opposite sides of the viewing frustum.

A camera's field of view is usually about 60 degrees. If you use a wide angle lens, the field of view is somewhere around 80 to 100 degrees. A telephoto lens usually has a field of view around 10 to 30 degrees.

Figure 4-4: The Field of View Variable & Default

```
scalar field_of_view = 60;
```

Figure 4-5: The Viewing Frustrum

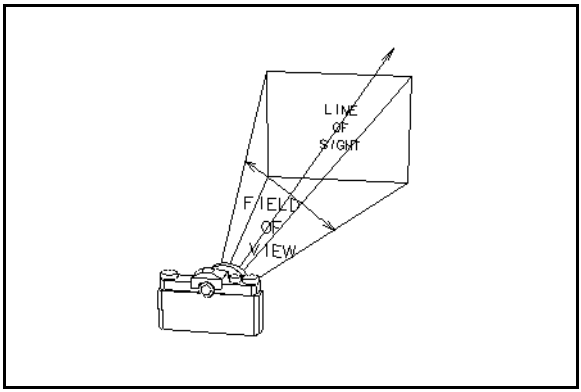


Table 4-1: Fields of View of Some Popular 35mm Lenses

Focal Length (mm)	Lens Class	Field Of View
8	Ultra Wide Angle	170
17	Very Wide Angle	140
28	Wide Angle	110
35	Moderate Wide Angle	90
50	Normal	60
80	Moderate Telephoto	30
200	Telephoto	10

Projection

In computer graphics, as in photography, there is no perfect way to project the three-dimensional world onto a flat surface. Cartographers face a similar problem when representing the curved surface of the earth on a flat piece of paper. While there is no perfect solution to the problem, there are several different approaches, each with its own merits. Hypercosm supports four different useful projection types: orthographic, perspective, fisheye, and panoramic.

Figure 4-6: The Projection Type, Variable, & Default

```
enum projection is orthographic, perspective,
                   fisheye, panoramic;
projection type projection is perspective;
```

The Orthographic Projection

The orthographic projection is distinguished from others by having a total lack of perspective. That is, objects far away do not appear smaller than objects close by, so everything in the image is on the same scale. This projection is unusual because it doesn't have a true counterpart in the real world. If you look through a telescope, or other optical device with a tiny field of view, then the image that you see is close to an orthographic image because there is very little perspective effect.

The orthographic projection is useful in cases where you want to produce an image without any perspective. This is common in engineering drawings because you want to give a sense of the 3D geometry of the object without distorting the relative scaling of various parts of the object.

Instead of a pyramidal viewing region, the viewing region takes the form of an infinitely long block with a rectangular cross section. Because all the sides of

the viewing region are perpendicular, you can no longer specify the field of view as an angle, so for this projection, unlike the others, the field of view variable sets the distance from one corner of the viewing region to the opposite corner.

Figure 4-7: The Orthographic Projection

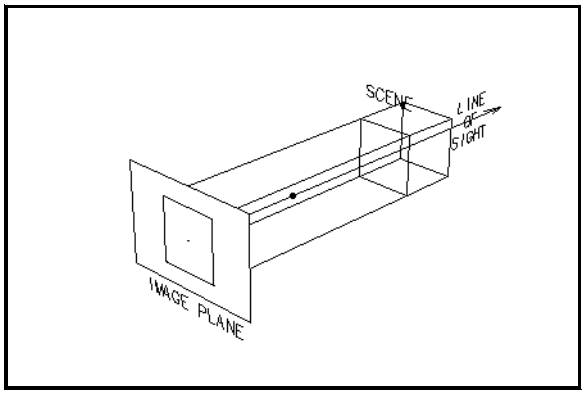
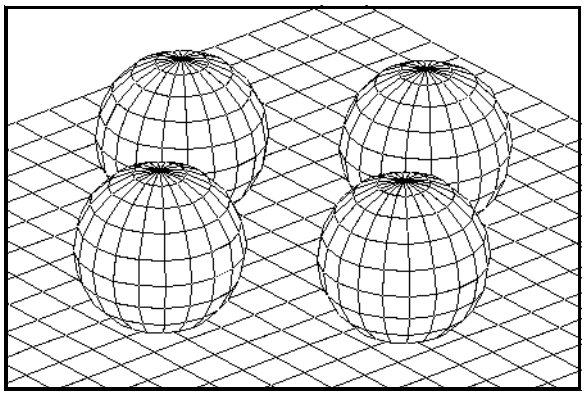


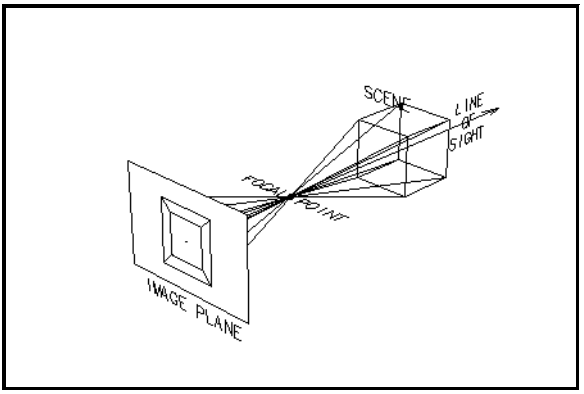
Figure 4-8: An Orthographic Image



The Perspective Projection

The perspective projection is the most common way to map our surroundings onto a flat 2D surface and is therefore the default for the Hypercosm system. If you were to build a pinhole camera that projected its image onto a flat piece of film, the resulting image would use the perspective projection. Note that the field of view used in this projection can never be more than 180 degrees because there is no way of projecting light from behind the camera onto the film plane.

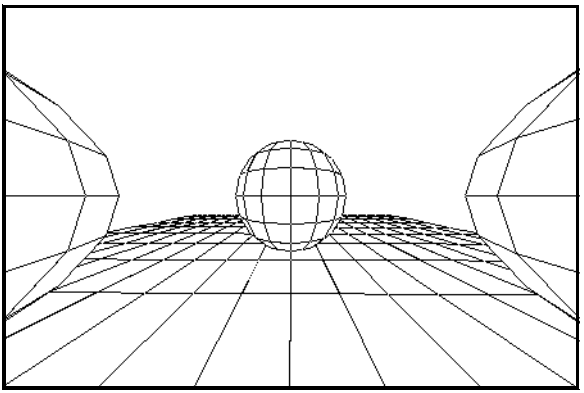
Figure 4-9: The Perspective Projection



One characteristic of this projection is that straight lines in the real world project to straight lines in the picture. This is useful to know if you are creating architectural renderings and you want the straight lines of the building to be straight in the rendering. A drawback of this projection is that for wide fields of view, it must severely distort the image in order to preserve the straight lines. Objects toward the center tend to shrink and objects at the perimeter of the image are stretched out. This is often especially disconcerting during animations because as the camera pans across the scene, objects will enter the picture, shrink noticeably as they cross the center of the image, and then stretch out again as they approach the edge of the picture again.

You can understand this if you look at the model of the camera. Towards the edge of the picture, the distance from the lens to the film increases, so the magnification of the image increases in those areas proportionately. The problem increases dramatically with wider fields of view. For very wide fields of view, the fisheye projection may be more suitable.

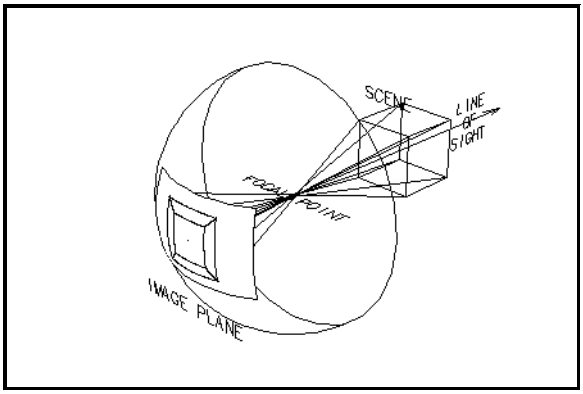
Figure 4-10: A Perspective Image



The Fisheye Projection

The fisheye projection is an attempt to model the camera more closely after the human eye. In the pinhole camera analogy, instead of projecting the image onto a flat piece of film, the image is projected onto a piece of film that is curved onto the back of a sphere with the lens at the center. You can imagine that this sphere represents the eyeball and the film represents the retinas.

Figure 4-11: The Fisheye Projection

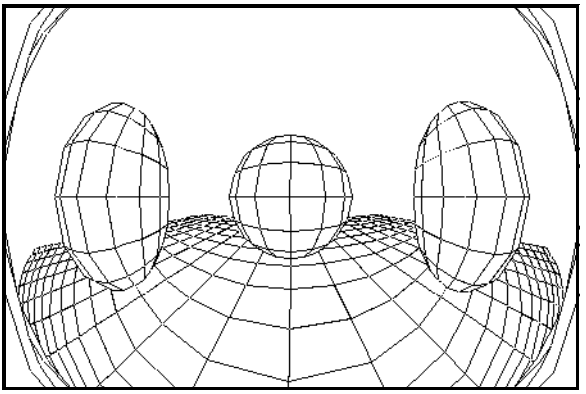


An important characteristic of this projection is that it is capable of presenting very wide fields of view, theoretically up to 360 degrees. For more moderate fields of view, the stretching problems of the perspective projection are alleviated since the film is the same distance from the film plane in all places.

One unavoidable problem with this projection is that straight lines in the scene project to curved lines in the image. This may make fisheye projection undesirable for many renderings.

Although the 'fisheye effect' may appear unnatural in very wide angle images, humans actually see this way all the time but just don't notice the effect because our attention is fixed on a small portion of our vision. If you were to lie down in the middle of a field, looking straight up at the sky, and focused your attention on your peripheral vision, you would notice that you can see the horizon appearing as a circular strip all the way around the perimeter of your 180-degree field of view.

Figure 4-12: A Fisheye Image

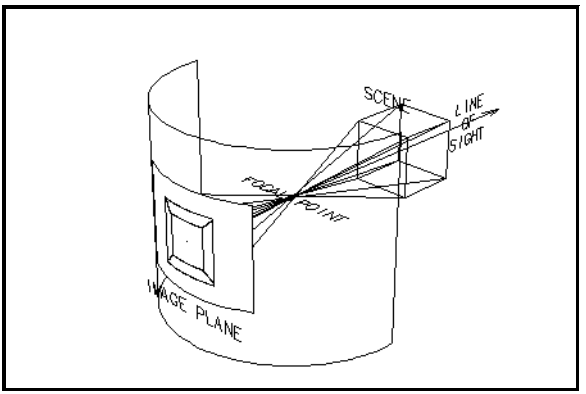


The Panoramic Projection

When people take wide angle pictures, they are usually interested in capturing the detail that is visible in a strip near the horizon, and much less interested in the sky above or the ground below. With this in mind, a special type of camera was made that takes pictures by using a cylindrical strip of film and revolving the lens in a circle around it. The results of this kind of photography can be seen in Circlevision 360 theaters at places like Walt Disney World's Epcot Center, where the audience stands in a circular theater with images projected along the entire inside surface of the cylindrical wall.

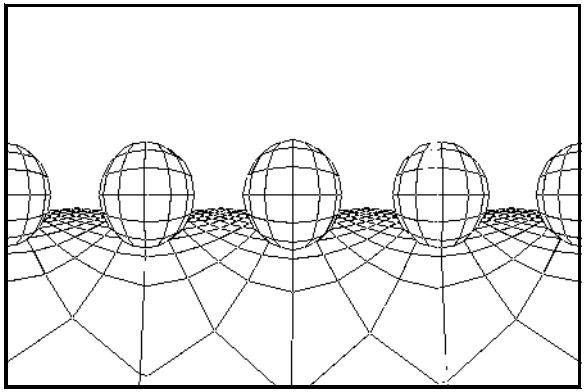
Since this projection is the only one that doesn't have radial symmetry, the field of view is measured across the horizontal dimension of the image instead of across the diagonal. The field of view can encompass all 360 degrees in the horizontal direction. A 90 degree field of view, for example would give you a pie shaped region with the left edge and right edges of the image 90 degrees apart.

Figure 4-13: The Panoramic Projection



The panoramic projections is similar to the fisheye projection in that straight lines in the three-dimensional world do not project to straight lines in a panoramic image. This projection may still be suitable for architectural renderings, however, because if the camera is level, then all vertical straight lines in the world will project to vertical straight lines in the image.

Figure 4-14: A Panoramic Image



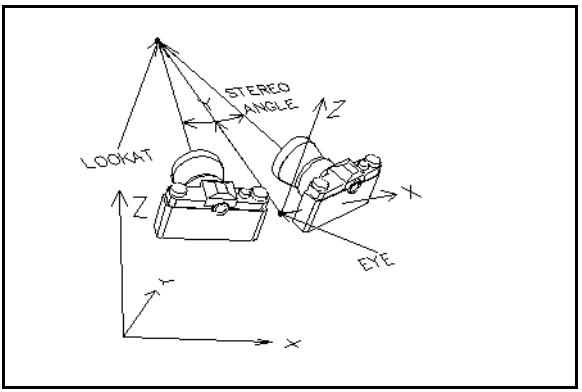
Stereoscopic Pictures

In the real world, you perceive objects as having depth. Your ability to perceive the depth of objects comes from the fact that your two eyes are separated from one another, and therefore each eye sees a slightly different view. Your brain integrates these two slightly different images into one three-dimensional image—a *stereoscopic* image.

In order to simulate stereo vision, Hypercosm uses two separate virtual cameras, one for each eye, and simultaneously presents a different image to each eye. The difficulty here is how to simultaneously present a different image to each eye when both of your eyes are looking at the same computer monitor.

Hypercosm uses a technique called *anaglyph*. This is the same technique used in the 3D movies that were popular in the 1950s. The technique takes advantage of color filtering: a red image can pass through a red filter, while a blue image is filtered out, and a blue image can pass through a blue filter, while a red image is filtered out. You can therefore display two separate images on the same screen by compositing their colors together and using two different complementary colored filters over the eyes to allow one image to pass through to each eye. The compositing operation may work with any two colors, but two complementary colors are necessary to achieve the stereo effect since each filter must be able to pass one image and filter out the other.

Figure 4-15: Stereo Viewing Geometry



Stereo Glasses

To view a stereo image, you must wear a pair of special stereo glasses with a differently colored filter over each eye. The convention that Hypercosm uses is to place a red filter over the right eye and a blue filter over the left eye. You can make your own colored glasses by placing pieces of colored acetate over a pair of clear lenses. Colored acetate is widely available at art supply stores. Several layers of colored cellophane works too, but is not as clear as the colored acetate.

How to Use the Stereo Feature

To activate the stereo rendering, set the stereo parameter to a value greater than 0. This stereo parameter refers to the angle between your two eyes in relation to the lookat point.

Figure 4-16: The Stereo Variables & Defaults

```
scalar stereo = 0;
vector lookat = <0 0 0>;
```

The stereo angle is measured in degrees. Values between about 5 and 10 work well. Too little stereo separation, and the depths of objects are difficult to perceive. Too much stereo separation, and you'll have a hard time 'fusing' the two images in your mind and it will soon give you quite a headache!

You may notice that some parts of the image appear to go into the computer screen and some parts seem to float out in front of the computer screen. This is controlled by the lookat parameter. Objects that are farther away from the eye than the lookat point recede behind the screen and objects that are closer than the lookat point float out in front of the computer screen.

Changing Stereo Colors

Some pairs of stereo glasses have the blue filter over the right eye and the red filter over the left eye, or you may want to experiment with using your own colored filters, so Hypercosm provides a way to change the colors used in the stereo renderings.

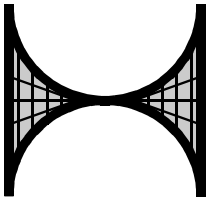
Figure 4-17: The Stereo Color Variables & Defaults

```
vector left_color = cyan;
vector right_color = red;
```

Also, you may need to tweak the colors slightly to match the phosphor colors of your computer monitor. Perhaps you just want to experiment with different colored filters. Theoretically, any two complementary colors should work.

Producing Stereo Pairs

Sometimes it is desirable to produce a pair of images that can be photographed off the screen and placed in a stereo viewer to yield stereo images in true color. To produce a rendering from the right eye's point of view, set right_color to white and set left_color to black. To produce a rendering from the left eye, set left_color to white and right_color to black.



CHAPTER 5
Rendering

The term *rendering* refers to the process of producing an image on a computer screen. Whereas modeling commands control what an image contains, rendering commands control how the image is displayed (for example, a cube may appear as a shaded block or as a group of connected lines). This chapter details Hypercosm's wide variety of sophisticated rendering capabilities.

Window Dimensions & Position

You can easily control the dimensions of the drawing window. Keep in mind that the larger the image, the longer it takes to produce, because more pixels must be rendered. For line drawings, the rendering time increases roughly linearly with the window dimensions, because if you double the window dimensions, then drawing a line requires drawing approximately twice as many pixels.

When you use the shaded rendering modes (described later), however, the time it takes to produce an image increases linearly with the *area* of the picture. This is because the time is roughly proportional to the number of pixels in the image, which is in turn proportional to the image's area. If you double an image's height and double its width, its area is four times larger, and the image takes four times longer to render.

These parameters (and others discussed in this chapter) may be set in the with section of a picture or anim. If these parameters are not explicitly set in your OMAR files, their default values are used.

Figure 5-1: Window Dimensions Variables & Defaults

```
integer width = 512;
integer height = 384;
```

Hypercosm Studio users should note that the height and width values that you set in your project settings take precedence over whatever values you may set in your code. If you'd rather set height and width in your code, you must prevent Studio from passing height and width as program arguments. (See the *Hypercosm Studio User Guide* for more details on this feature.)

The drawing window can also be moved about on the screen. You can center it, or move it to the upper left hand corner of the screen, if you prefer. You can even move it off the screen entirely. The default position (specified in **native_windows.ores**) is the screen center.

Figure 5-2: Window Position Variables & Defaults

```
integer h_center = screen_width div 2;
integer v_center = screen_height div 2;
```

Screen Dimensions

Since computer makers have all decided to make machines with different screen resolutions, Hypercosm must ask the computer about its screen. This lets you create an image that takes up the whole screen, or just a fraction of the screen, no matter what computer displays that image.

Some computers have a variety of different video modes to make your life even more complicated. On these machines, the screen width and height reported depend upon the video mode in use.

Figure 5-3: Functions for Finding the Screen Dimensions

```
integer question screen_width;
integer question screen_height;
```

Although the program can determine the shape and size of the screen in terms of the number of pixels, it has no way of knowing the actual, physical shape of the screen. You may have to supply this yourself to ensure images are not squashed or stretched. This information is given as the *aspect ratio*.

The aspect ratio of the screen is a measure of how elongated the screen is in the vertical direction. A skyscraper has a high aspect ratio and a sports car has a low aspect ratio. The aspect ratio of a square is exactly 1. Most computer monitors are built so that the ratio of (height / width) is about 3/4, or 0.75. Because of slight differences among manufacturers, however, this is not always exactly the case, so an image that looks fine on one computer monitor may look slightly squashed or stretched on another. To compensate for this, Hypercosm lets you set the aspect ratio of the screen so that the program automatically squashes or stretches the image to compensate for the monitor.

To do this, simply measure the height and width of your computer screen and use the (height / width) as the aspect ratio.

Figure 5-4: Screen Aspect Ratio Variable

```
scalar aspect_ratio;
```

Background Color

Changing the background color is a simple matter of setting the variable, background. The background color can have a profound effect on the appearance of shapes in the scene because highly reflective materials such as metals depend strongly upon their surroundings.

Figure 5-5: The Background Color Variable & Default

```
vector background = black;
```


Rendering Mode

A single image may take anywhere from a fraction of a second to many hours or days for a computer to generate. The speed at which an image is generated depends on many factors related to how realistic you want the image to be and the complexity of the image. For this reason, several different rendering modes are available that vary widely in realism and speed.

In addition, you may want to use different rendering techniques for aesthetic reasons. The most desirable picture for a particular application may not always be the most realistic one. In some cases, different rendering modes may be mixed by resetting the render_mode variable at various locations in a program.

Figure 5-6: The Render Mode Type, Variable & Default

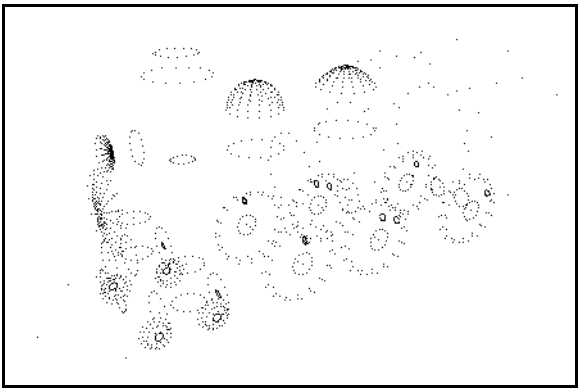
```
enum render_mode is pointplot, wireframe,
                    hidden_line, shaded, shaded_line;
render_mode type render_mode is shaded;
```

The Pointplot Rendering Mode

This rendering mode simply draws the scene as a collection of points. Polygonal objects such as cubes or polygons have a point at each vertex while curved objects such as spheres have points evenly spaced upon a mesh across their

surface. The number of points in the mesh can be increased or decreased using the facets variable, which is described in the *Tessellation* section.

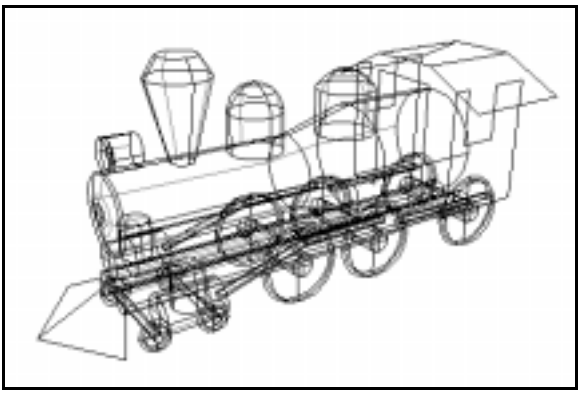
Figure 5-7: The Pointplot Rendering Mode



The Wireframe Rendering Mode

This rendering mode produces a common style of computer graphics images in which objects are rendered by drawing only their edges. For curved objects, Hypercosm draws a curved grid of straight lines across the surface like the longitude and latitude lines that are found on a globe. Since you are only drawing the edges of objects, you can see through them. This rendering mode makes objects tend to look like they are constructed from a skeletal framework of thin wires, hence the name, *wireframe*. The edges that are drawn are determined by the edge mode (described in the next section) and may include all edges, the outline edges, or just the silhouette edges.

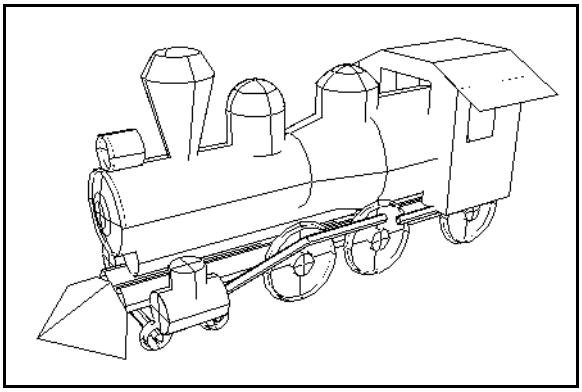
Figure 5-8: The Wireframe Rendering Mode



The Hidden Line Rendering Mode

The hidden line rendering mode increases the realism of the wireframe rendering mode by using hidden surface removal. This means that closer objects block objects that are farther away. As in the wireframe rendering mode, the edges that are drawn are determined by the edge mode.

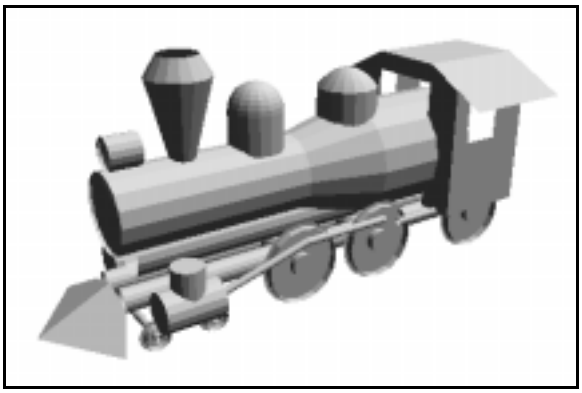
Figure 5-9: The Hidden Line Rendering Mode



The Shaded Rendering Mode

This rendering mode draws the surfaces of objects as you see them in the real world. Closer objects block farther objects from view and surfaces are shaded with respect to the lighting in the scene. In addition, shadows, reflections, transparency, fog, and other effects may be added to provide additional realism. The shaded mode is Hypercosm's default rendering mode.

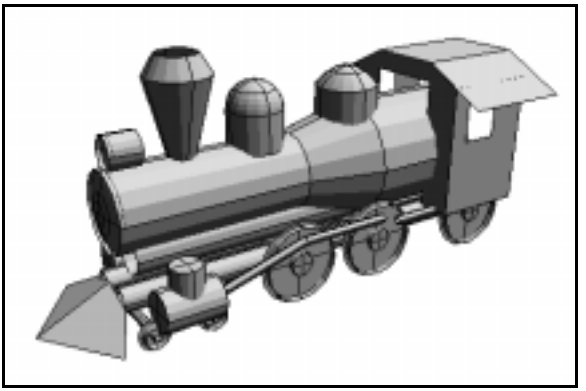
Figure 5-10: The Shaded Rendering Mode



The Shaded Line Rendering Mode

This rendering mode is a hybrid between wireframe and shaded modes. The edges of all the objects are drawn, as in wireframe mode, but the surfaces of the objects are also drawn. Objects may be shaded using all of the normal options of the shaded mode, as described later in the *Shading* section.

Figure 5-11: The Shaded Line Rendering Mode



Edges

In the wireframe and hidden_line rendering modes, where only the edges of shapes are drawn, you can choose which edges are rendered. Most 3D graphics systems render every edge of an object, which has the tendency to create cluttered and confusing renderings. Hypercosm can 'clean up' the renderings considerably by drawing only selected edges.

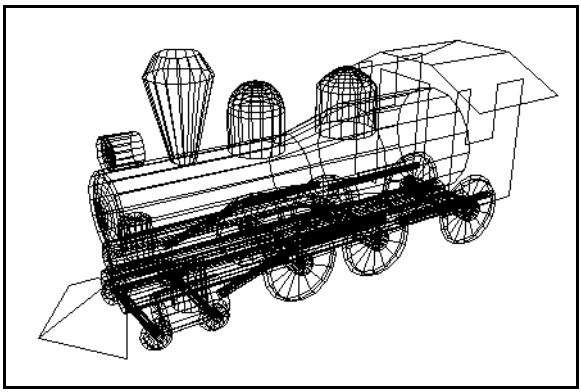
Figure 5-12: The Edge Mode Type, Variable, & Default

```
enum edges is silhouette, outline, all;
edges type edges is outline;
```

All Edges

If all edges are drawn, objects have an adequate amount of detail; however, for some complex objects, you may find that the images appear cluttered and also that some details may be obscured by the lines of other objects.

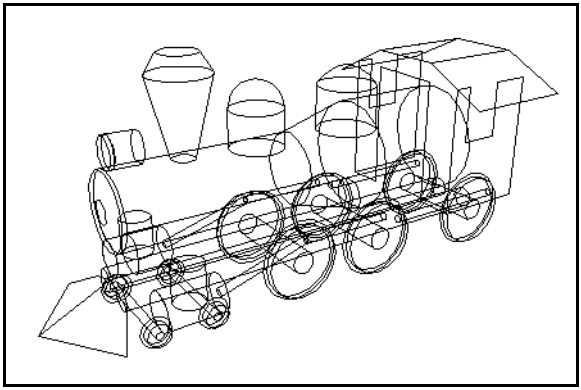
Figure 5-13: All Edges



Silhouette Edges

The silhouette mode is much less cluttered because it draws only the silhouette edges of curved objects. However, the images are often too sparse to convey the three-dimensional geometry of the scene accurately.

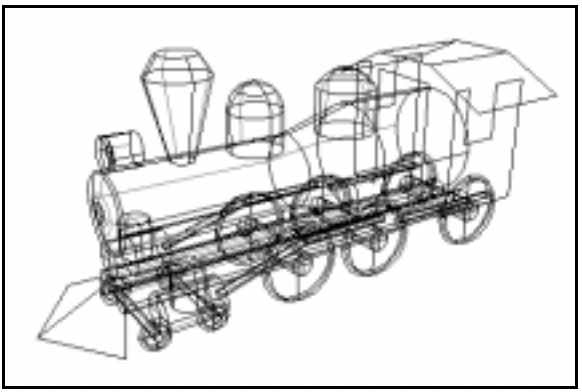
Figure 5-14: Silhouette Edges



Outline Edges

Outline mode draws the silhouette edges just like silhouette mode, but also adds lines on the surface of the curved objects at regular intervals to better convey the true shape of the object.

Figure 5-15: Outline Edges



Tessellation

Hypercosm's renderer is capable of drawing a number of basic, or primitive, surface types. Some of these primitives are simple flat primitives, such as polygons and triangles. Others are more complex curved surfaces such as spheres and torii. Because the mathematics involved with curved surfaces is more complex than the mathematics for flat surfaces, the curved surfaces can be more quickly rendered by breaking them down into a multitude of flat facets. This process is also called *tessellation*.

Hypercosm uses a few tricks to hide the tessellation to avoid your having to be concerned with the mechanics of the rendering process. For instance, the outline rendering mode draws only the silhouette edges and the important edges so we don't have to see the facets that are used internally by the renderer. Also, the number of facets has a default value so you don't necessarily have to be aware of it.

However, if you need faster rendering and can sacrifice some quality or if you wish to produce very high quality renderings and don't care as much about the time that it takes, then you must be able to control the number of facets produced by the tessellation process. If you find that drawing is too slow, you can decrease the number of facets, and although the curved surfaces will not be as smooth, the drawing will be much faster. If you are interested in drawing a higher quality shaded image, you can increase the number of facets to make them smaller. Changing the number of facets has no effect on flat surfaces.

Figure 5-16: The Tessellation Variable & Default

```
integer facets = 6;
```

Figure 5-17: Facets = 2

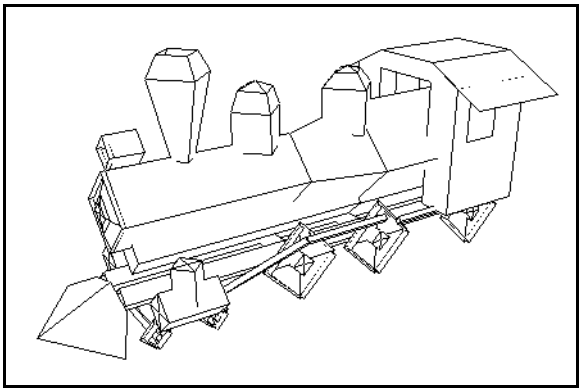


Figure 5-18: Facets = 8

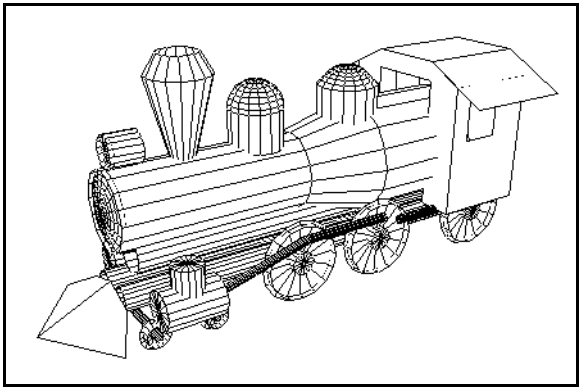
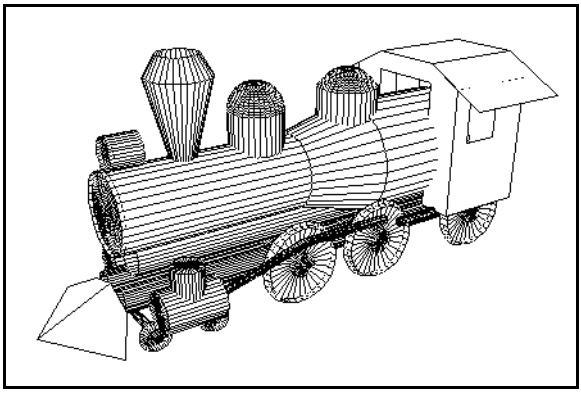


Figure 5-19: Facets = 16



The facets are more noticeable in some rendering modes than in others. For example, in the wireframe mode, the facets are more noticeable when you draw all of the edges than if you draw just the silhouette or outline edges. In the shaded modes, the facets are more noticeable when you use face (flat) shading than when you use vertex (Gouraud) or pixel (Phong) shading. (The shading modes are described later in this chapter.)

Ray Tracing

Ray tracing is a method of rendering without tessellation. If you want very high quality images, you can choose to render the objects from their exact mathematical surface descriptions, instead of tessellating them. To do so with the Hypercosm system, you use an algorithm that is known as ray tracing because it simulates the behavior of light rays.

To ray trace an image, set the facets variable to 0. The only disadvantage to rendering this way is that it takes longer to create an image; use this technique sparingly in animations. A common practice is to do most test renderings with the tessellated shading and to use the high quality rendering for the finished image when you have all of your objects and lighting set up the way you want it.

Scanning

When Hypercosm uses the ray tracing mode, you will notice that the picture is not drawn in the same way as it is when you use regular shading. When Hypercosm uses the regular shading algorithm, each object is created by drawing each facet of the object sequentially. In ray tracing, however, there are no facets, so Hypercosm draws the image by *scanning* imaginary light rays across the object to see where they hit the surface, and what color the surface is at that point. Hypercosm can scan the rays across the entire picture just as easily as scanning the rays across each object one at a time. This is why you see the entire picture being drawn all at once instead of each object drawn one at a time.

When it uses ray tracing, Hypercosm fires a ray out into the scene from every pixel on the screen. Since it doesn't matter what order the rays are fired in, Hypercosm provides three different scanning modes: linear, ordered, and random. All three modes produce the same image when finished, but each draws the image in a different order.

Figure 5-20: The Scanning Type, Variable, & Default

```
enum scanning is linear, ordered, random;
scanning type scanning is ordered;
```

Linear Scanning

In linear scanning, Hypercosm starts at the top of the screen and works its way slowly down to the bottom, completely drawing everything in the region of the screen that it's working on, before moving on to the next.

The problem with drawing the image this way is that you don't get a good idea of what the image will look like until it's almost completely finished. Also, if you have a mistake in the picture, like the wrong color or two interpenetrating objects, for example, you won't see the mistake until the renderer gets to that part of the image, which, if you are very unlucky, may not be until the end.

Watching the image being created in this way can be a little like watching the grass grow. If you don't want to watch the image being created, however, then this mode is preferable because it is slightly faster than the other two scanning modes.

Ordered Scanning

This rendering mode is much more fun to watch because it scans the light rays across the image in a grid which becomes progressively finer as the picture gets closer to being finished. This enables you to get a general idea of what the image will look like very soon, with the finer details being filled in later.

Random Scanning

This mode is much like the ordered mode, except that instead of scanning the image in a progressively finer grid, it scans the image in a generally random pattern, so it looks like the rays are being scattered randomly across the image. This has the effect of making the image seem to 'dissolve' into view. This mode is slightly slower than ordered, but it is fun to watch.

A Note About Ray Tracing: *Voxels*

When using ray tracing, if the double buffer is not enabled, so you can see the drawing taking place, you will notice that the program first draws an outline view of the object and then proceeds to draw lots of boxes around the object with the boxes getting progressively smaller. These boxes are called *voxels* and are used internally by the program to speed up the ray tracing process.

Shading

In real life, when you look at the surface of an object, the color that you see normally changes continuously across the surface. There may be sharp discontinuities and there may be very gradual changes, but normally, the color of the object never stays exactly the same as your eyes move across its surface.

The computer has to perform a number of mathematical computations to find what the color is at any given point on the object. Because there are an infinite number of points on any surface, you can't ever know exactly what the color of the object is at every point on its surface.

Instead, Hypercosm has to ask what the color is at various places, and how this is done is controlled by the variable called shading. The shading mode determines how often Hypercosm performs the shading calculations. When it evaluates the lighting model more often, you get more accurate shading at the cost of a slower rendering time. The three different shading modes are face, vertex, and pixel.

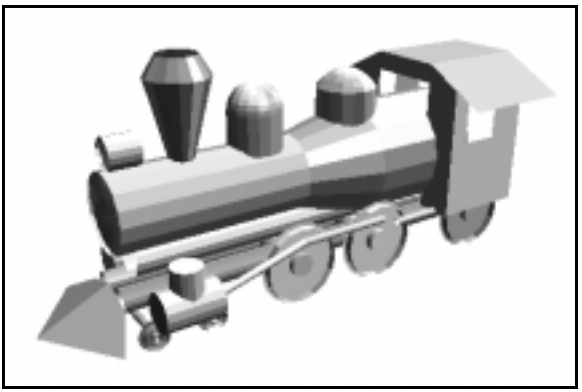
Figure 5-21: The Shading Type, Variable, & Default

```
enum shading is face, vertex, pixel;
shading type shading is vertex;
```

Face Shading

When you use *face shading*, Hypercosm computes the color at each facet by sampling it at one vertex and then using that color for the entire facet. In computer graphics this is also known as *flat shading* because it makes each facet looks flat.

Figure 5-22: Face Shading

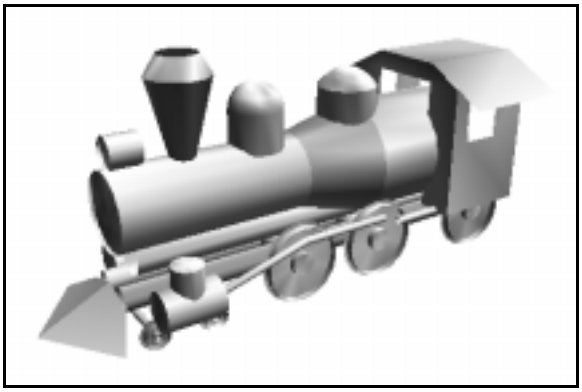


Vertex Shading

When using face shading, you may find that even when you make the facets very tiny, they are still visible. This is because the eye is very sensitive to the sharp changes in color that occur across the polygon boundaries. To eliminate the sharp discontinuities in the shading, you can use *vertex shading*, a technique that computes the lighting at each vertex and blends, or interpolates, the color across the facet. This gives the facet the appearance of being curved. In

computer graphics this technique is also known as *Gouraud shading*, after the French mathematician, Henri Gouraud.

Figure 5-23: Vertex Shading

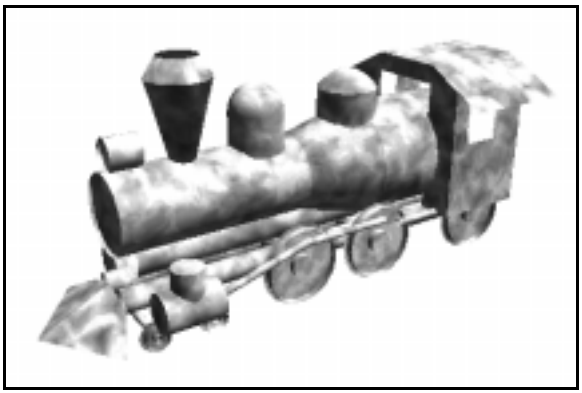


Pixel Shading

While vertex shading tends to work extremely well when the color of an object changes smoothly across its surface, it is less effective on objects that are textured or very reflective. For these kinds of surfaces, sampling the color at every vertex isn't good enough because the color of the surface may change radically in between the vertices. In these cases it's best to sample the shading at every pixel. The *pixel shading* technique, also known in computer graphics as *Phong shading*, recomputes the color for each tiny dot on the surface of the object.

Since an object typically has only a few hundred vertices, and a computer screen typically has around a million pixels, the technique can be considerably slower than vertex shading, and is a poor choice for interactive applets.

Figure 5-24: Pixel Shading



Note that although the pixel shading technique often looks almost as good as rendering the exact surface description, it still uses the underlying tessellation of the objects, which is visible in the segmented silhouette of curved objects. If Hypercosm were rendering from the exact surface description (ray tracing), these silhouettes would be perfectly smooth.

Feature Abstraction

Often, when you are rendering scenes with a lot of dynamic range, you find that a large amount of time is spent drawing tiny little details that you can barely see, because they occupy only a few pixels on the screen. If the object only occupies a few pixels on the screen, then it is a waste of time to go through all the mathematical operations necessary to draw the complete object. When the object is very small, it may suffice to substitute a very simple object, like a box, for the more complicated object. This is called *feature abstraction*.

Figure 5-25: The Feature Abstraction Variable & Default

```
scalar min_feature_size = 0;
```

The variable min_feature_size, specified as a fraction of the field of view, is the size on the screen below which the object will appear as its bounding box. For example, if you want objects which occupy less than 1/20 of the field of view to be drawn as simple boxes, then set min_feature_size = .05; If you set min_feature_size to a value that is fairly large, like .1, then you can easily see the objects turn into boxes when they drop below the minimum size, which can be distracting. The feature abstraction capability is disabled entirely when min_feature_size is set to 0.

Coarse Ray Tracing

You can use feature abstraction in ray tracing mode to cause the ray tracer to ray trace the image coarsely, drawing the image as an array of rectangular blocks of a certain size, instead of tracing a ray for every pixel on the screen. This is useful for producing ray traced animations that are capable of running in real-time. No computer is fast enough to produce images at animation rates where a ray is cast at every pixel.

If you set min_feature_size to 1/20, then the rectangles that compose the image are roughly 1/20 of the size of the image as measured across the diagonal of the image.

Antialiasing

Because the computer screen is composed of a grid of discrete dots, or pixels, the rendering process causes certain artifacts which are collectively known as *aliasing*. You can see an example of aliasing when you draw a straight line or an object with many straight edges. Unless these lines are completely vertical or horizontal, the computer shows a series of stair steps, known as the *jaggies*. Methods for eliminating the jaggies are known as *antialiasing*.

The jaggies can never be completely eliminated because there is no perfect way to represent a diagonal line on a grid with only vertical and horizontal elements. The stair-step effect can be lessened, however, by blending the lines so that pixels that are closer to the line are affected more by the color of the line. To enable antialiased line drawing, set antialiasing to true.

Figure 5-26: The Antialiasing Variable & Default

```
boolean antialiasing is false;
```

Figure 5-27: Close-up of Lines Showing 'Jaggies'

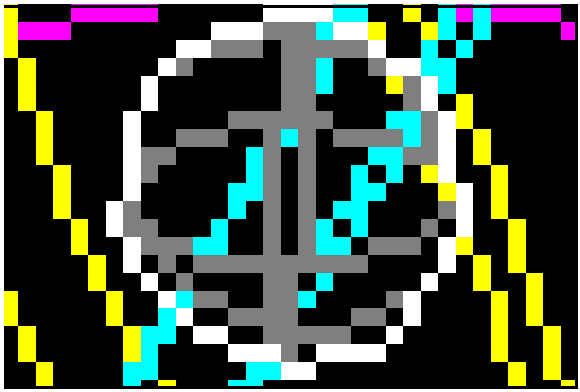
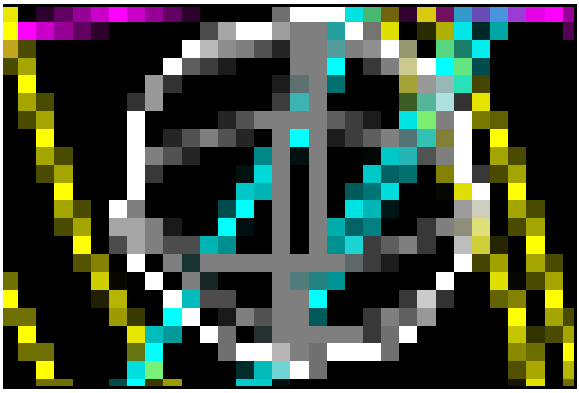


Figure 5-28: Close-up of Antialiased Lines



Supersampling

When you are creating line drawings of objects, antialiasing is fairly easy because the lines are a uniform shape and color. When you are computing the antialiasing for material images, however, the antialiasing problem becomes more complex.

The color of a shape's surface may change anywhere on the surface because of surface textures and the effects of the surrounding environment. It is not possible to compute what the exact contribution of surface color will be to each pixel. Instead, Hypercosm must sample the pixel at a number of places to determine what may be a good approximate average surface color at the pixel. It can never know exactly what the precise color for that pixel is because it would have to sample it an infinite number of times. In practice, however, sampling a fixed number of times provides a good enough approximation to the exact pixel color to yield good results. You can change the number of samples per pixel by setting the variable, supersampling. The supersampling algorithm is only used for shaded images.

Figure 5-29: The Supersampling Variable & Default

```
integer supersampling = 16;
```

Keep in mind that the amount of time that it takes to create the image is proportional to the number of samples. If you take 16 samples per pixel, the image will take 16 times longer to compute than an image with no supersampling. Supersampling can easily make images take an impractical amount of time to create.

Figure 5-30: No Antialiasing

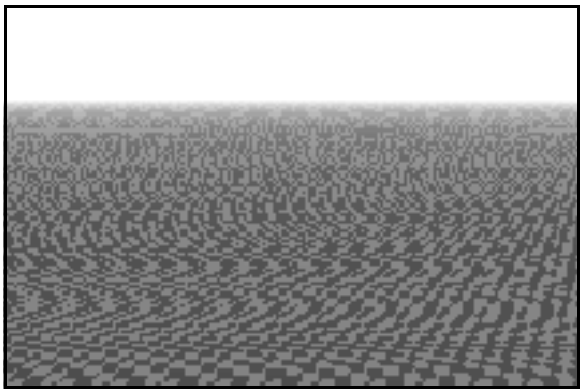


Figure 5-31: Supersampling = 4 Samples / Pixel

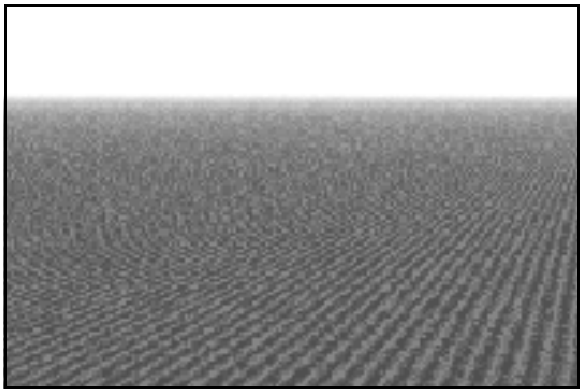
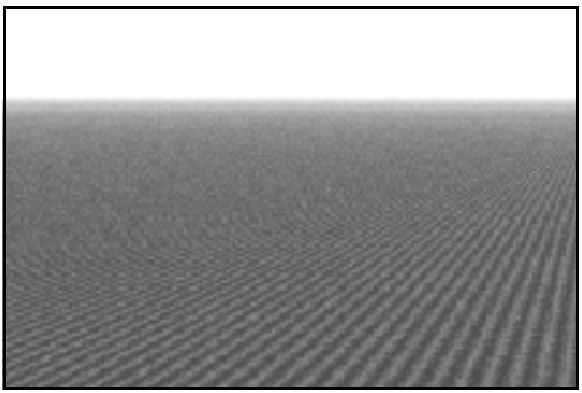


Figure 5-32: Supersampling = 16 Samples / Pixel



Shadows

Shadows are a common, everyday phenomenon, yet they are often omitted from computer graphics because they are relatively hard to compute. With the Hypercosm system you can enable them by setting the shadows variable to true.

The shadows that are cast by the distant, point, and spot light sources are sharp because these are point light sources, meaning that the light sources are infinitesimally small. In the real world, shadows are often fuzzy because light is cast by lights that aren't so tiny in size, like fluorescent light bulbs.

Figure 5-33: The Shadows Variable & Default

```
boolean shadows is false;
```

Reflections

If the surface of an object is defined by a material such as metal that calls for reflections, then you can enable the object to reflect by turning reflections on. If no material type is specified, then you see no reflections.

Figure 5-34: The Reflections Variable

```
boolean reflections is false;
```

Refractions

Refraction describes the way that light is bent as it passes through a surface. Hypercosm's refraction variable turns on transparency, which allows light to pass through surfaces. If the surface of an object is defined by a material that calls for refraction, such as a transparent or glass shader, then the refraction effect is used. If the object has no material assigned, then you don't get the transparency effect.

Figure 5-35: The Refractions Variable

```
boolean refractions is false;
```

Figure 5-36: No Shadows, Reflections, or Refraction

Figure 5-37: Using Shadows, Reflections, and Refractions

Fog

Pictures that depict outdoor scenes often benefit greatly from the addition of just a touch of fog. Even on a clear day, the atmosphere contains a certain amount of dust and water vapor that tend to obscure distant objects. The world in the computer is so perfectly clear, that unless you specifically add a touch of fog, pictures that are supposed to depict the outdoors look fake.

Without fog, fractal mountain landscapes look like they are sitting on a table top instead of stretching for miles in the distance. The fog tells you that the objects are far away; this is known as depth cueing. If you are rendering Big Ben, you could make a thick fog. You can even make colored fog.

The color of the fog is determined by the background color. The thickness of the fog is determined by the fog_factor variable. The fog_factor is the distance at which an object is halfway obscured by fog. At this distance, the color of an object is a mixture of half the color the object would be if there were no fog, and half the background (fog) color. If you set the fog factor to 10, objects 10 units away are halfway foggy, objects farther away are very foggy, and objects closer have very little fog attenuation. To create a gentle haze on this same scene, you might set fog_factor to 500. Setting fog_factor to 0 eliminates fog completely.

Figure 5-38: The Fog Variables & Defaults

```
scalar fog_factor = 0;
vector background = black;
```

Figure 5-39: Fog Physics

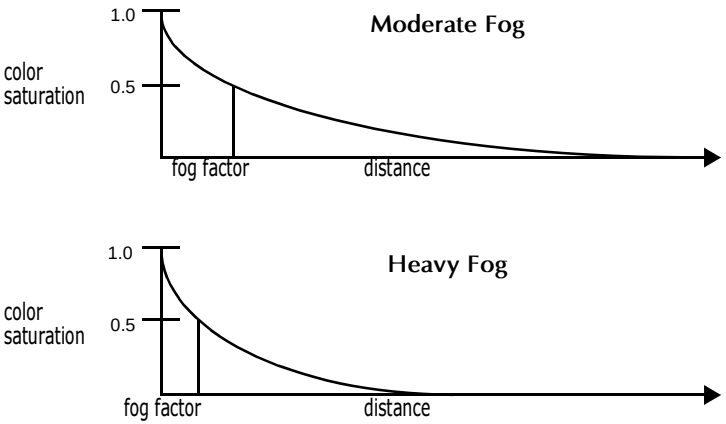


Figure 5-40: A Gentle Fog

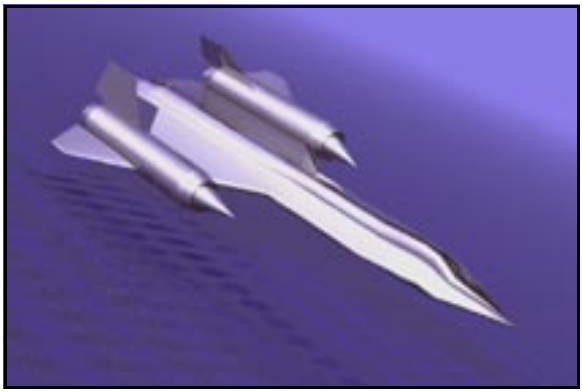
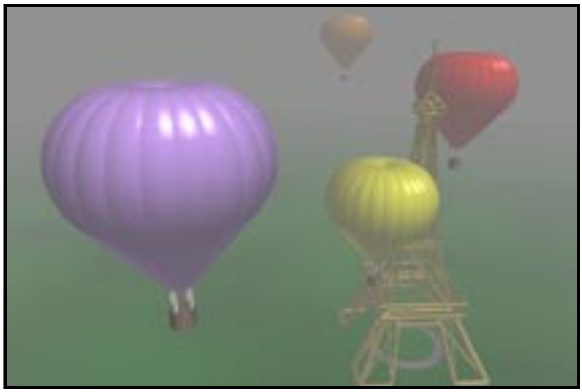
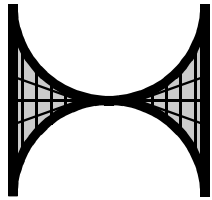


Figure 5-41: A Heavy Fog





CHAPTER 6
Animation

Animation is, literally, the process of bringing something to life. To bring imagery to life, you must make it dynamic, that is, give it the characteristic property of life, the ability to change over time.

You can create the illusion of an animated image by presenting a sequence of pictures in quick succession. If each image is only slightly different from the previous one and the time between each image is short, on the order of 1/30 of a second, then the brain fills in the tiny gaps between the images and you perceive a continuous, fluid animation. This is the underlying principle of all forms of animation, including motion pictures, television, video, and Saturday morning cartoons. It is also how Hypercosm animation works.

The Principles of Animation

Acceptable Frame Rates

Fluid animation displays at a rate of around 30 frames per second. Videotape and television are shown at 30 frames per second. Film projectors traditionally present a slightly lower frame rate of 24 frames per second. Some special projector systems, such as IMAX, present 60 frames per second which requires special projectors and film. Even at 24 frames per second, the number of individual frames required for a short animation adds up pretty quickly, so you can sometimes try to get by with a lower frame rate. If there is not much rapid change in the animation, a lower frame rate may be acceptable. Below about 10 frames per second, though, the eye can detect the changes between frames and the resulting animation may appear to be jerky instead of smooth and fluid.

Animation Speed

Many computers are not yet capable of animation speeds that match what you see on television or in the movies. The rule of thumb is that you can have image quality or you can have speed, but not both. If you find that the animation speed is too slow, you can take a number of steps to sacrifice image quality in hopes of attaining enough speed to do the job. When you're creating animations to be placed on the web, even an animation that runs smoothly on your computer may require a reduction in complexity in order to run smoothly on other computers as well.

Real-Time Animation

If you wish to create real-time animations, you are almost certainly limited to line drawing or simple shaded renderings. Many computers are still incapable of producing shaded images at an acceptable rate and no computers are capable of producing high quality shaded images with shadows and reflections at interactive rates.

Line drawing is always faster that shaded rendering. You can also increase the drawing speed by reducing the number of facets. If your objects have lots of little parts that don't need to be drawn in full detail unless they are very close, then you can raise the min_feature_size which tells Hypercosm to draw small objects as their bounding boxes instead of drawing them in full detail.

High-Quality Animation

If you wish to create high-quality animations, a common technique is to produce a quick, low-quality animation first to check the camera and object movements, and sometimes the shading, before going on to invest a large amount of computer time on a finished product. If you're producing shaded images, the more features that you enable, the slower the rendering becomes. The shadows, reflections, and refractions in particular slow things down because they use ray tracing to simulate light ray behavior. Face shading is faster than vertex shading, and both face and vertex shading are faster than pixel shading. The slowest but most realistic shading mode is ray tracing (used when facets = 0).

Animation with the Hypercosm System

When you animate images with Hypercosm, you create a series of pictures that display in rapid succession. You could create a series of individual, slightly differing pictures and run them in order, or you can use Hypercosm's modeling and rendering commands to change an existing image. For example, to create an animation of a rotating sphere, you can create the sphere, then repeatedly use the rotate transformation to rotate the image by a little bit at a time.

When you animate images you can change all kinds of things. For example you can:

- Animate shapes or light sources using transformation commands such as move, rotate, skew, etc.
- Move around a shape by changing the camera positioning variables eye, lookat and roll.
- Zoom in and out on a scene by changing the field_of_view variable.

Unlike some graphics animation packages, Hypercosm does not include a series of canned animation commands like *turn* or *flip*. While these commands can be useful when you first start to work with animations, sooner or later they limit your creativity. Instead, Hypercosm animations require you to do *some* programming. You can produce effective animations with only a minimum of programming, or you can make your animations as complex as you like, using Hypercosm's OMAR programming language. This approach, while initially a bit harder to learn, ultimately yields animations that are far richer than those available with other packages.

Anims

Hypercosm animations make use of an OMAR procedure known as an *anim*. An anim is a section of code that generates a sequence of pictures. Inside an anim is a looping statement that tells the computer to create pictures repeatedly. Each picture differs slightly from the one before it (because you specify different parameters or conditions). You can do this by using a for loop to create a fixed number of frames, or by using a while loop to create pictures continuously while a certain condition is true.

An anim is just like a verb procedure except that it is allowed to call pictures. The graphical procedures (anims, pictures, and shapes) may call the non-graphical procedures (verbs and questions) but not vice versa. Note: an anim cannot directly call a shape; in order to use a shape, an anim must call a picture that calls that shape.

Double Buffering

If you are creating still pictures, you usually want to be able to watch the computer draw the picture. Watching the computer draw lets you track the progress of the rendering, and gives you some idea of how long it will take to complete. It's also convenient because if you see something you don't like—for example, an object that's the wrong color—you can stop the rendering and fix the problem instead of waiting for the image to be completed.

When you do real-time animation, however, the situation is different. You don't really want to see the computer draw each picture; you only want to see the finished images. To make this possible, Hypercosm uses a technique called *double buffering*.

Double buffering means that there are two buffers to hold the images. One buffer is for viewing and the other one is for drawing. The front buffer holds the image that you see on the screen and always holds a completed image. The back buffer is where the computer stores the image that it is busy drawing. When the computer is finished rendering the image, it swaps the buffers so you see the newly completed image.

Figure 6-1: The Double Buffer Variable & Default

```
boolean double_buffer is false;
```

Enable this process by setting the double_buffer variable to true. For very slow animations, you may still wish to use single buffering so you can watch the images being created.

Parameter-Based Animation

The parameters with which a shape is created define a particular instance of that object. When objects are animated, the objects in the scene and/or the relationship between objects in the scene must change for the images to change. Therefore, the animation is the result of the changing parameters.

For example, you might model a car where the rotation of its wheels is a function of a distance parameter that tells how far the car has moved. When the distance parameter is changed, the wheels turn. To animate the car, you need a loop that repeatedly calls a picture containing the car, and the picture must pass a changing distance parameter to the car to make the wheels change from frame to frame.

Another example is a time parameter that regulates the changes in the animation. The animation is a simple loop incrementing the time parameter that is passed to the picture. The picture defines where objects are in terms of the time parameter. To make the animation run more slowly and smoothly, you make the time parameter increase by smaller increments.

Animation Examples

The following listings demonstrate some Hypercosm animations. The first one, labeled **Animating the Eye Location**, produces a rotating goblet that moves in and out

Listing 6-1: Animating the Eye Location

```
do rotating_glass;

include "3d.ores";

shape goblet is
        hyperboloid1 with
                end1 = <0 0 .1>; end2 = <0 0 .6>;
                radius1 = 1; radius2 = .2;
        end;
        hyperboloid1 with
                end1 = <0 0 .6>; end2 = <0 0 3>;
                radius1 = .2; radius2 = 1;
        end;
        hyperboloid1 with
                end1 = <0 0 .7>; end2 = <0 0 3>;
                radius1 = 0; radius2 = .9;
        end;

        ring with
                center = <0 0 3>; normal = <0 0 1>;
                inner_radius = .9; outer_radius = 1;
        end; // rim

        cylinder with
                end1 = <0 0 0>; end2 = <0 0 .1>; radius = 1;
        end; // base of goblet

        disk with
                center = <0 0 0>; normal = <0 0 1>; radius = 1;
        end; // bottom of goblet
end; // goblet

picture glass_picture with
        field_of_view = 35;
        lookat = <0 0 1.5>;
is
        distant_light from <1 -3 2>;
        goblet with color = light orange; end;
end;        // glass_picture
```

Listing 6-1: Animating the Eye Location (Continued)

```
anim rotating_glass with
        double_buffer is on;
is
        scalar angle = 0;
        scalar x, y, z;

        while true do
                // eye rotates around glass in an ellipse
                x = sin angle * 5;
                y = cos angle * 10 + 5;
                z = 5;

                glass_picture with
                        eye = <x y z>;
                        roll = angle;
                end;

                angle = itself + 5;
        end;
end;        // rotating_glass
```

The following animation shows a ringed planet like Saturn orbiting around a sun, viewed from the point of view of another orbiting planet.

Listing 6-2: Orbiting Planet

```
do moving_solar_system;

include "3d.ores";

shape saturn is
        sphere with color = aqua; end;
        ring with
                inner_radius = 1.5; outer_radius = 2;
                normal = <.5 -.3 1>;   color = yellow;
        end;
end;        // saturn

picture solar_system
        vector location1, location2, location3;
with
        eye = location3;
        lookat = location1;
        field_of_view = 100;
        ambient = white * .1;
is
        sphere with material is constant_color yellow; magnify by 2;  move to location1; end;   // star
        point_light with brightness = 8; move to location1; end;                                // star light
        saturn with move to location2; end;                                                     // planet
end;        // solar_system
```

Listing 6-2: Orbiting Planet (Continued)

```
anim moving_solar_system with
        double_buffer is on;
is
        vector v, acceleration;
        scalar k = .05;          // change this to adjust orbit speed
        vector star_location1 = <-4 0 0>, star_velocity1 = <0 .05 0>;
        vector star_location2 = <10 0 0>, star_velocity2 = <0 -.1 0>;
        vector star_location3 = <3 -25 8>, star_velocity3 = <.2 0 0>;

        while true do
                // compute star and planet attraction to each other
                v = star_location2 - star_location1;
                acceleration = v * (k / (v dot v));
                star_velocity1 = itself + (acceleration / 10);
                star_velocity2 = itself - acceleration;

                // compute third party's attraction to star
                v = star_location3 - star_location1;
                acceleration = v * (k / (v dot v));
                star_velocity3 = itself - acceleration;

                star_location1 = itself + star_velocity1;
                star_location2 = itself + star_velocity2;
                star_location3 = itself + star_velocity3;

                solar_system star_location1 star_location2 star_location3;          // draw picture
        end;
end;    // moving_solar_system
```

Mouse-Controlled Anims

The animations created in the examples above play out like movies: every time they are run they follow the exact same pattern. A more powerful and interesting animation would allow the user to control it somehow. Using OMAR, you can program into your animations a limitless range of interactivity.

The easiest way to turn a static 3D image into an interactive animation is to use a *mouse-controlled anim*. Most shapes or pictures you define can simply be 'plugged in' to one of the mouse-controlled anims to enable three different useful interactions. How you can control the interactions depends on how many buttons your mouse has:

Table 6-1: Interactions Provided by Mouse-Controlled Anims

Interaction	On a one-button mouse, hold down:	On a two-button mouse, hold down:	On a three-button mouse, hold down:
Spin—Rotates the shape or picture about the origin.	mouse button	left mouse button	left mouse button
Pan—Pans the camera around.	Control + mouse button	right mouse button	right mouse button
Zoom—Moves the eye forward and backwards.	Control + Shift + mouse button	both mouse buttons	middle mouse buttons

With a two-button mouse, you can also use the controls specified for a one-button mouse, and with a three-button mouse, you can also use the controls specified for a one or two-button mouse. When using one-button controls, you can use the Alt/Option key in place of the Control key.

Mouse-controlled anims automatically turn the double buffer on and also implement the Hypercosm procedure called `check_keys`. This means that when

using a mouse-controlled anim, pressing various special keys will change rendering parameters:

Table 6-2: Special Keys When Using a Mouse-Controlled Anim

Key	Effect
p	changes render_mode to pointplot
w	changes render_mode to wireframe
h	changes render_mode to hidden_line
s	changes render_mode to shaded
l	changes edges to silhouette
o	changes edges to object
a	changes edges to all
f	changes shading to face
v	changes shading to vertex
3	turns on/off stereo view (sets stereo to 0 or 5)

There are two basic mouse-controlled anims:

- mouse_controlled_shape takes a shape as its parameter. It also adds lighting for you. The lighting source is stationary in world coordinates, even as the shape is spun around.
- mouse_controlled_picture takes a picture as its parameter and has no added lighting. If you want a lighted scene, you must add light yourself in your picture definition. Added lighting sources will move with respect to the objects in the picture, so the same side of objects will always be lit up.

Both mouse_controlled_shape and mouse_controlled_picture are defined in the file **anims.ores**, which is not included automatically with **3d.ores**, so in order to use a mouse-controlled anim, you must include **anims.ores** in your OMAR file.

The following two example listings demonstrate how to use mouse-controlled anims.

Listing 6-3: Using mouse_controlled_shape

```
do interactive_ball;

include "3d.ores";
include "anims.ores";          //*** "anims.ores" must be included to use a mouse-controlled anim. ***

shape ball is
    sphere with color = orange; end;
end;

anim interactive_ball is       // No need to set double_buffer: mouse_controlled_shape does it for you.
      mouse_controlled_shape ball;
end;
```

Listing 6-4: Using mouse_controlled_picture

```
do interactive_balls;

include "3d.ores";
include "anims.ores";          //*** "anims.ores" must be included to use a mouse-controlled anim. ***

shape ball is
      sphere with color = orange; end;
end;

picture balls is
      distant_light from <1 -2 1>;
      ball;
      ball with move to <2 -1 0>; end;
      ball with move to <-2 -1 0>; end;
end;

anim interactive_balls is      // No need to set double_buffer: mouse_controlled_picture does it for you.
      mouse_controlled_picture balls;
end;
```

Modifiable Shapes & Pictures

If a shape or picture has parameters, that means that every time that shape or picture is used, it may have a different definition. For example, a sphere, which has parameters such as center and radius, can be given a radius of any size when it is used. We therefore call such shapes and pictures *modifiable*.

A mouse-controlled anim cannot accept a modifiable shape or picture as its parameter because it needs something with a constant definition. In order to use a mouse-controlled anim with a modifiable shape or picture, you need to encapsulate the shape or picture inside another one that doesn't take any parameters.

Suppose you define a shape like this:

```
shape ball with          // The use of with reveals that this shape takes parameters
                         // and is therefore modifiable.
      scalar spin = 0;
is
      sphere with
            color = white;
            rotate by spin around <1 0 0>;
      end;
end;
```

You cannot then do the following:

```
mouse_controlled_shape ball;          // causes error
```

because ball is modifiable. Similarly, you can't say:

```
mouse_controlled_shape sphere;
```

because sphere is modifiable (it takes parameters such as center and radius). But, you *can* put ball in a non-modifiable shape, and then use that:

```
shape non_modifiable_ball is          // note: no parameters
      ball with spin = 45;
end;
```

It's okay, then, to say:

```
mouse_controlled_shape non_modifiable_ball;
```

Or, you could put ball in a non-modifiable picture:

```
picture non_modifiable_picture is          // again: no parameters
      ball;
      distant_light from <1 -3 4>;
end;
```

and then say:

```
mouse_controlled_picture non_modifiable_picture;
```

Customizing Mouse Controls

If you want to customize how your animations respond to a mouse, you need procedures that can tell you where the cursor is and whether mouse buttons are being pressed. The details of how the animation responds to the mouse are then entirely up to you to program.

Hypercosm provides three different procedures for getting data from the mouse:

- get_mouse returns the location of the mouse cursor on the screen.
- mouse_down indicates whether a mouse button is pressed down or not.
- get_click returns information about individual mouse clicks.

The following sections describe these procedures in more detail.

Cursor Location

The question procedure, get_mouse, returns a vector that contains the location of the mouse cursor relative to the center of the viewing window. The x-component of the vector indicates the horizontal position of the cursor, and the y-component indicates the vertical position. If the cursor is positioned over the upper left corner of the viewing window, get_mouse returns <-1 1 0>. If the cursor is positioned over the lower right corner, get_mouse returns <1 -1 0>.

Figure 6-2: The Cursor Location Procedure

```
vector question get_mouse;
```

The components of the vector may be extracted using the dot operator. The third component of the vector is meaningless since most mice only have two degrees of freedom. If you want the mouse location in other units, such as relative to the screen dimensions or in the range of (0 to 1), then you can write your own functions to convert the coordinates.

Cursor location can be used to adjust any number of scene attributes. You could rotate a scene, change viewing parameters, or even change an object's color. In the example below, changing the cursor's horizontal location rotates an object about the Z-axis, and changing the cursor's vertical location rotates it about the X-axis.

Listing 6-5: Using get_mouse *to Rotate an Object*

```
do thing_anim;

include "3d.ores";

shape thing is
        block;
        cylinder with end1 = <-1.5 0 0>; end2 = <1.5 0 0>; radius = .5; end;
end;        // thing

picture thing_picture with
        eye = <0 -8 0>;
is
        scalar x = get_mouse dot <1 0 0>;                   // horizontal mouse location
        scalar y = get_mouse dot <0 1 0>;                   // vertical mouse location

        distant_light from <1 -3 2>;
        thing with
                rotate by y * -180 around <1 0 0>;          // use vertical location to rotate about X-axis
                rotate by x * 180 around <0 0 1>;           // use horizontal location to rotate about Z-axis
        end;
end;        // thing_picture

anim thing_anim with
        double_buffer is on;
is
        while true do thing_picture; end;
end;        // thing_anim
```


The Mouse Button

Another useful procedure is the mouse_down question, which takes a mouse button number as its parameter, and then checks whether the corresponding button is pressed. If the button is pressed, mouse_down answers true, otherwise it answers false.

Figure 6-3: The Mouse Button Procedure

```
boolean question mouse_down
        button integer number = 1;
end;
```

The button on a one-button mouse is button 1, which is the default button for mouse_down. On a two or three-button mouse, the left button is button 1, and the right button is button 3. On a three-button mouse, the middle button is button 2.

Table 6-3: Mouse Button Numbers for Different Mouse Types

Button Number	One-Button Mouse	Two-Button Mouse	Three-Button Mouse
1	mouse button	left mouse button	left mouse button
2	—	—	middle mouse button
3	—	right mouse button	right mouse button

If you're using a one or two-button mouse, then mouse_down button 2 will always answer false because no button 2 exists for your system.

Listing 6-6: Using get_mouse *and* mouse_down *to Rotate and Move an Object*

```
do thing_anim;

include "3d.ores";

shape thing is
        block;
        cylinder with
                end1 = <-1.5 0 0>;
                end2 = <1.5 0 0>;
                radius = .5;
        end;
end; // thing

picture thing_picture
        vector location, orientation;
with
        eye = <0 -8 0>;
is
        distant_light from <1 -3 2>;
        thing with
                rotate by (orientation dot <1 0 0>) around -<1 0 0>;
                rotate by (orientation dot <0 0 1>) around <0 0 1>;
                move to location;
        end;
end; // thing_picture
```

Listing 6-6: Using get_mouse *and* mouse_down *to Rotate and Move an Object*

```
anim thing_anim with
      double_buffer is on;
is
      vector old_mouse, new_mouse, delta;
      vector orientation = <0 0 0>, location = <0 0 0>;
      scalar dx, dy;

      old_mouse = get_mouse;
      while true do
            new_mouse = get_mouse;
            delta = new_mouse - old_mouse;
            old_mouse = new_mouse;
            dx = delta dot <1 0 0>;
            dy = delta dot <0 1 0>;

            if mouse_down button 1 then
                  // if button number 1 is down, move object
                  location = location + <dx 0 dy> * 2;
            else
                  // If button number 1 is not down rotate object
                  orientation = orientation + <dy 0 dx> * 360;
            end;

            thing_picture location orientation;
      end;
end;        // thing_anim
```


Mouse Clicks

The mouse_down procedure is useful for interactions that require the user to press and hold down a mouse button, as in the example listing above. However, mouse_down is not very useful for interactions that require the user to click or double-click a mouse button.

This is because mouse_down only checks the *current* state of the mouse. If the animation frame rate is slow, then mouse_down might check the mouse too infrequently and might miss a click. If the frame rate is very fast, then mouse_down could check the mouse too frequently, and might report several clicks when the user has actually only clicked once.

Another significant problem with mouse_down is that, because it only checks the current mouse state, it can never tell you exactly when and where a mouse button was pressed or released. If the example animation above was running very slowly, you could drag your mouse across the graphics window very quickly, and mouse_down might not detect that the mouse was pressed at all, or, it might detect that the button was pressed once, but the locations that get_mouse would return would not accurately reflect the locations of where the mouse was actually pressed.

Instead of using mouse_down to detect mouse clicks then, you should generally use get_click, a procedure that lets you know accurately whether the mouse has

been clicked or not, and also tells you where the click occurred and what kind of click was made: a down click (pressing the button down), a double click (pressing the button twice in quick succession), or an up click (letting the button up after it has been pressed down).

Figure 6-4: The Mouse Click Type & Procedure

```
enum click is down, double_click, up;

click type question get_click
return with
        integer button;
        vector location;
        boolean shift;
        boolean alt;
        boolean control;
        boolean caps_lock;
end;
```

The Mouse-Event Queue

To understand how get_click works, and to be able to use it properly, you should first understand the concept of a *mouse-event queue*. Whenever the mouse is clicked, the computer enters an *event* into its mouse-event queue. This *queue* works like a ticket line or a conveyor belt. New items are placed at the end, and items at the front are serviced and removed.

To 'service and remove' an event from the mouse-event queue, you use the get_click procedure. When get_click is called, it removes an event from the event queue and returns various information about the event. The event that is removed is always the *oldest* event that has not yet been handled, because the most *recent* event is always placed at the end of the queue.

For example, imagine you're running an animation that calls get_click once per frame. Between two calls to get_click, you double-click on the graphics window. By double-clicking, you enter *four* events into the mouse-event queue: the initial down click, the following up click, the second down click (which is tabbed as a double click), and the final up click. The initial down click is the oldest of the four events, and the final up click is the most recent.
The next call to get_click then answers down, because the oldest event in the queue is your initial down click. Following calls to get_click answer up, then double_click, then up again. Remaining calls to get_click, assuming you do not click any more after your first double-click, all answer none because the queue is empty.

Using get_click

The get_click procedure has a number of useful optional return parameters. The button parameter indicates which mouse button was clicked. The location parameter indicates where the cursor was when the click occurred, using the same format as the get_mouse return value. The boolean parameters—shift, alt,

control, and caps_lock—indicate which of the corresponding modifier keys were pressed when the click occurred (with alt corresponding to the Option key on Macintosh keyboards).

When you call get_click, you can use any combination of the return parameters, and you needn't use any at all if they aren't needed. The syntax for using get_click and its parameters is demonstrated in the following listing.

The listing produces a spinning metallic cylinder. You can change the color of the ground below the cylinder by clicking on it, or change the sky's color by clicking on it. To change the cylinder's color, press Shift and click anywhere. You can also turn fog on and off by pressing Control while clicking.

Listing 6-7: Using get_click *to Change Colors in a Scene*

```
do color_picker;

include "3d.ores";

color type colors[0..18] = [(dark red) (dark green) (dark blue) red green blue yellow cyan magenta
                            black charcoal grey white sky_blue olive gold rust eggplant lavender];

picture color_scape is
      static integer sky_color = 13;                    // Default sky color is 13: sky blue.
      static integer ground_color = 1;                  // Default ground color is 1: dark green.
      static integer thing_color = 15;                  // Default cylinder color is 15: gold.
      static integer t = 0;                             // This variable is the amount of the
                                                        // cylinder's spin.
      integer button;
      vector click_loc;
      boolean shift, ctrl;

      // *** Syntax for using optional return parameters: ***
      //
      click type click is
            (get_click return with
                  click_loc = location;
                  static shift is shift;     // "static shift" is the variable, "shift" is the parameter.
                  ctrl is control;);         // Note: no end keyword because the procedure call is
                                             // an expression, not a statement.

      if click is down or click is double_click then
            if ctrl then                     // If Control is pressed, toggle fog_factor.
                  if fog_factor = 0 then fog_factor = 150;
                  else fog_factor = 0;
                  end;
            else
                  if shift then thing_color = (itself + 1) mod (num colors); // Change cylinder's color.
                  elseif click_loc.y > 0 then sky_color = (itself + 1) mod (num colors);
                                                                  // Change sky's color.
                  else ground_color = (itself + 1) mod (num colors);       // Change ground's color.
                  end;
            end;
      end;
```

Listing 6-7: Using get_click *to Change Colors in a Scene (Continued)*

```
        background = colors[sky_color];                    // Set the "sky's" color.
        distant_light from <-1 -2 3>;

        sphere with
                vmax = 0;
                scale by .2 along <0 0 1>;
                magnify by 100;
                color = colors[ground_color];              // Set the ground's color.
        end; // ground

        cylinder with
                scale by 2 along <0 0 1>;
                magnify by 5;
                rotate by t around <0 1 0>;
                material is metal colored colors[thing_color];   // Set the cylinder's metal color.
        end; // spinning cylinder

        t = itself + 2;                                    // This increments the cylinder's spin.
end; // color_scape

anim color_picker with
        double_buffer is on;
        eye = <0 -50 0>;
is
        while true do
                color_scape;
        end;
end; // color_picker
```

Customizing Keyboard Controls

You can also use the keyboard to control animations. The built-in I/O procedure, read, is unsuitable for animations because it waits for the user to enter data and then press Enter or Return. What you need instead is a keyboard procedure that reports the instantaneous state of the keyboard without waiting for the Enter key. For this purpose, Hypercosm provides two keyboard procedures, get_key and key_down, that function like the mouse procedures get_click and mouse_down.

Figure 6-5: The Keyboard Procedures

```
integer question get_key
return with
        boolean shift;
        boolean alt;
        boolean control;
end;

boolean question key_down
        integer key;
end;
```

Just as get_click provides information about events in the mouse-event queue, get_key provides information about events in the keyboard-event queue. Each time you strike a key on your keyboard, your computer takes note by entering an event in the keyboard-event queue. When get_key is called, it removes the oldest event remaining in the queue and returns information about it. The integer that get_key returns is the *keycode* of the key that was struck. Keycodes are similar to mouse button numbers: there is a different keycode for each of the buttons on the keyboard.

Listing 6-8: Program to Report the Keycode of Any Key That Is Struck

```
do get_keycodes;

include "3d.ores";

anim get_keycodes is
      picture blank is
      end;

      while true do
            integer k = get_key;

            blank;                              // get_key requires an open graphics window.
            if k <> 0 then                      // If some key was struck, then write the keycode.
                  write "keycode = ", k,  ;
            end;
      end;
end;        // get_keycodes
```


In addition to returning a keycode, get_key also has optional return parameters that indicate whether a modifier key—Shift, Alt/Option, Control, or Caps Lock—was pressed down when a key was struck. There is no caps_lock parameter. Instead, the shift parameter is adjusted accordingly: if Caps Lock is down and a letter key is struck, then shift will be true; otherwise, the Caps Lock key has no effect.

The second keyboard procedure, key_down, works very much like mouse_down. It takes a keycode as its parameter, and then checks whether the corresponding key is pressed. If that key is pressed, key_down answers true, otherwise it answers false.

Converting Between Characters and Keycodes

Since all computers have different keyboards, a certain key may have different keycodes on different computers, or may not even exist on certain computers. On a Macintosh, for example, the keycode for the A key is 0, whereas on a UNIX machine, it is 97. If you were to use only your system's keycodes to refer to keys, then your programs would not be portable between different makes of computers.

To ensure that your OMAR code is portable, Hypercosm provides its own intermediary keycode. You can use the procedures key_to_char and char_to_key to convert between characters and the Hypercosm keycode. Then, the Hypercosm system can handle the conversion between the Hypercosm keycode and system-dependent keycode internally. Thus, your OMAR code never has to take into account system-dependent keycodes.

Figure 6-6: Character/Keycode Conversion Procedures

```
char question key_to_char
        integer key;
with
        boolean shift is false;
end;

integer question char_to_key
        char c;
end;
```

The question char_to_key takes a character parameter and returns its keycode. The question key_to_char takes a keycode and returns its corresponding character. If you set its shift parameter to true, key_to_char returns the character that corresponds to a shifted key.

Note that there are many special keys on a keyboard, such as F1 or Page Up, that do not correspond to any printable character and cannot therefore be converted to chars. For this reason, the conversion procedures only work for the letter, number, and symbol keys, and for certain special keys such as the Space, Tab, and Enter keys. If you need to know the keycode of a special key such as F1, then you can look in the resource file, **keycode.ores**, where you can find the conversion procedure definitions together with documented conversion tables.

Listing 6-9: Program to Report the Corresponding Character of Any Key That Is Struck

```
do write_chars;

include "3d.ores";

anim write_chars is
        picture blank is end;

        while true do
                boolean shift;
                integer k = (get_key return with static shift is shift;);
                char c is (key_to_char k with shift is static shift;);

                blank;                          // get_key requires an open graphics window.
                if k <> 0 then                  // If some key was struck, then write its corresponding character.
                        write "you typed ", c , ;
                end;
        end;
end;        // write_chars
```

The following example uses some advanced programming techniques to create a robot that you can control with your keyboard and also manipulate using a mouse-controlled anim.

Listing 6-10: A Keyboard-Controlled Robot

```
do interactive_robot;

include "3d.ores";
include "anims.ores";          // "anims.ores" must be included to use a mouse-controlled anim.

shape robot with
      scalar base_rotation = 0, arm1_angle = 0, arm2_angle = 0;
is
      shape base is
            cylinder with end1 = <0 0 0>; end2 = <0 0 .1>; radius = 1; end;
            cone with
                  end1 = <0 0 .1>; radius1 = 1;
                  end2 = <0 0 .5>; radius2 = .4;
            end;
            sphere with
                  material is plastic colored white;
                  center = <0 0 .5>; radius = .4;
            end;
      end;        // base

      shape arm
            scalar angle;
      is
            cylinder with end1 = <0 0 0>; end2 = <0 0 1>; radius = .1; end;
            cylinder with end1 = <-.2 0 1>; end2 = <.2 0 1>; radius = .2; end;
            cylinder with
                  end1 = <0 0 0>; end2 = <0 0 1>; radius = .1;
                  rotate by angle around <1 0 0>; move to <0 0 1>;
            end;
      end;        // arm

      base with material is plastic colored red; end;
      arm arm2_angle with
            material is plastic colored blue;
            move to <0 0 .4>;
            rotate by arm1_angle around <1 0 0>;
            rotate by base_rotation around <0 0 1>;
            move to <0 0 .5>;
      end;
end;        // robot

anim interactive_robot with
      eye = <2 -6 4>;
      lookat = <0 0 1>;
is
      integer angle1_key = char_to_key "j";                    //angle1 rotates arm around base.
      integer reverse_angle1_key = char_to_key "u";
      integer angle2_key = char_to_key "k";                    //angle2 raises arm up and down.
      integer reverse_angle2_key = char_to_key "i";
      integer angle3_key = char_to_key "l";                    //angle3 bends 'elbow.'
      integer reverse_angle3_key = char_to_key "o";
      scalar angle1 = 60, angle2 = 10, angle3 = 30;
```

Listing 6-10: A Keyboard-Controlled Robot (Continued)

```
    verb check_robot_keys is
        if key_down angle1_key then
            angle1 = angle1 + 5;
        elseif key_down reverse_angle1_key then
            angle1 = angle1 - 5;
        elseif key_down angle2_key then
            angle2 = angle2 + 5;
        elseif key_down reverse_angle2_key then
            angle2 = angle2 - 5;
        elseif key_down angle3_key then
            angle3 = angle3 + 5;
        elseif key_down reverse_angle3_key then
            angle3 = angle3 - 5;
        end;
    end;        // check_robot_keys

    picture scene is
        distant_light from <-1 -3 2>;
        robot with
            base_rotation = angle1; arm1_angle = angle2; arm2_angle = angle3;
        end;
    end;        // scene

    // Now, the keyword doing can be used to run check_robot_keys
    // while using a mouse-controlled anim.
    mouse_controlled_picture scene doing check_robot_keys;
end;        // interactive_robot
```

The Time Procedure

Use the question procedure, **get_time,** to attain the time from the system clock. This may be useful to regulate the speed of real-time animations so that they run consistently on different machines. The time question returns a vector that contains the hours, minutes, and seconds as floating point values.

Figure 6-7: The Time Procedure

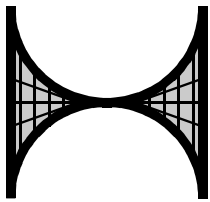
```
vector question get_time;
```

The resolution of the time question is system-dependent because the resolution of system clocks differ. Usually it is accurate to 1/60 of a second.

Note that the hours, minutes and seconds are scalar (floating point) values. This means that if it is 5:30 am, for example, the number of hours could be reported as 5.5 hours. However, on some systems, the hours and minutes are reported as integral values, and only the seconds are given fractional precision. This means that if you were to use **get_time** to draw an animated real-time clock, the hour and minute hands might move smoothly or in steps, depending on your system and on your implementation. If you wanted to ensure that the hands move in steps to make the clock 'tick', you would have to round or truncate fractional values into integers.

Listing 6-11: Time-Controlled Clock Animation

```
do clock_anim;

include "3d.ores";
include "time.ores";          // *** "time.ores" must be included to use the get_time procedure ***
include "anims.ores";

shape clock_face is
      disk with normal = <0 1 0>; end;
      cylinder with end1 = <0 0 0>; end2 = <0 -.4 0>; radius = .05; end;

      // hour marks:
      for integer count = 1..12 do
            block with
                  magnify by .1; scale by .5 along <0 0 1>;
                  move to <.8 0 0>;
                  rotate by count * 360 / 12 around <0 1 0>;
                  material is chalk colored grey;
            end;
      end;
end;       // clock_face

shape clock with
      vector time = <0 0 0>;
is
      scalar hours = time.x, minutes = time.y, seconds = time.z;

      // clock_body
      torus with
            inner_radius = .95; outer_radius = 1.2; normal = <0 1 0>;
            material is golden;
      end;
      clock_face with material is plastic colored white; end;

      // hands
      triangle <-.1 0 0> <.1 0 0> <0 0 .6> with
            rotate by hours / 12 * 360 around <0 1 0>; move to <0 -.2 0>;
            color = magenta;
      end;        // hour hand
      triangle <-.05 0 0> <.05 0 0> <0 0 .8> with
            rotate by minutes / 60 * 360 around <0 1 0>; move to <0 -.25 0>;
            color = blue;
      end;        // minute hand
      triangle <-.025 0 0> <.025 0 0> <0 0 .6> with
            rotate by seconds / 60 * 360 around <0 1 0>; move to <0 -.3 0>;
            color = red;
      end;        // second hand
end;       // clock

picture clock_picture is
      distant_light from <.3 -1 .5>;
      clock with time = get_time; end;
end;       // clock_picture

anim clock_anim with
      eye = <1 -4 1>;
is
      mouse_controlled_picture clock_picture;
end;       // clock_anim
```



Glossary

A

Aggregate Shape

An aggregate shape is a shape that is defined by the user to be composed of a number of primitive shapes or other aggregate shapes. Primitive shapes are the built-in shapes that are not user-defined and can not be changed.

Aliasing

An artifact resulting from the fact that images are represented by computers as a grid of pixel values. Ideally, each pixel value should represent the exact average intensity of all of the colors that lie in the pixel. However, computer graphics typically only sample an image once per pixel, so noticeable artifacts often occur. Such artifacts are called *aliasing*.

One common artifact is a staircase effect, often called the *jaggies*, that is noticeable along diagonal lines and edges. Another artifact occurs when an image has a repetitious pattern that is finer than the grid of pixels that are used to represent it. In this case, the discrete sampling at pixels causes large, obvious patterns to be visible because of a spatial resonance that occurs between the pixel grid and the pattern in the image.

Ambient Light

Ambient light is the diffuse, scattered light that comes from all directions and illuminates shapes regardless of the orientation or location of their surfaces. Ambient light is the reason that surfaces in shadow are not perfectly dark. Surfaces receive light scattered by all other nearby shapes and they, themselves, scatter light to contribute to the ambient lighting of other shapes.
Outer space is a setting where there is very little ambient light because there are few nearby shapes and no atmosphere to scatter light. When you look at the moon, the dark side often appears completely black because the only other substantial shape around to scatter the sun's light back to the moon is the Earth.

Anaglyph

The anaglyph technique is used to present stereo images by compositing

two complimentary-colored images and using colored filters over the eyes to separate them. This technique was popular in the movies of the 1950s. The 3D effect is achieved by presenting a slightly different image to the right and left eyes. Two grey-scale images are tinted by two complimentary colors, commonly blue and red, and then combined so they overlap on the screen. Then, the images are separated again with colored glasses so each eye sees just one of the images. This technique does have drawbacks: it cannot present 3D images in full color, and differently-colored filters tend to cause eyestrain.

Argument

In a mathematical sense, *arguments* are the values passed into a function. For example, in the function call, sin (50), the argument of the function is 50. In OMAR, arguments to procedures are typically called *parameters*. *Program arguments* are the text commands passed into a program from a command line or HTML code.

B

Bump Mapping

Bump mapping is a technique that is used to modify a surface's properties in such a way as to make it look bumpy. Instead of actually changing the geometry of a surface, however, bump mapping works by perturbing the effective normals of the surface. When shading is calculated for the surface, then, the perturbed normals are used to calculate reflectance instead of actual normals. This has the effect of making a surface shade as if it were bumpy without actually changing the shape of the underlying object.

C

Concave

A closed two or three-dimensional shape has the property of being concave if a line can pass through the shape and intersect it in more than two points. Concave shapes can be said to 'curve in on themselves.' Any shape with an indentation is concave. For example, a two-dimensional moon shape is concave. A torus and a bowl shape are concave shapes. Shapes that are not concave are, by definition, convex.

Convex

A convex shape is any closed two or three- dimensional shape that never curves in on itself or has any indentations. If a line passes through a convex shape, then it will always intersect the shape in exactly two points. Examples of convex shapes are circles, spheres and cubes. A mirror whose surface bulges out slightly is called a convex mirror. Shapes that are not convex are, by definition, concave.

Coordinates

Coordinates are a set of numbers that uniquely specify a point's location in space relative to a certain frame of reference. The frame of reference is called the *coordinate system* and may be thought of as a set of orthogonal axes that come together at a point called the *origin*. Each coordinate indicates the distance between a particular point and the origin along one of the axes. The number of axes and the number of coordinates are determined by the number of dimensions.

Cross Product

The cross product is a mathematical operation that is defined between two three- dimensional vectors. The result is a new vector that is perpendicular to both of the other two vectors. The length (magnitude) of the cross product is equal to the product of the lengths of the two operands times the sine of the angle between them. The cross product is sometimes called the vector product because the result is a vector. The cross product is calculated as follows:

vector1: (a b c)
vector2: (d e f)

vector1 cross vector2 = (i j k)
where
i = (b f) - (e c)
j = (c d) - (a f)
k = (a e) - (b d)

D

Dot Product

The dot product is a mathematical operation that is defined for two vectors of any dimension. The result of a dot product is always a scalar, so the dot product is sometimes called the scalar product. Geometrically, the length of the dot product is equal to the product of the magnitudes of the operands times the cosine of the angle between them. The dot product of a vector with itself is therefore equal to square of its length. To compute the dot product, you compute the product of each component of one vector and the corresponding component of the other vector and sum them all together. For two three-dimensional vectors, you compute the Dot Product as follows:

vector1: $(u_1\ u_2\ u_3)$
vector2: $(v_1\ v_2\ v_3)$

vector1 dot vector2 =
$(u_1 * v_1) + (u_2 * v_2) + (u_3 * v3)$

Double Buffer

Double buffering is a technique that enables animations to be displayed as a smoothly changing sequence of completed images. Without the double buffer, you would see each image in the sequence as the computer is drawing it. If the images take a noticeable amount of time to draw, then you could watch each image being created. If the images can be rapidly drawn, then you would see a flickering as the screen is cleared and each new image is quickly created.

The double buffer technique eliminates these distracting effects and enables you to display only the finished image on the screen. The image that you see on the screen is actually held in a block of memory known as the *front buffer*. While you view the front buffer, the next image in the sequence is assembled by the computer in a second, hidden buffer that is called the *back buffer*. When that image is complete, the front and back buffers are quickly swapped so that you can see the new image in the front buffer and you can clear the back buffer for the next image.

E

Edge

An edge is the line that joins two vertices together and the place where two faces come together. All of the edges of polygons and other faceted primitives such as triangles and meshes are defined to be straight lines in three-dimensional space. If two shapes intersect, then new edges will be formed where they intersect. If your models are all represented as faceted polyhedra,

then all edges will be straight lines and therefore all of the silhouette edges and intersection edges will also be straight lines. If unfaceted models such as spheres and cylinders are used, then the silhouette edges and intersection edges will be curves. Whether the edges appear as straight lines on the screen depends upon the projection that is used.

Extrusion

An extrusion is a shape that is created by moving a two-dimensional shape through a path in space. The three-dimensional shape that results is known as an extrusion after the fabrication process where material is forced out through a specially shaped hole in a die. Extrusions are commonly used to produce 3D letters for logos.

F

Facet

Facets are polygonal faces that can be used to represent the surfaces of shapes. Curved surfaces can not be perfectly represented by the flat facets but may only be approximated. The reason that facets are used to represent curved surfaces is because the mathematics that are used is simpler (linear equations) and faster to solve than the mathematics for perfectly curved surfaces.

Field Of View

The field of view is a measure of how much of your surroundings you can see at once. Usually, the field of view is represented as an angle across your field of vision. For example, the volume of space that you can see with one eye forms a cone extending out infinitely far into space with your eye at the apex. The field of view is the angle between two opposite sides of the cone. In a camera or computer graphic, the field of view is usually more of a pyramid shape, so the field of view is measured as the angle between opposite edges of the pyramid. The exact shape of the viewing region depends upon the projection that is used.

Flat Shading

Flat shading has two related meanings in computer graphics that are often confused. The first meaning refers to a rendering technique, also called *face shading*, whereby the renderer depicts surfaces as a collection of facets where each facet is a constant color. The second meaning has to do with the lighting model that is used in the shading calculations and means that a diffuse, Lambertian lighting model is in use, which gives the surfaces a dull, chalky appearance.

Focal Point

The focal point is the precise location where the light from the outside universe is brought together to a point by a lens or mirror to form an image. After coming together at the focal point, the light diverges onto a recording medium such as film in a camera or the retina in the eye. The lens in the model used by computer graphics is a point, like in a pinhole camera, so the location of the eye is effectively the focal point of the lens.

Fractal

A fractal is a shape that has the property of self-similarity, meaning that it reveals the same general pattern at all scales. The fractal shape contains tiny copies of itself and those copies each contain tinier copies of the same shape ad

infinitum. If you magnify a portion of the fractal, then you see the same shape no matter how much magnification you use. A famous example of fractals is the Mandlebrot set, an exquisitely intricate two-dimensional mathematical shape. Three-dimensional shapes can be fractals as well.

G

Global Illumination

Global illumination is a name for all techniques that are used to compute the appearance of a shape by looking at the effects of its surroundings. There are a number of ways that the environment of a shape can change its appearance, and all come under the topic of global illumination.

One way is through reflection and refraction. Shiny shapes that reflect their surroundings and transparent shapes that transmit light from their surroundings can be simulated by the process of ray tracing. Another way that shapes can be affected by their environment is by shapes illuminating each other through diffuse reflection. This process can be simulated in computer graphics by a technique known as radiosity. Global illumination techniques are more costly and difficult to implement because they must be able to access any portion of the entire database of the scene in order to compute the shading on any surface.

Gouraud Shading

Gouraud shading, also known as *vertex shading*, is a technique developed by Henri Gouraud to render smoothly shaded curved surfaces from a faceted representation. Gouraud shading works by blending the color smoothly across the facets using a linear interpolation between the edges of the facet. The benefit of Gouraud shading is that the shape has a smooth appearance instead of a faceted appearance and it can still be rendered quickly because the mathematics for the surface is linear.

Unfortunately, the way that Gouraud shading blends color doesn't perfectly match the way the color changes across actual curved surfaces, so Gouraud shading still causes visible artifacts. In addition, since underlying surface geometry is still faceted, the silhouette edges of surfaces give away their polygonal representation.

H

Hierarchical Geometry

Hierarchical geometry in a graphical database is the grouping of shapes into aggregate shapes. The hierarchy can be imagined as a tree with the picture object (the entire scene) as the root shape and the primitive shapes at the leaves.

I

Image Plane

The image plane is the surface that an image is projected onto after light rays are focused through a focal point by a lens. The image plane is usually a flat surface in 3D graphics, as in a real camera where the image plane is the surface of the film. In some systems, however, the image plane is actually a curved surface. In your eyes, for instance, the image plane is the spherical surface of the retina.

Interpolation

Interpolation is a technique for estimating what some intermediate values are, given known values at nearby points. Interpolation is used in Gouraud (vertex) shading, which evaluates shading precisely only at vertices and estimates the colors between the vertices by interpolating from the known colors at the vertices. For example, if the color is dark red at one vertex and white at another, then it is very likely that the color in between the two vertices should be a smooth gradation from dark red to red to light red to white.

Interpolation sometimes fails because the thing that you are interpolating changes very rapidly and abruptly between the points that you are sampling. For example, if the surface is textured with a fine pattern, then smoothing the color between the vertices will not give you a good representation of the underlying pattern.

J

Jaggies

Jaggies are an antialiasing artifact that occur frequently in computer graphics because images of non-rectangular shapes cannot be perfectly represented by lighting up rectangular pixels in a grid. For example, when a diagonal line is drawn on a computer screen, the line typically appears as a stair step pattern of pixels. The little saw tooth edges of the line are what are informally referred to as the 'jaggies'.

L

Lathe

A lathe is a shape created by rotating a two-dimensional form around an axis of rotation. The resulting three-dimensional shape has a rotational symmetry about its axis. This type of shape is named after the machining process whereby a piece of material is rapidly turned while a cutting device is used to remove material. Wooden dowels used in furniture or porch railings are often fabricated in this way.

Latitude

The latitude lines are the horizontal lines that encircle a shape around its axis of revolution. Latitude can be distinguished from longitude by remembering that the latitude lines lie flat. Latitude is defined for a sphere as the angular distance north or south from the Equator. The north pole of the Earth is at 90 degrees latitude and the south pole is at -90 degrees latitude. Since the north and south poles are 90 degrees from the equator, the latitude has a range of -90 to 90 degrees. If the latitude goes over 90 degrees or under -90 degrees, it wraps around. For example, if you travel more than 90 degrees north from the equator, you start to get closer to the equator again, so 100 degrees of latitude wraps around to 80 degrees.

Lighting Model

A lighting model is a mathematical formula that is used to predict how light will interact with surfaces. Lighting models attempt to describe the physics that are involved in the interaction, but often the physics are too complicated to model precisely, so a number of simpli-

fying assumptions are made to make the lighting model manageable.

Line Of Sight

The line of sight is the direction in which you are looking. Shapes that are directly in the line of sight will appear in the center of an image and shapes that are at greater angles to the line of sight will appear farther from the center of the image. The line of sight is the vector that is formed between the eye point and the lookat point, which is sometimes also called the *center of vision*.

Linear

Linear is a term that refers to anything pertaining to a line. In a mathematical sense, an equation is linear if its graph is a line. Any equation is linear if it consists only of terms that are of the first degree. That means that none of the independent variables in the equation are raised to a power other than one and there are no non-linear functions involved such as sine or cosine.

Local Coordinates

When you define an aggregate shape, the subshapes are defined in the frame of reference of the aggregate shape. This frame of reference is described by the shape's *local coordinates*. When an aggregate shape is moved around, all of the sub-shapes automatically follow the aggregate shape. This is a great help in modeling complex scenes. Shapes can be created by defining all of their parts relative to the shape without worrying about where the shape will eventually be placed in the scene because when the aggregate shape is moved, all of its parts automatically follow.

Local Variable

A local variable is a variable that exists only inside the scope of a particular procedure. Local variables encourage better, safer programming because if a variables only live within a procedure, you don't need to worry about other procedures inadvertently changing them. Almost all variables in a typical procedural language are local variables.

Longitude

Longitude lines are the vertical lines that run perpendicular to the latitude lines (assuming that the axis of rotation is vertical). On a sphere, longitude is a measure of how far around the globe you have travelled. Since a globe is rotationally symmetrical, you must arbitrarily pick a location from which to measure. By convention, on Earth, we have chosen Greenwich, England. Longitude may run from 0 to 360 degrees. Below 0 degrees or above 360 degrees, it wraps around, so 400 degrees of longitude is the same as 40 degrees of longitude. Longitude lines may be drawn on any shape that is rotationally symmetrical, so all of the quadrics and the torus may be sectioned into longitudinal sweeps.

M

Mandlebrot Set

The Mandlebrot Set is a mathematical entity that is known for its exquisite beauty and complexity. The Mandlebrot Set and related mathematical shapes are known as fractals because they possess the property of self-similarity. This means that the shape has the same structure at all scales. If you zoom into the Mandlebrot Set, tiny Mandlebrot Sets begin to appear which, upon closer

inspection, have all of the intricate detail of the original.

Mesh

A mesh is a general purpose primitive that consists of a group of vertices, edges, and faces that are usually, but not necessarily connected into a surface. Since any surface can be approximated by a group of planar facets, the mesh primitive is the most general-purpose primitive.

Mirror Direction

The direction that a light ray will reflect off of a surface is called the *mirror direction*. The mirror direction is computed by reversing the component of the direction vector of the light beam that is perpendicular to the surface.

N

Normal

A normal is a vector that is perpendicular to a surface. The normal can be thought of as an arrow that always points out from a surface. When surfaces are shaded by the computer, their normals are used to determine how light reflects off of them. Computer graphics techniques such as bump mapping and Phong shading actually perturb the normals of surfaces during the shading process to make flat surfaces appear bumpy or round.

O

Orthogonal

If two lines or vectors are orthogonal, then they are perpendicular, or set at a right angle to each other. This term is closely linked to orthographic, for instance, which means that the direction of sight is always perpendicular to the film plane. Since the rays never diverge, there is no perspective effect where shapes get smaller as they recede into the distance.

P

Parallel

Parallel means extending in the same direction without ever converging. If two vectors are parallel, then the angle between their directions equals 0. This leads to a test for parallel vectors. If two vectors are parallel, then their dot product equals the product of their lengths. This works because the dot product equals the product of the lengths of two vectors times the cosine of the angle between them and if the angle equals zero, then the cosine equals one. Similarly, if two planes are parallel, then their normals are also parallel, otherwise, the planes must intersect.

Perpendicular

Perpendicular means that two things are at right angles to each other. If two vectors are orthogonal, then you can check this by testing if their dot product equals zero. Since the dot product between two vectors equals the product of the lengths times the cosine of the angle between them, it must equal zero when the angle is ninety degrees because the cosine of ninety degrees is zero. If two planes are orthogonal, then their normal vectors are also orthogonal. If a vector is orthogonal to a plane, then the normal to the plane is parallel to the vector.

Perspective

Perspective is a geometrical effect that occurs when you project a three-dimen-

sional scene onto a two-dimensional surface. The result of perspective is that shapes appear to become smaller as they move farther away from you. Your field of view encompasses all shapes within a certain viewing region called the *viewing frustum*. The viewing frustum is usually shaped like a cone or pyramid with the eye at the apex. Any cross section of the viewing frustum will exactly fill the field of view. Since the cross section of the viewing frustum gets larger and larger farther away from the eye and the actual size of a shape never changes, a shape gets smaller and smaller in relation to the cross section of the viewing frustum as it moves further away from the eye. This means that the projection of the shape gets increasingly small as the shape moves away from the eye.

Phong Shading

Phong shading, also called *pixel shading*, is a technique that was developed by Bui-Tong Phong as an improvement upon Gouraud shading. It is similar to Gouraud shading because it uses a faceted polygonal representation of the model. To achieve more accurate shading computations, instead of smoothing the color across the polygons, Phong shading smooths the normals across the surface and applies the lighting model to each pixel with the interpolated normal to achieve the effect of a smooth surface. The cost of more accurate shading is a much slower rendering algorithm since the shading calculations must be performed at every pixel.

Photorealistic

The term *photorealistic* came into vogue in the 1980s when computer graphics began to be realistic enough to masquerade as photographs of the real world. As the technology improves, the qualifications for being considered photorealistic continue to become more stringent. The most realistic computer graphics require a trained and discerning eye to distinguish them from reality.

Pitch

Pitch, in photography and computer graphics, is the amount that your camera is tilted up or down in the frame of reference of the camera. You can think of pitch as the way that your head rotates when you nod or the way that an airplane rotates when it climbs or dives. The axis of rotation is the horizontal, left to right vector in the frame of reference of the camera.

Pixel

Since computers are digital machines, they cannot manipulate or represent any forms of continuous data but must instead represent everything in a quantized, discrete form. For this reason, all forms of digital manipulation, including television and video, process images by breaking them up into a rectangular array of tiny, colored squares called *pixels*. If you look closely at a computer monitor or a television screen, you will see that the images are not smooth, continuous forms, but are in fact composed of tiny squares. Since a digital computer must represent all data as discrete units instead of continuous ranges of values, the color of the pixels may look continuous but is also quantized into a number of discrete levels.

Polygon

A polygon is a two-dimensional shape that is bounded by edges that are straight lines. A polygon must have

three or more sides. A triangle is a polygon with three sides. Polygons may be any shape that can be formed from straight lines and may be either convex or concave.

Primitive

A primitive, in a computer graphics system, is an atomic, immutable entity that is built into the system and can not be changed. Other things may be user-defined, such as procedures, shapes, and user-defined data types. These are non-primitive entities.

Q

Quadric

A quadric is a mathematical classification for a family of shapes that have similar mathematical properties. One property of quadrics is that they may only intersect a line in two places or less, hence they are also called *second-degree shapes*. The set of quadrics includes the sphere, cylinder, cone, paraboloid, one-sheet hyperboloid, and two-sheet hyperboloid. These shapes are similar because they are all derived from conic sections, which are the types of curves that are produced by slicing a cone in a variety of ways. Mathematically speaking, conic shapes can all be described by the second-degree quadratic equation: $ax^2 + bx + c$.

Quantization

You say that something is quantized if it can only take certain discrete values or states. Since computers are digital machines, absolutely everything that the computer can represent must be quantized. For example, if you draw a spectrum or a smoothly shaded image with a computer, it looks like the color changes smoothly over the entire range. In fact, the image is represented by a fixed set of intensity levels (usually from 0 to 255). Since the intensity levels are very close together, the intensity appears to be continuously changing instead of having a number of discrete, fixed levels.

Quartic

A quartic is a mathematical classification for a family of shapes that have similar mathematical properties. The only well known quartic shape is the torus. Quartic shapes have the property that they may only intersect a line in four places or less, hence they are called *fourth-degree shapes*. Mathematically speaking, these shapes can all be described by the fourth-degree quartic equation: $ax^4 + bx^3 + cx^2 + dx + e$

R

Radiosity

Radiosity is a technique for computing ambient light more accurately by computing the inter-reflections between surfaces. Hypercosm does not implement the radiosity technique. Instead of trying to compute the exact amount of ambient light, a simple constant value is used to estimate it.

Real-Time

The term, *real-time*, is used when a computer can run a process steadily while still responding almost instantly to changes in input. The fuzzy part of this definition is in just how fast 'almost instantly' really is. In most cases, if the computer can respond within between 1/30 of a second and about 1 second, then it is considered to have a real-time response. In some cases, real-time

response is more necessary than in others. A video game must have a quick response or else it has a sluggish feel to it and is no fun to play. An animation, such as an architectural fly-through, does not require such an immediate response, so it may be acceptable to update the picture every second or so and still be considered a real-time application.

Rendering

Rendering is the general process of producing an image. A rendering (as in an architectural rendering) is usually intended to simulate or depict something else, either real or imaginary. A random collection of lines, for example, would not be a rendering because it doesn't depict anything, although it is still an image. In computer graphics, rendering refers specifically to the process of drawing an image to screen.

Roll

Roll is the amount that a camera is tilted around its line of sight. You can think of roll as the way that your head rotates when you tilt it from side to side.

S

Scalar

A scalar is any number with or without a fractional part. In mathematics, these are known as real numbers. Real numbers include the set of integers, or whole numbers; rational numbers, which can always be expressed as a ratio of whole numbers; and transcendental numbers such as π or e, which can not be written as finite expressions of whole numbers.

Shader

A shader is a procedure that is executed to compute the apparent color of a surface under a certain set of lighting conditions. In most systems, the shaders are built in to the program and can not be changed by the user.

Silhouette

The silhouette is the apparent edge of a shape from the point of view of an observer. As the observer moves relative to the shape, the silhouette changes. The silhouette marks the place on the shape where the surface changes from visible to hidden and vice versa.

SMPL

The programming language that was later called SAGE, and is now OMAR.

Specular

The word *specular* is an old word that means *mirror-like*. Many surface types have a specular quality to them, such as metals and plastics. Specular materials characteristically have very smooth outer surfaces that enable them to reflect their surrounding and have very sharp, concentrated highlights. An example of a non-specular material is chalk, which reflects light in a very diffuse way.

T

Texture Mapping

Texture mapping is a technique that is used to modulate the color of a surface. The term is somewhat misleading because texture mapping is typically used to change the color of the surface but doesn't actually impart a texture to the surface, like making the surface look bumpy or ridged. To do this, a similar

technique, called *bump mapping*, is used. In texture mapping, the color is usually modulated by taking the color from a two-dimensional image that is mapped onto the surface in some way. This is sometimes called *image mapping*. Alternatively, the color for the surface can be computed using a mathematical function or an algorithm for computing the color. This is known as *procedural texture mapping*. Image mapping can be implemented in the Hypercosm system by using textured materials. Both forms of texture mapping can be implemented in the Hypercosm system by using shaders.

Transformation

A transformation is a way of modifying the location, orientation, or scaling of a shape. The transformations implemented by the Hypercosm system are known as linear transformations because any straight lines in the shapes remain straight after any transformations are performed on the shape.

To imagine how transformations work, think of a piece of rubber graph paper with a picture on it. To perform linear transformations on the picture, you may stretch, move, or rotate the graph paper in any way so long as the lines in the graph paper remain straight and equally spaced. The distance between horizontal lines need not be equal to the distance between the vertical lines, but all the horizontal lines and vertical lines must be consistently spaced. Note that the lines may even be skewed so that they are no longer perpendicular and still satisfy the criteria.

V

Vertex

A vertex is a point where two edges meet. Primitives that are defined by vertices are triangles, polygons and meshes.

Voxel

The word *voxel* is short for *volume element*. A voxel is a box that encompasses a volume of space. Voxels are used internally in the program to speed up the ray tracing process. Hypercosm displays the voxels when ray tracing is used and the double buffer mode is not enabled.

W

World Coordinates

World coordinates are the coordinates of the picture or scene. All of the shapes in the picture must be defined in terms of this coordinate frame. If all shapes were defined in terms of world coordinates, moving complex shapes around would be difficult because it would be necessary to change the world coordinates of each and every part of the shape individually. This is why aggregate shapes are used. When aggregate shapes are used, the sub-shapes may be defined in the local coordinates of the aggregate shape and when the aggregate shape is moved, all of the sub shapes move together as a group.

Y

Yaw

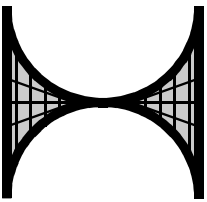
Yaw is the amount that your camera is tilted horizontally in the frame of reference of the camera. You can think

of yaw as the way that your head rotates when you turn your head from side to side. The axis of rotation is the vertical, up-and-down vector in the frame of reference of the camera.

Z

Z-buffer

The z-buffer algorithm is used to eliminate hidden surfaces in the shaded and hidden line rendering modes. It works as a kind of 3D painting algorithm. As you draw the shapes, you check the depth of the shape at each pixel with a depth value of the closest point in current scene at that pixel. A depth value for each pixel on the screen is stored in a big array that is called the z-buffer, or depth-buffer. If the new surface is closer than any surfaces that have been previously drawn, then the color of that surface is written into the image and the depth of the surface at that pixel is written into the z buffer.



Index

3

3D coordinates 10
3d.ores 13

A

absolute transformations 45, 47
    implementation 45
actors 8
aggregate object 119
aliasing 119
all edges 79
ambient light 34, 119
ambient variable 34
anaglyph 69, 119
animation
    high-quality 96
    parameter based 98
    real-time 96
    replaying 116
    saving 116
    speed 96
anims 10, 13, 97
    mouse_controlled_picture 103
    mouse_controlled_shape 103
    mouse-controlled 102
anims.ores 103
antialiasing 87
antialiasing variable 87
args type 8
argument 120
aspect_ratio variable 75

B

background color 75
background variable 75, 92
blob primitive 17, 24
block primitive 17, 23
bump mapping 120

C

C programming language 2
C++ programming language 5
camera
    orientation 62
    placement 12, 61
capping 17
chalk material 50
char_to_key procedure 114
check_keys procedure 102
click type 110
closest-point detection 8
collision detection 8

color variable 48
colors 48
    assigning 48
    background 75
    precedence 51
    predefined 49
comments 13
compiler 2
compiling 2
concave 120
cone primitive 16, 19
constant_color material 50
convex 120
coordinates 10, 120
cross product 121
current transformation state 43
cursor location 106
cursor style 8
cylinder primitive 16, 19

D

dark procedure 49
declarations 13
deferred texturing 53
dimensions transformation 47
direct transformation 41
disk primitive 16, 21
distant light 34, 35
dot product 121
double buffer 97, 121
double_buffer variable 98

E

edge 121
edge_mode 78
edges 78
    all 79
    outline 80
    silhouette 79
edges type 78
edges variable 78
extrusions 9, 122
eye variable 12, 62

F

face shading 84, 122
facet 122
facets variable 81
feature abstraction 86
field of view 63, 122
field_of_view variable 63
finish_loading method 53
fisheye projection 67
flat shading 84, 122
focal point 122
fog 92
fog_factor variable 92
FORTRAN programming language 2
fractals 122
    implementing 57
    in nature 56
frame rates 95

G

geometric primitives 16
get_click procedure 110
get_key procedure 112
get_mouse procedure 106
get_time procedure 116
global illumination 123
Gouraud shading 85, 123

H

h_center variable 74
header statement 12
height variable 73
hidden line rendering mode 77
hierarchical geometry 5, 123
hierarchical modeling 36
hulls 9
hyperboloid1 primitive 16, 20
hyperboloid2 primitive 16, 20
Hypercosm 3D Player 4
Hypercosm Sojourner 2, 7
Hypercosm Studio 2, 7

I

image plane 123
image type 53
include statements 13
infinite plane 17
interpolation 124
interpreter 2, 3

J

jaggies 87, 124
Java 2
Java programming language 5

K

key_down procedure 112
key_to_char procedure 114
keyboard procedures 112
    converting between chars & key-codes 113
keyboard-event queue 113
keycodes 113

L

lathes 9, 124
latitude 26, 124
lattices 9
left_color variable 71
light
    distant 34, 35
    point 34, 35
    spot 34, 35
light procedure 49
lighting 33
    ambient 34
    model 124
    primitives 33, 35
limit transformation 47
line of sight 125
linear 125
linear scanning 83
lines primitive 17, 25
local coordinates 125
local variable 125
longitude 26, 28, 125
lookat variable 62, 70

M

magnify transformation 39
Mandlebrot Set 125
materials 50
    assigning 50
mesh 126
mesh primitive 17, 24, 28
metal material 50
method
    finish_loading 53
    texture status 53
min_feature_size variable 86
mirror direction 126
modeling 15
    hierarchical 36
    procedural 55
mouse
    button 107
    clicks 109
    cursor location 106
    setting cursor style 8
mouse procedures 106
mouse_controlled_picture anim 103
mouse_controlled_shape anim 103
mouse_down procedure 107
mouse-controlled anims 102
mouse-event queue 110
move transformation 39

N

native declarations 9
nesting
    transformations 43
noise function 9
non-planar primitives 17, 24
non-surface primitives 17, 25
normals 126

orientation of 31

O

OMAR 1, 2, 5
OMAR files 8, 9, 12
basic requirements 12
resource files 8, 13
sample source files 9
source files 8, 9
types of 8
ordered scanning 83
ores files 8
orient transformation 41
origin 10, 11
orthogonal 126
orthographic projection 64
outline edges 80

P

painted keyword 53
panoramic projection 68
paraboloid primitive 16, 20
parallel 126
parallelogram primitive 16, 22
parametric procedural models 55
partial surfaces 26
Pascal programming language 2
perpendicular 126
perspective 126
perspective projection 65
Phong shading 85, 127
photorealistic 127
pictures 10, 13
declaring 13
saving 75
pipes 9
pitch 127
pitch variable 63
pixel 127
pixel shading 85, 127
planar primitives 16, 21, 22
plane primitive 16, 21
plastic material 50
point clouds 9
point light 34, 35
pointplot rendering mode 75
points primitive 17, 25
polygon 127
polygon primitive 16, 22
poster keyword 51
precedence
of colors 51
predefined colors 49
primitive 128
blob 17, 24
block 17, 23
cone 16, 19
cylinder 16, 19
disk 16, 21
hyperboloid1 16, 20
hyperboloid2 16, 20
lines 17, 25
mesh 17, 24, 28
paraboloid 16, 20
parallelogram 16, 22
plane 16, 21
points 17, 25
polygon 16, 22
ring 16, 21
shaded polygon 17, 24, 31
shaded triangle 17, 23, 31
sphere 16, 19
torus 17, 23
triangle 16, 22
volume 17, 25, 32
primitive shapes 16
primitives 16
lighting 33, 35
non-planar 17, 24
non-surface 17, 25
planar 16, 21, 22
quadric 16, 19
procedural modeling 55
procedure
char_to_key 114
check_keys 102
finish_loading 53
get_click 110
get_key 112

get_mouse 106
get_time 116
key_down 112
key_to_char 114
mouse_down 107
screen_height 74
screen_width 74
set_cursor 8
program arguments 8
projection 64
fisheye 67
orthographic 64
panoramic 68
perspective 65
projection type 64
projection variable 64
pyramids 9

Q

quadric 128
quadric primitives 19
quadrics 16
quantization 128
quartic 128
queue 110, 113

R

radiosity 128
random function 9
random scanning 83
ray tracing 82
ray tracing, coarse 86
real-time 128
real-time animation 96
reflections 90
reflections variable 90
refractions 90
refractions variable 90
relative transformations 37, 39
render_mode type 75
render_mode variable 75
rendering 15, 73, 129
rendering mode 75
hidden line 77
pointplot 75
shaded 77
shaded line 78
wireframe 76
resource files 8, 13
right_color variable 71
ring primitive 16, 21
roll 129
roll variable 63
rotate transformation 39

S

SAGE 129
saving pictures 75
scalar 129
scale transformation 40
scanning 82
linear 83
ordered 83
random 83
scanning type 82
scanning variable 82
scene 15
screen dimensions 74
screen_height procedure 74
screen_width procedure 74
set_cursor procedure 8
shaded line rendering mode 78
shaded polygon primitive 17, 24, 31
shaded rendering mode 77
shaded triangle primitive 17, 23, 31
shaders 8, 10, 129
shading 83
face 84
pixel 85
vertex 84
shading type 84
shading variable 84
shadows 90
shadows variable 90
shapes 10
primitives 16
silhouette 129
silhouette edges 79

size transformation 47
skew transformation 40
slant transformation 41
SMPL 129
Sojourner 2, 7
sounds 8
specular 129
sphere primitive 16, 19
spot light 34, 35
stack
    transformation 43
status
    texture method 53
stereo glasses 70
stereo pairs 71
stereo variable 70
stereoscopic pictures 69
    changing stereo colors 71
stretch transformation 40
Studio 7
subject
    texture 53
supersampling 88
supersampling variable 88
surfaces
    of revolution 16
    partial 26
sweeps 26

T

tessellation 80
texture mapping 129
texture type 53
    finish_loading method 53
    status method 53
texture_status type 53
textured keyword 53
textures 51
    deferred texturing 53
time procedure 116
torus primitive 17, 23
transformation
    absolute 47
    dimensions 47
    direct 41
    limit 47
    magnify 39
    move 39
    nesting 43
    orient 41
    rotate 39
    scale 40
    size 47
    skew 40
    slant 41
    stretch 40
transformation stack 43
transformation state 43
transformations 130
    absolute 45
    mixing relative and absolute 45
    relative 37, 39
    transforming a series of objects 43
triangle primitive 16, 22
turbulence function 9
type
    click 110
    edges 78
    image 53
    projection 64
    render_mode 75
    scanning 82
    shading 84
    texture 53
    texture_status 53

U

umax parameter 26
umin parameter 26
units 12
url, setting 8

V

v_center variable 74
variable
    antialiasing 87
    aspect_ratio 75
    background 75, 92

color 48
double_buffer 98
edges 78
eye 62
facets 81
field_of_view 63
fog_factor 92
h_center 74
height 73
left_color 71
lookat 62, 70
min_feature_size 86
pitch 63
projection 64
reflections 90
refractions 90
render_mode 75
right_color 71
roll 63
scanning 82
shading 84
shadows 90
stereo 70
supersampling 88
v_center 74
width 73
yaw 63
vertex 130
vertex shading 84, 123
vmax parameter 26
vmin paramter 26
volume primitive 17, 25, 32
voxels 83, 130

W

web utilities 8
width variable 73
window dimensions 73
window position 74
wireframe rendering mode 76
world coordinates 130

X

X-axis 11

Y

yaw 130
yaw variable 63
Y-axis 11

Z

Z-axis 11
Z-buffer 131