An Introduction to
3D Computer Graphics

*Exploring Photo-Realism
with MacRenderMan*

© Malcolm A. Kesson

CONTENTS

Positioning the world relative to the camera
Modelling a coffee mug
The effects of scaling and translation
Reusable geometry
Playing with materials – surface shaders wood, carpet and spatter
Making a composition the wrong way
Making a composition the correct way
Another way of grouping objects
Summary of methods relating to the grouping of objects
A simple polygon model (to be added)

6 An Improved Camera

Overview
Depth of field
Motion blur
Field of view
Matching a VR camera to a real camera

7 Animation

Using FrameUP
Animated texture and displacement maps

8 Basic Lighting

Overview
Defining a light source
Types of light sources – descriptions
Types of light sources – examples
An example script
Reference
Positioning lights in space

9 Advanced lighting – Casting Shadows

Overview
An example script
The shadow algorithm: how it works
An example animation

10   Importing Fragments

Overview
A sample fragment
Importing correctly
Importing incorrectly
Fragments and objects
Restrictions

Appendix A – Overview of MacRenderMan

Appendix B – RenderMan Quick Reference

Appendix C – Shaders Reference

Appendix D – Projects

Separating Shape from Shading
Combining the 'real' and the 'imaginary'
Three Dimensional Icons for a Graphical User Interface

Preface

These notes are intended to explain the basics of theRenderMan system by providing a series of examples of its use in theMacintosh environment. Although a number of exercises and projects have been included they will only be effective when used by those who wish to explore and experiment with the RenderMan system.

I wish to acknowledge the support I received from PIXAR, especially in graciously providing several pre-release versions of their photo-realistic renderer that was being ported to the new RISC based Power-Macintosh computers at the time that I was preparing this booklet for teaching under-graduate students of graphic design the principles of 3D computer graphics.

Malcolm Kesson

April 1994
Wellington
New Zealand

In Progress

The following sections are incomplete,

5 Shaping Up
A sub-section dealing with the way in which RenderMan handles polygons has yet to be added. Several modelling exercises using polygons will also form part of this chapter.

10 Importing Fragments
The contents of this chapter are almost complete but some diagrams have yet to be included and the body copy still requires some editing.

Additional sections that may be added later include a general explanation about "viewing" and shading as well as the following,

11 Advanced Texture Mapping
A chapter dealing with the relationships between cartesian space, texture space and parameter space. Use of texture 's' and 't' parameters to control the texturing of polygons and the use of the command TextureCoordinates to likewise control the texturing of quadric surfaces.

12 Solid Modelling
A chapter dealing with the principles of boolean operations on sets of enclosed objects.

Malcolm Kesson
April 9th 1995

Introduction

Most designers, especially those who are new to computer systems, assume the only way to work with a computer is to use interactive software. Indeed, graphical user interfaces (GUI's, pronounced goo-eez) are taken so much for granted that it may appear strange, if not bizarre, to reject the ease-of-use that such systems offer in favour of an environment based on **text** and **scripting**. What possible advantage could there be in using a keyboard rather than a mouse for graphical input? Why exchange pull-down menu's, floating windows, dialog boxes and icons for an unfamiliar way of making images that requires a large investment of time to master and that emphasises thought, care and perfect attention to detail? The answer to these questions lies principally in the nature of a GUI.

The problem with interactive software is that their interfaces are designed to hide the intricacies of the algorithms and techniques upon which they are based. Infact, just as a conjurer deceptively presents fiction as fact, GUI's organise their illusions around metaphors that routinely entice us to accept the impossible. For example, in illustration software such as Aldus FreeHand or Adobe Illustrator, users interact with elements of their artwork as if they are on **separate layers**. Even operating systems encourage users to perceive windows as being stacked and ordered into layers. Thus, windows can be moved to the front or sent to the 'back'. But the notion that an image on a computer screen can have depth, let alone be comprised of layers, is pure fiction. This course is intended to take you behind the illusions in order to more fully understand the principles of 3D modelling and rendering.

Working in the area of 3D computer graphics without a GUI involves communicating directly with a software package called a renderer. A renderer is somewhat like a laser printer but instead of turning a **2D page description**, normally in a computer language called PostScript, into a printed image, it accepts a **3D scene description** and converts, or renders, it as an image that is either viewed on the computer monitor, or saved as an image file. Because most renderers are embedded within an interactive modeller or animation system the ways in which they can be used are strictly limited by the 'host' software. Infact, the only people who can really 'get at the renderer' are the programmers who wrote the modelling or animation software!

Renderers also form part of software libraries used on high-end graphics workstations. But these require a knowledge of a programing language such as "C", and traditionally, artists and designers have not been given access to such skills. Fortunately, there is a renderer that supports the type of communication that we require–**PRMAN** is part of the innovative **RenderMan** system developed by **PIXAR**. RenderMan is intended to support the production of photo-realistic images based on a 'mini language' called **RIB–R**enderMan **I**nterface **B**ytestream. The intention of RenderMan is to separate modelling from rendering. In formulating their scene description standard, PIXAR established a number of rules by which the characteristics of a virtual world,

and a virtual camera to view that world, can be communicated to a renderer. Because RenderMan organises the way modellers can pass information to renderers, PIXAR refers to their system as an **interface**. Information about a 3D scene is written as text and is stored in a RIB file. Normally these files are produced by an interactive modelling or animation application and are rarely seen by a naive user of a computer system. However, because the details of the RenderMan Interface have been published by PIXAR, anyone with access to a word processor can write or edit a RIB file "by hand" and can gain greater control over the entire image making process. In this course you will use RenderMan to explore the fundamentals of photo-realistic 3D computer graphics.

What is a Script?   Scripts are used to convey information about a production or performance. The samples given below are examples of textural and symbolic scripts. What ever form it takes, a script typically enables an author to pass sufficient information about the structure of a performance so that it can be, in some sense, true or faithfull to the original design. To work effectively, a script must adhere to certain rules that are understood by the author and the performer. For example, it would be a disaster for an actor playing the role of King Henry to speak the lines given in italics, "Aumerle locks the door."

Richard II Act 5.3 – scripting a theatrical performance

> *Enter Bolingbroke, crowned King Henry, with*
> *Harry Percy, and other nobles*
> **AUMERLE** *(rising)*
> Then give me leave that I may turn the key,   35
> That no man enter till my tale be done.
> **KING HENRY**
> Have thy desire.
> *Aumerle locks the door.*
> *The Duke of York knocks at the door and crieth*
> **YORK** *(within)*   My liege, beware! Look to thyself!
> Thou hast a traitor in thy presence there.
> *King Henry draws his sword*
> **KING HENRY** *(to Aumerle)*   Villain, I'll make thee safe.

(NeXT Digital Press 1988)

The scripting you will use in this course is no different to any other type of traditional scripting–you will be the author, PRMAN will be the performer and you will both conform to the rules defined by RenderMan.

The Labanotation System – scripting human movement

forward   side   backward

low   middle   high
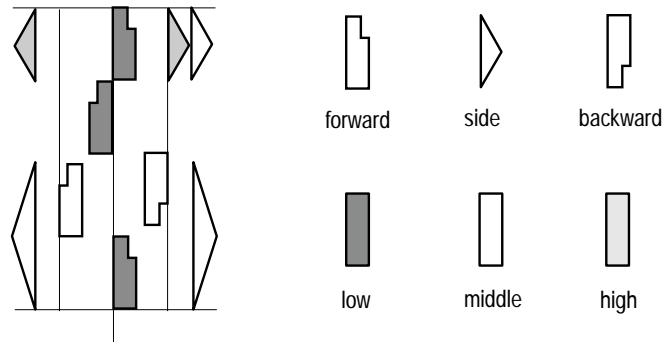
(The New Encyclopaedia Britannica vol 7 page 78)

From Three Pieces for String Quartet (No. 1) by Igor Stravinsky – notation for scripting music
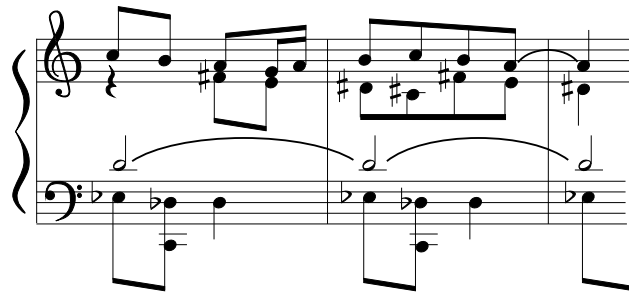
(The New Encyclopaedia Britannica vol 24 page 530)

Why use scripting?

If scripting is so powerful it is appropriate to ask why interactive software is so popular? The answer lies in the breadth and flexibility of modern software design. In a production environment the majority of tasks a designer needs to address can be quickly and adequately tackled with interactive software. But for those who undertake innovative and experimental work, scripting of one kind or another, can offer significant advantages. At one end of the scale, scripting can mean writing an entire software package and at the other end it can mean writing so-called macro's for a spreadsheet. In an educational context, and more especially for a third level degree course, an investigative approach based on scripting means you will learn the general principles of 3D work rather than a single implementation. However, it should be recognized that RIB scripts (files) are NOT normally written **by hand**, but are usually produced by modelling and animation software and these can handle levels of modelling detail that would be impossible for any human to reproduce manually.

Getting Started

Overview
The RIB files in this section are intended to guide you through the basics of working with RenderMan. Each example has been carefully chosen to introduce a broad selection of concepts relating to 3D computer graphics. The explanations accompanying each example are quite brief and are only intended to touch upon the ideas being presented. Don't worry if the material looks terribly confusing. As the course unfolds, the principles underpinning the concepts will be reiterated and illustrated many times over.

When a technical term is used for the first time it is printed in italics. You should make every effort to understand its meaning before continuing with the next section, "Shaping Up – Library Objects and Polygons".

At the conclusion of this section you will be able to

- write, save and send a simple scene description to PRMAN,
- set the basic characteristics of a virtual camera,
- use the basic transformations ie. translation, rotation and scaling,
- distinguish parameters from RIB statements,
- differentiate world space from camera space,
- understand the role of default settings.

Example 1

```
RIB

#ortho disk1.RIB
#using a default camera
WorldBegin
    Disk 1 0.5 360
WorldEnd
```

The purpose of this RIB file is to present a minimal scene to PRMAN and to introduce the basics of interacting with the scripting and rendering environment.

The first two lines show the use of the hash symbol # to indicate these lines are comments and must be ignored by the renderer. Comments can be included anywhere in a RIB file - they are the equivalent of post-it notices.

WorldBegin is a RIB statement and as such must be spelled exactly as shown ie. a single word with two capitalisations. Essentially it notifies RenderMan that objects comprising a scene description–a *virtual world*–are about to be defined.

Disk is a RIB statement that defines, by the three *parameters* (numbers) that follow it, a flat circular disk situated 1 unit along the z axis, 0.5 units in radius and a full 360 degrees in extent. Approximately, half the RIB statements (or commands as they will be referred to) you will use in this course require parameters. In all cases each parameter must be separated by at least one space. They may, however, be spread over several lines of text and have comments at the end of each line, for example,

```
Disk
1      #unit along the z axis
0.5    #units in radius
360    #degrees
```

Finally, WorldEnd indicates the description of the scene, or world, has been completed. This small RIB file is interesting not just for what it describes but also for what it omits. Although it does not specify the characteristics of a virtual camera to view the scene ie. its location and orientation, or the surface colour and material characteristics of the disk, or how it is lit, the renderer is able to produce an image because, in the absence of specific information, it makes several assumptions and uses a number of *default* settings. In particular, RenderMan has provided

- an *orthographic* view looking along the z axis with the camera and the world sharing a common *origin*,
- an image called Untitled measuring 320x240 pixels,
- a matte white surface for the disk that does not require external lighting.

Visualizing example 1

```
#ortho disk1.RIB
#using a default camera
```

two comments about the scene, the first gives the name of the file and the second is a brief note about the scene
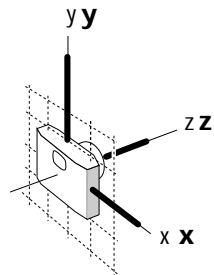
the default camera creates an orthographic view, 320x240 pixels, with a name supplied by RenderApp
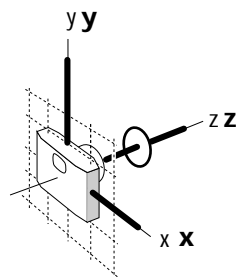
initially the origins of the camera and world coincide

```
WorldBegin
```

begin describing the world

```
Disk   1   0.5   360
```

create a disk situated 1 unit along the z axis, 0.5 units in radius and 360 degrees in circumference

```
WorldEnd
```

description of the world complete

Example 2

RIB

```
#perspective disk.RIB
#setting a perspective view
Projection "perspective" "fov" 40
WorldBegin
    Translate 0 0 3
    Disk 0 0.5 360
WorldEnd
```

The purpose of this file is to show the way in which a virtual camera using *perspective* projection can be set up before the world is defined and also to introduce the use of *translation* to move objects in a scene.

Before the world is defined the statement Projection establishes a perspective view with a *field of vision* of 40 degrees - this is one of several statements that control a virtual camera. Note that two of its three parameters are words. So that RenderMan does not attempt to interpret them as RIB statements or commands, textual parameters are always given in quotes ie. "".

As in the previous example, the scene consists of a single disk but this time the origin of the *coordinate system* has been moved 3 units along the z axis before the disk is defined. The Translate command has three parameters,
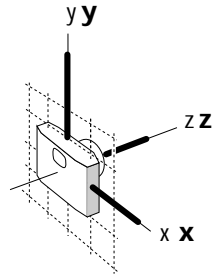
Translate x y z

to move, what may be thought of as a three dimensional cursor around the *world space*.

Visualizing example 2

RIB

Projection "perspective" "fov" 40

set the camera to give a perspective view with a field of vision of 40 degrees, the size and name of the image are supplied by RenderApp
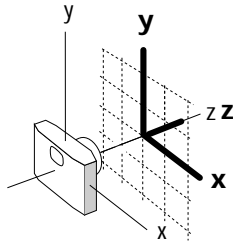
initially the origins of the camera and world coincide

WorldBegin

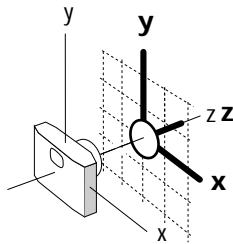begin describing the world

Translate 0 0 3

move the origin 3 units along the z axis

Disk   0   0.5   360

create a disk situated at the origin, 0.5 units in radius and 360 degrees in circumference

WorldEnd

description of the world complete

Example 3

```
#tube.RIB
#rotating an object
Projection "perspective" "fov" 40
WorldBegin
    Translate 0 0 5
    Rotate -120 1 0 0
    Color 1 0 0
    Cylinder 1 -1 1 360
WorldEnd
```

In this scene a cylinder is introduced into a world space that has, for the purpose of better viewing, been rotated and moved away from the camera. The cylinder has also been coloured.

The RIB statement Cylinder, with four parameters,

    Cylinder   radius depth height arc

shows how, like the disk in the previous example, an object from Render-Man's library of primitive shapes can be used in a scene. The cylinder and disk, as well as the other surfaces in the RenderMan library, will be dealt with in detail in the next section.

This file uses another type of *transformation*,

    Rotate   angle x y z

which in this instance turns the coordinate system 120 degrees anti-clockwise around the x axis BEFORE the origin is translated 5 units along the z axis of the world. Although the renderer reads the transformations in the order in which they appear, it postpones applying them until an object is *declared*, at which time it back-tracks and uses the transformations from last to first–like bullets in the magazine of a gun, the last one loaded is the first to be shot!

A cylinder is created within the redefined *world coordinate system*. Since the camera is fixed to the old world origin, the renderer produces an image looking slightly into the top of the cylinder. Using a fixed camera and trying to obtain a particular viewing angle by orientating an object in a scene is only adequate for simple compositions. In the next section the virtual camera is positioned relative to the world - much like a hand held camera in real photography.

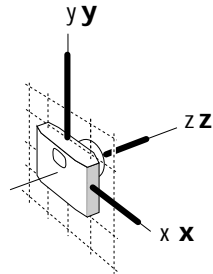Color (note the north American spelling) specifies a hue in terms of three *components*,

    Color   red green blue

A colour is applied to each object until another is declared in the RIB file.

Visualizing example 3

Projection "perspective" "fov" 40

set the camera to give a perspective view with a field of vision of 40 degrees, the size and name of the image are supplied by RenderApp

initially the origins of the camera and world coincide

WorldBegin

begin describing the world

Translate 0 0 5
Rotate -120 1 0 0
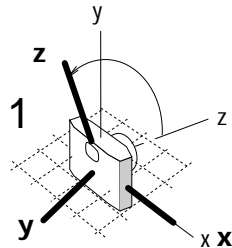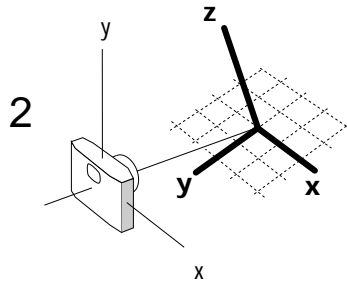
the transformations are applied in reverse order; first an anti-clockwise rotation of 120 degrees around the x axis, followed by a translation of 5 units along the z axis

Cylinder 1 -1 1 360
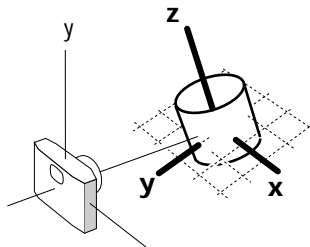
create a cylinder, 1 unit in radius, with a base 1 unit below and a top 1 unit above the origin, 360 degrees in circumference

WorldEnd

description of the world complete

Example 4

RIB

#scaled tube.RIB
#scaling
Display "scaling" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1
WorldBegin
    Scale 0.3 0.3 0.3
    Translate 0 0 5
    Rotate -120 1 0 0
    Cylinder 1 -1 1 360
WorldEnd

This example introduces the idea of *scaling* the world space, and therefore, any objects placed in it. It also illustrates the way in which the characteristics of a virtual camera can be further refined and controlled.

Like the previous example, a cylinder is introduced into a world space that has been rotated and translated for better viewing. However, in this example the world space has also been uniformly reduced to 30% of its original scale.

In this and all future scenes, the RIB statements Display and Format are used to provide additional control over the imagery produced by the virtual camera. Display uses three parameters to specify

- the name of the image,
- where to put the image, and
- what information the image should contain.

Format uses three numeric parameters

   Format   image width   image height   pixel ratio

Although it appears first, Scale only takes effect after the rotation and translation have been applied - remember, transformations are applied in reverse order. The scale statement uses three parameters,

   Scale   x y z

to enlarge or reduce a coordinate system along its x, y and z axes.

Visualizing example 4

Display "scaling" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

set the camera to give a perspective view with a field of vision of 40 degrees, set the size of the image to 200x150 pixels storing rgb information
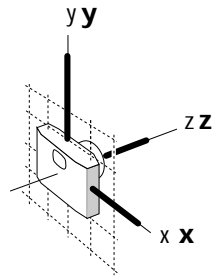
initially the origins of the camera and world coincide

WorldBegin

begin describing the world

3   Scale 0.3 0.3 0.3
2   Translate 0 0 5
1   Rotate -120 1 0 0
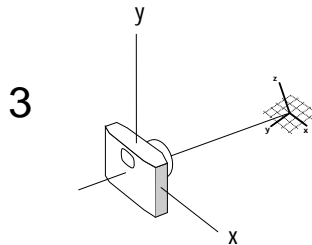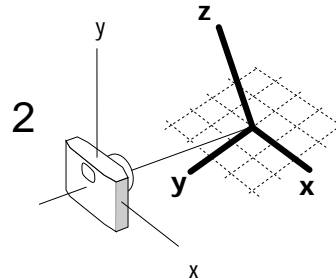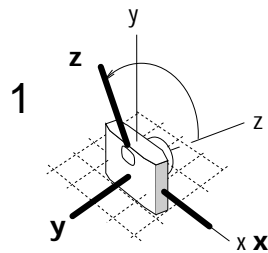
the transformations are applied in reverse order; first an anti-clockwise rotation of 120 degrees around the x axis, then a translation of 5 units along the z axis, followed by the uniform scaling

Cylinder 1 -1 1 360

create a cylinder, 1 unit in radius, with a base 1 unit below and a top 1 unit above the origin, 360 degrees in circumference
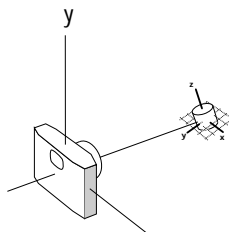
WorldEnd

description of the world complete

Example 5

```
#goblet.RIB
#assembling an object
Display "goblet" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 150 200 1

WorldBegin
    Scale 1 1 1                  #change these to squash and stretch the goblet
    Translate 0 0 5
    Rotate -120 1 0 0

    Color 1.0  0.978  0.34       #gold
    Cylinder 1 0 1.5 360         #container
    Disk 0 1 360                 #base of the container
    Cylinder 0.25 -1.5 0 360     #stem
    Disk -1.5 1 360              #base of the goblet
WorldEnd
```

In this example a number of basic library shapes are assembled into a simple goblet.

Color applies a uniform yellow hue to the entire goblet. Experiment with alternative colour schemes by introducing additional color statements between each part of the goblet. In particular, add another disk positioned a little below the rim of the goblet so that it appears to contain a coloured liquid.

The Scale statement, in effect, does nothing because it applies a uniform scaling factor of one. Change the scaling factor of each parameter to see how the goblet can be individually squashed and stretched in height, width and depth, for example,

    Scale 1 2 1
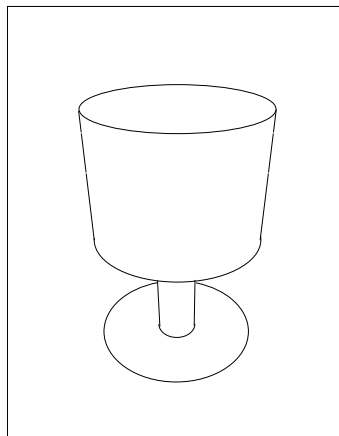
Conclusion

By the time you finish the examples in this section and, no doubt, completed a few modifications of your own, you will have been introduced to many concepts, not only in 3D computer graphics in general, but also in the abstract world of RenderMan. This section concludes with a brief review of the *syntax* of RenderMan and an over-view of the structure of a RIB file.

Syntax   Twelve RIB statements or commands were used in "Getting Started" - by the conclusion of the course you will have dealt with approximately 35 of the entire range of 96 RIB commands. In addition to the hash symbol, the following statements:

- Projection/Display/Format – define a virtual camera,
- WorldBegin/WorldEnd – relate to the concept of a virtual world,
- Translate/Rotate/Scale – are examples of transformations, and finally
- Disk/Cylinder – insert library objects/surfaces into the world.

Incidentally, the words "statement" and "command" are used interchangably. RIB statements form part a *language* recognised by the renderer. By human standards it is an impovished language, but nonetheless, it is in its own right a complete system of communication. Some statements go together in pairs,

```
WorldBegin
WorldEnd
```

and bracket, what are called *blocks* of RIB. Other statements have words and/or numbers, called parameters, associated with them, for example,

```
Translate 0 0 5
Projection "perspective" "fov" 40
```

that, in the majority of cases, provide essential information without which the statement makes no sense.

Structure of a RIB file   At the beginning of a RIB file only a virtual camera exists, and therefore, all statements relate to it and to nothing else, for example,

```
Display…
Projection…
Format…
(anything else that is appropriate…)
```

As soon as the renderer 'reads' WorldBegin, the camera is 'frozen' and all subsequent statements effect the virtual world, for example,

```
WorldBegin
    Objects etc…
WorldEnd
```

and finally, WorldEnd marks the completion of the scene description.

Transformations

There are four basic methods of changing or modifying 3D objects; they can be repositioned, reorientated, resized or distorted in space. These alterations to an object, called *transformations*, are carried out **relative to the origin** of the coordinate system and are known as

- *translation* – moving
- *rotation* – turning
- *scaling* – stretching or squashing
- *skewing* – shearing

The illustrations show the effect of applying the transformations to two cones that are positioned at the origin and a short distance along the x–axis. In each case the z–axis is pointing "into" the page.

Translation

Translate 3 1 0

Rotation

Rotate 45 0 0 1

Scaling

Notice how the cone shifts toward the origin of the coordinate system. The opposite would occur if it were being enlarged.

Scale 0.5 0.5 0.5

Skewing

Skew -30 0 1 0 1 0 0

ie. lean the y–axis 30° toward the x–axis
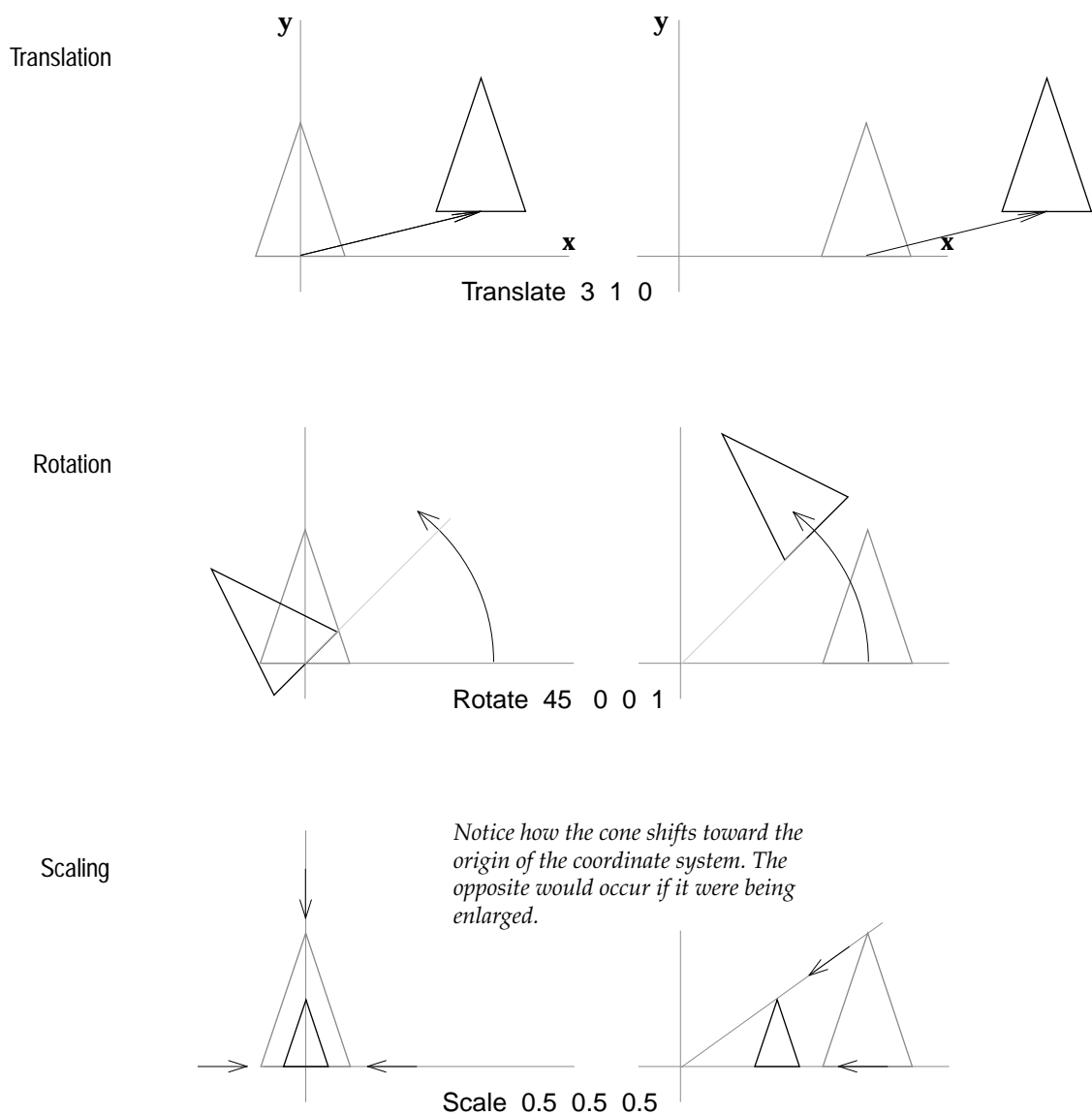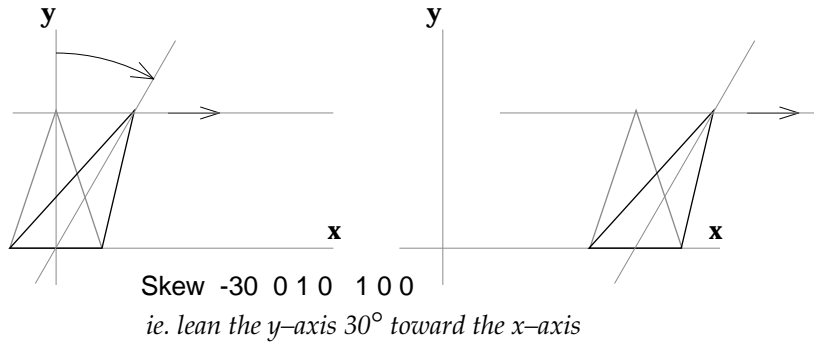
Applying Transformations

Rotating and scaling an object that is NOT positioned at the origin of the coordinate system can give rise to unexpected results. Scaling, for example, has the effect of moving the surface of an object toward or away from the origin depending on whether the object is being reduced or enlarged. If the space into which an object is to be placed is translated, rotated and/or scaled, it normally makes more sense to apply the translation AFTER the rotation and scaling – as shown in the lower drawing.

The following illustrations are based on "Getting Started – example 4", they are intended to show how swapping the order in which Scale and Translate are applied results in the cylinder being placed in two entirely different locations in space. In each example the camera is located at the origin of the coordinate system.

The cylinder appears to be large from the viewpoint of the camera
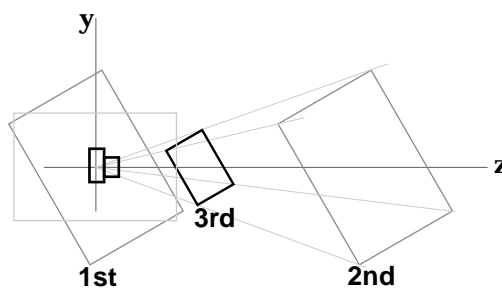

```
Scale  0.3  0.3  0.3    #3rd
Translate  0  0  5      #2nd
Rotate  -120  1  0  0   #1st
Cylinder  1  -1  1  360
```

Here the cylinder appears to be much smaller
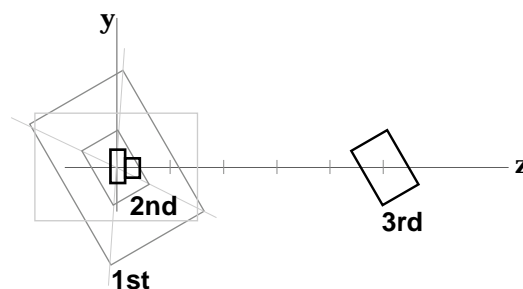

```
Translate  0  0  5      #3rd
Scale  0.3  0.3  0.3    #2nd
Rotate  -120  1  0  0   #1st
Cylinder  1  -1  1  360
```

Applying Sequencies of Transformations

```
RIB

Display "3tubes" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 400 300 1

WorldBegin
    Translate 0 0 7
    Rotate -110 1 0 0
    Rotate 25  0 0 1

    #insert the upright red cylinder
    Scale .5 1 1
    Color  1 0 0
    Cylinder 0.5 -2 2 360

    #insert the green second cylinder
    Rotate  90 1 0 0
    Scale .5 1 1
    Color  0 1 0
    Cylinder 0.5 -2 2 360

    #insert the blue third cylinder
    Rotate  90 0 1 0
    Scale .5 1 1
    Color   0 0 1
    Cylinder 0.5 -2 2 360
WorldEnd
```
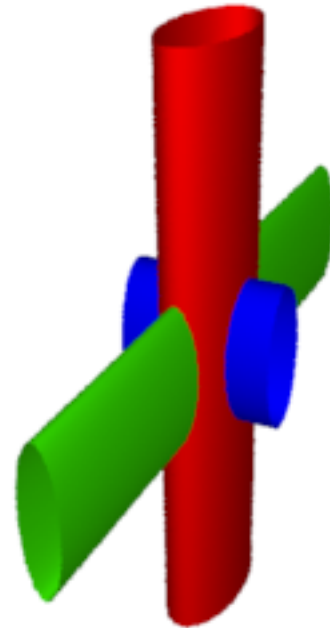
The purpose of this section is to gain a better understanding of the cummu-
lative effect of applying repeated sets of transformations to a modelling space.
The RIB file shown above illustrates one of the unexpected results that can
occur with transformations. In particular notice how the repeated scaling in
the x direction (shown in bold in the RIB script) not only compounds the
overall squashing of each tube but also reduces the **length** of the blue tube.
This is quite an unexpected result. After all why should the length of the last
tube be effected along its z–axis by scalings that have only been applied to the
x–axes? The illustrations on the next two pages provide a step-by-step
explanation.

The camera coordinate system may be considered to lie on the surface of the computer screen – as if it were a large view-finder of a virtual camera. At the beginning of the RIB script the axes shown below (a) mark what is called the *current coordinate system*.

(a)
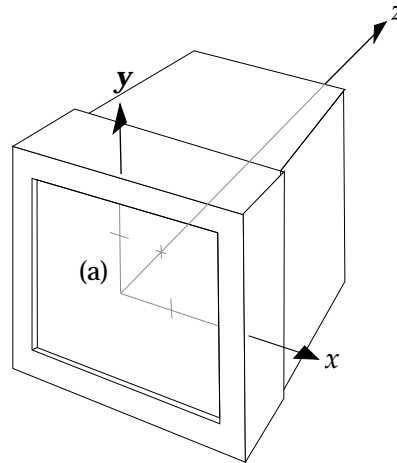
The following RIB statements transform what may be thought of as a **copy** of the current coordinate system **before** the first object is inserted into the world.

```
_3 Translate  0  0  7
_2 Rotate  -110   1 0 0
_1 Scale   0.5  1  1
   Cylinder   0.5  -2  2  360
```

As soon as the cylinder is declared, the transformed copy of the coordinate system (b) becomes the current coordinate system. Subsequent transformations will take place with reference to the new axes (c).
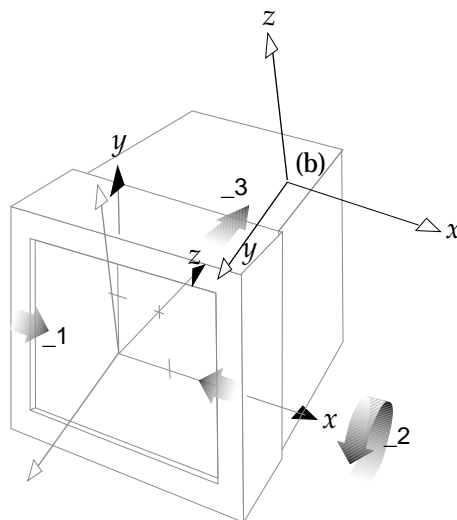
Notice how the x–axis has been scaled by 50%.
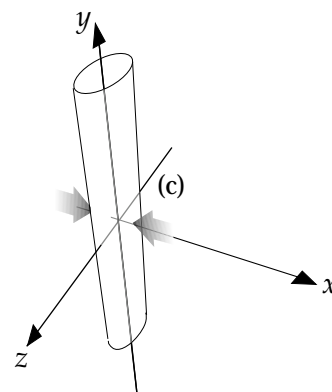
The second set of RIB statements shown below transform what again may be thought of as a copy of the current coordinate system.

```
Rotate  90  1 0 0
Scale  .5  1  1
Cylinder  0.5  -2  2  360
```

Once again as soon as the second cylinder is declared, the copy of the coordinate system (d) becomes the current coordinate system (e). Notice the x–axis has received a further reduction in scale.

The hollow arrows indicate the transformed coordinates before they become the current system.
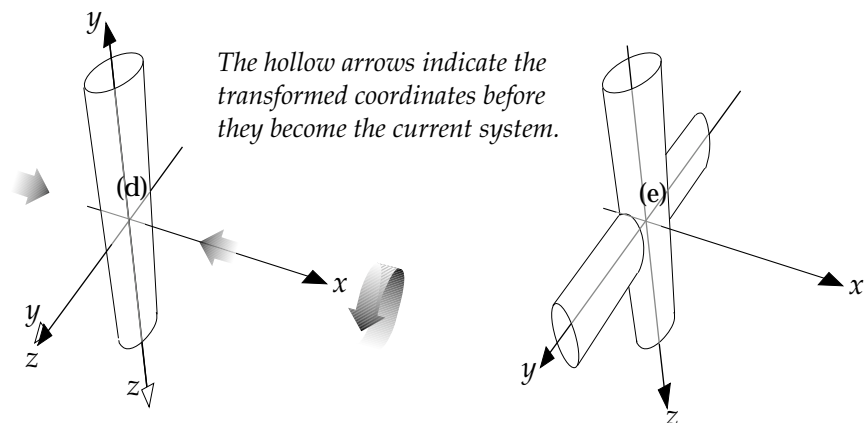
The last set of RIB statements apply the final transformations prior to inserting the third cylinder.

```
Rotate  90  0 1 0
Scale  .5  1  1
Cylinder  0.5  -2  2  360
```

Again the x–axis of the modelling space receives another reduction in scale. However, because of the rotation of the coordinate system around the y–axis, the final cylinder is inserted parallel to a coordinate axis that has been considerably squashed – hence the dramatic reduction in the length of the cylinder.
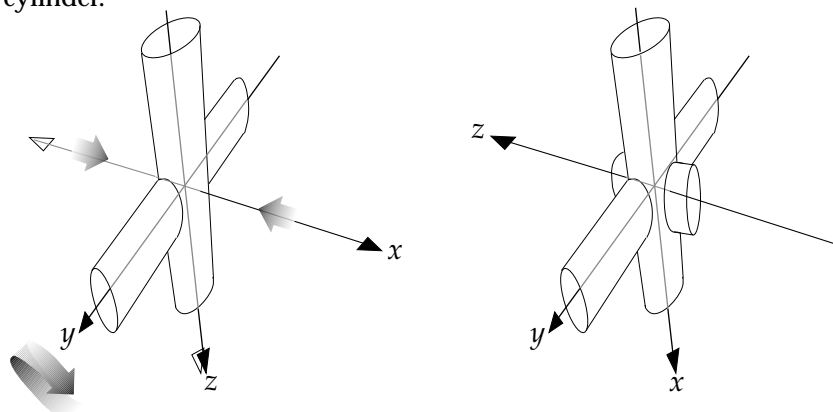
Shading – the basics

The purpose of this section is to introduce the basics of rendering objects photo-realistically. It is intended to be a practical introduction and as such it does not address the theory of rendering.

To produce a realistic image a renderer must be provided with information about the nature of the light sources and the material and geometric attributes of the objects in a synthetic scene. In addition, the renderer also requires information about the virtual camera that is being used to view the scene. For the sake of simplicity the examples in this section use a camera with a standard focal length lens, that provides full coloured low resolution images, eg.

```
Projection "perspective" "fov" 40
Display "untitled" "framebuffer" "rgb"
Format 300 200 1
```

A simple object from the RenderMan library of quadric surfaces, eg.

```
Sphere  1  -1  1  360
```

is used throughout this section so that any undue complications that might arise as a result of modelling a complex object(s) can be avoided.

If computer graphics can be likened to "painting by numbers", then the process of rendering a 3D scene can be thought of as "painting with light". Therefore, a photo-realistic system such as RenderMan, whose purpose is to create imagery that is indistinguishable from 'real' photography, is useful only to the extent that it enables users to make subtle changes to the way light interacts with each part of a scene.

At each stage in the rendering process the way that light is changed as a result of these interactions is called *shading*, and RenderMan provides a mechanism for controlling the outcome of each interaction through the use of what it describes as *shaders*. For example, the interaction of light with the surface of an object is controlled by a *surface shader*; while the characteristics of the light produced by a source of illumination are governed by a *light source shader*. In addition to these two types of shaders this section also introduces a shader that controls the way light reacts with an object whose surface has been displaced to form small bumps and pits – a *displacement shader*.

Although RenderMan provides seven different types of shaders, this section only introduces three of them. The following RIB scripts have been carefully selected to act as templates for your own experiments in the creative use of Surface, Lightsource and Displacement shaders.

Using lights and materials

The following scene consists of a coloured 'plastic' sphere that is illuminated by two lights.

```
RIB

Projection "perspective" "fov" 40
Display "untitled" "framebuffer" "rgb"
Format 300 200 1

Translate  0 0 5
Rotate  -120  1 0 0
Rotate  25  0 0 1

WorldBegin
   LightSource "ambientlight" 1 "intensity" 0.1
   LightSource "distantlight" 2 "intensity" 1.5 "from" [0 0 4] "to" [0 0 0]
   Color  1 0 0
   Surface  "plastic"
   Sphere  1  -1  1  360
WorldEnd
```

Of the two light sources used to illuminate the scene the first provides a very small amount of background, or ambient lighting. The second light is positioned directly 'above' the sphere and is pointing toward the origin of the coordinate system and hence toward the centre of the sphere. Both "ambient-light" and "distantlight" are provided by RenderMan as part of a basic library of four types of light sources, the others are "pointlight" and "spotlight".

The surface of the sphere has been assigned the reflective properties of plastic eg.

```
   Surface  "plastic"
   Sphere  1  -1  1  360
```

(Using a shiny material makes it somewhat easier to see how the high-lights respond to any changes that are made to the position of the light sources.) RenderMan supplies a small number of materials eg. plastic, wood, granite, carpet etc., that can be used with the Surface statement. Each material has its own way of being "tuned" to the individual requirements of a scene. For example, the sphere could have been given the properties of a very matte and non-reflective roughened plastic with the following statement,

```
   Surface  "plastic" "Ks" 0.1 "roughness" 0.5
```

The surface shader "plastic" can use five parameters (the default values for these are given in parentheses), namely,

"Ka"	response to ambient light (1.0),
"Kd"	diffuse reflections (0.5),
"Ks"	specular reflections (0.5),
"roughness"	graininess of the surface (0.1), and finally,
"specularcolor"	the colour of the high-lights ([1 1 1]).

Applying an image to an object

```
RIB

Projection "perspective" "fov" 40
Display "untitled" "framebuffer" "rgb"
Format 300 200 1

Translate  0 0 5
Rotate  -120  1 0 0
Rotate  25  0 0 1

MakeTexture "your picture.tiff" "your picture.tx" "periodic" "periodic" "gaussian" 2 2

WorldBegin
   LightSource "ambientlight" 1 "intensity" 0.1
   LightSource "distantlight" 2 "intensity" 1.5 "from" [0 0 4] "to" [0 0 0]
   Surface  "texmap" "texname" ["your picture.tx"] "maptype" 2
   Sphere  1  -1  1  360
WorldEnd
```

The scene used in this example is very similiar to the first except that a surface shader called "texmap" is used to 'wrap' a 2D image around the sphere – a technique known as *texture mapping*,

   Surface "texmap" "texname" ["your picture.tx"] "maptype" 2

The term texture in the context of 3D computer graphics is a little misleading because it only refers to variations of the colour of a surface, it does not imply anything about its structure or roughness. Since there are a number of ways an image can be wrapped around an object, "texmap" must be told to use "maptype" 2 ie. spherical mapping. Before "texmap" can apply an image to a surface, the image must first be used to generate an intermediate *texture file*. The following statement does the necessay conversion,

   MakeTexture "your picture.tiff" "your picture.tx" "periodic" "periodic"
   "gaussian" 2 2

The main thing to note is that the image to be used as the source for the texture file, which in this instance is called "your picture.tiff", is located in the same folder as the RIB file itself – otherwise the renderer has no way of knowing where to find the appropriate file. Generally, picture files are either created or modified using PhotoShop. It is essential they are stored as RGB files rather than, say, gray scale images. Once an image has been used to create a texture file the MakeTexture statement can be 'commented-out'.

The purpose of the last part of the MakeTexture statement ie. "periodic" "periodic" "gaussian" 2 2, is to allow the texture to be repeatedly tiled over the sphere should that be necessary and to ensure the resulting texture map has a smooth, or anti-aliased, appearance. Like many of the area's touched upon by this section, discussions about the finer details of texture mapping are dealt with elsewhere.

Preparing an image for texture mapping

Step 1 — Scan and/or modify a graphic using PhotoShop, save it as either a TIFF or a PhotoShop 2.5 file. Even if it is a monochrome image be sure to manipulate it in RGB mode within PhotoShop.

Step 2 — Reduce the graphic to a square format by choosing "Image Size…" from the "Image" menu item. The Image Size dialog box will allow the graphic to be resized to a square aspect ratio, say 800 x 800 pixels.

Step 3 — "Select All" using the Select menu item then rotate the graphic 180 degrees with the "Rotate" command under the Image menu.

Step 4 — Use "Save As…" to store the graphic as a TIFF file in the same folder as the RIB file that will use it for texture mapping.

DO NOT MODIFY THE ORIGINAL GRAPHIC – ALWAYS WORK ON A COPY.

Although the Macintosh operating system does not require file extensions, make sure the image file is named with a ".tiff" extension eg. me.tiff. Naming image files and RIB scripts with ".tiff" and ".rib" extensions makes them very easy to identify on the desktop. It is advisable to compress the file using LZW compression.

Step 5 — Within the RIB script convert the graphics file to a texture file with the statement,

```
MakeTexture  "me.tiff" "me.tx" "periodic" "periodic" "gaussian" 2 2
```

Once the RIB script has been used successfully the MakeTexture statement can be commented-out ie.

```
#MakeTexture  "me.tiff" "me.tx" "periodic" "periodic" "gaussian" 2 2
```

This will ensure that subsequent renderings will be completed as quickly as possible. Of course if the original graphics file is altered then a new texture file must be produced, in which case the comment (ie. #) must be removed.

Step 6 — Use the surface shader "texmap" to wrap the texture file around the sphere.

```
Surface "texmap" "texname" ["me.tx"] "maptype" 2
Sphere 1  -1  1  360
```

The "texmap" surface shader allows the same parameters as the "plastic" shader (ie. Ka, Kd, Ks, roughness and specularcolor) to be used to control the way light reflects from the surface of the texture map.

Using an image to displace a surface

```
RIB

Projection "perspective" "fov" 40
Display "untitled" "framebuffer" "rgb"
Format 300 200 1

Translate  0 0 5
Rotate  -120  1 0 0
Rotate  25  0 0 1

MakeTexture "your picture.tiff" "your picture.tx" "periodic" "periodic" "gaussian" 2 2

WorldBegin
    LightSource "ambientlight" 1 "intensity" 0.1
    LightSource "distantlight" 2 "intensity" 1.5 "from" [0 0 4] "to" [0 0 0]
    Displacement  "emboss" "texname" ["your picture.tx"] "Km" 0.03
    Sphere  1  -1  1  360
WorldEnd
```

In this example an image is used to upset or displace the surface of a sphere –
a technique known as *displacement mapping.* Similiar to the surface shader
"texmap", the "emboss" displacement shader requires an image file to be
converted to an intermediate *texture file* ie.

    MakeTexture "your picture.tiff" "your picture.tx" "periodic" "periodic"
    "gaussian" 2 2

Again you must ensure the image to be used as the source for the displace-
ment map is located in the same folder as the RIB file itself. A tiff file to be
used as a source image for displacement mapping can be saved in either
PhotoShop's gray-scale or rgb mode. However, gray scale images give a more
pronounced embossing effect. Once an image has been used to create a texture
file the MakeTexture statement can be 'commented-out' eg.

    #MakeTexture "your picture.tiff" "your picture.tx" "periodic" "periodic"
    "gaussian" 2 2

Unlike a technique called "bump mapping", displacement mapping really
does make changes to the geometry of the surface to which it is applied. It is,
therefore, superior to bump mapping which only makes a surface **appear** to be
bumpy. The "emboss" displacement shader must be given the name of a
texture file that it will use for embossing ie.

    Displacement "emboss" ["your picture.tx"] "Km" 0.03

It should also be given a reasonable value for the parameter "Km" which sets
the magnitude of the embossing. The "emboss" shader responds to the gray
values in the texture file – lighter parts of the image are 'pressed' deeper into
the surface. The default value for "Km" is 0.03.

Avoiding rendering errors and improving performance

Bounding boxes

The renderer uses each objects bounding box to quickly determine where their surfaces are located in a scene. In this way it avoids trying to render "empty space". However, when the surface of an object is shifted as a result of displacement mapping, PRMAN may make severe rendering errors. The renderer literally ignores those parts of the object that have been displaced outside their bounding box. RenderMan provides a mechanism by which the renderer can be warned about such displacements eg.

```
Attribute "bound" "displacement" [0.2]
Displacement "emboss" ["your picture.tx"] "Km" 0.03
 Shere 1  -1  1  360
```

As long as it appears before the name of the object, Attribute can be inserted before or after the Displacement statement. Unfortunately, even though the displacement magnitude is set with the Km factor their is no way of knowing the exact Attribute value to use. In the example shown above it is set to 0.2 but often 0.1 is enough to prevent rendering errors from occuring.

Memory and speed

Rendering operations that involve texture files require more memory than those that don't. To ensure it can operate on computers with modest amounts of memory, RenderMan only sets aside a small amount of memory for working with textures. To improve performance an option may be set to inform the renderer to work with larger "chunks" of texture ie.
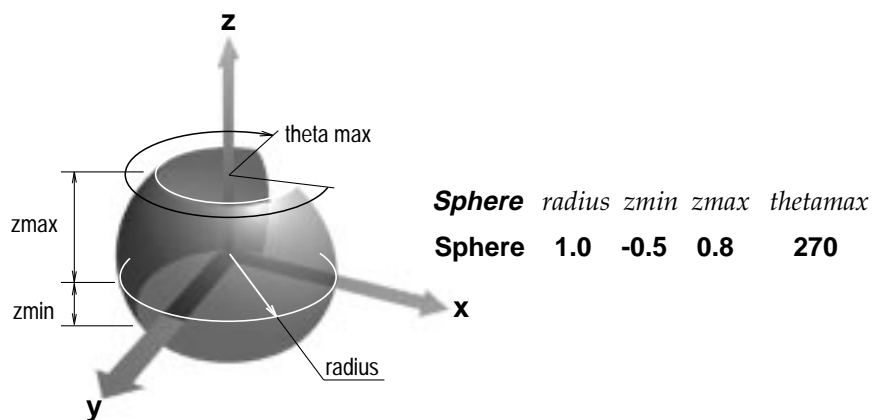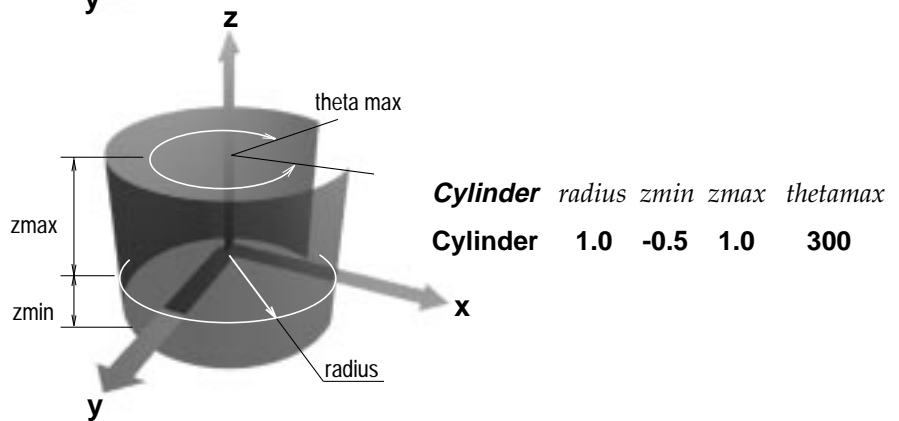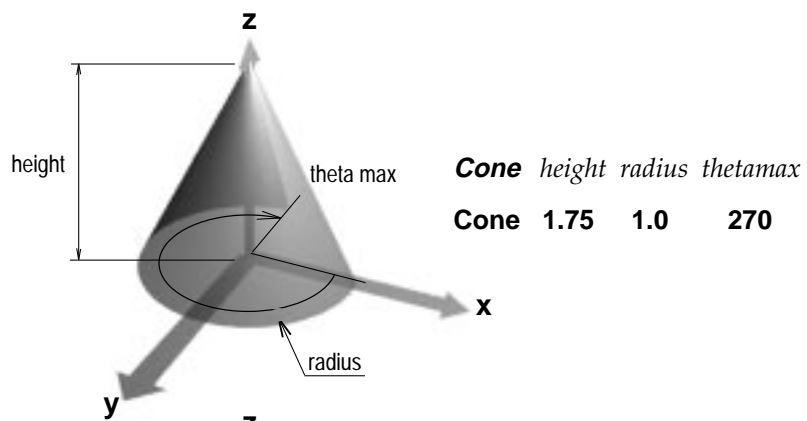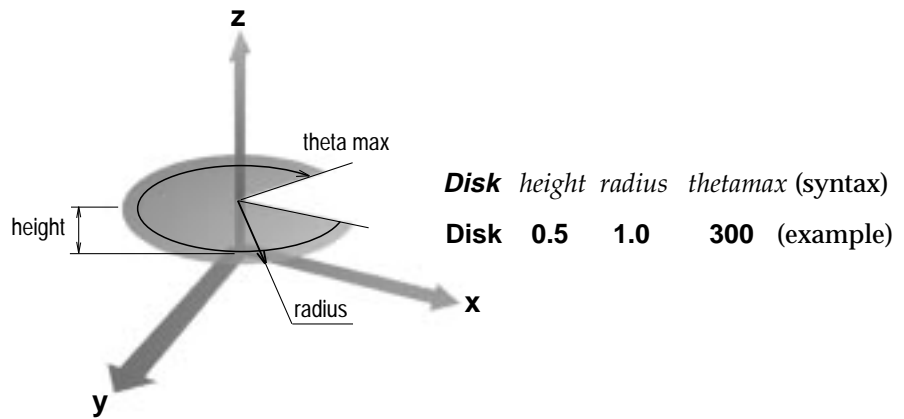
```
Option "limits" "texturememory" [4096]
```

The value "4096" specifies the number of Kbytes (4 MB) to set aside for memory to be used to store information read from a texture file. If you wish to use this option place it at the beginning of the script – options effect the whole scene and must be set before the camera and world are described.
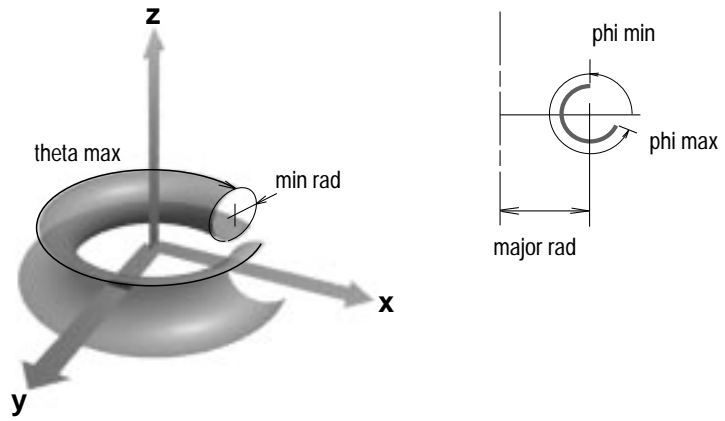
Shaping Up – library objects and polygons

Overview

This section explores the ways that objects are defined in a virtual world. Because our 3D worlds are described by hand written RIB files they will be relatively simple. However, this is not a disadvantage because it will focus attention on imparting as much visual interest through the use of careful shading techniques and sensitive lighting, rather than gratutious complexity obtained all too easily by the use of an interactive modelling system. Before embarking upon the intricacies of lighting and shading some competence must be gained with modelling. This section is designed to provide you with these skills.

"Shaping Up" takes an in-depth look at two types of surfaces commonly used to construct virtual models, namely, *quadrics* and *polygons*. Sophisticated modellers also use surfaces based on curves called splines. If you have used an illustration program such as Adobe Illustrator or Aldus FreeHand you would have employed 2D splines to create curves. However, 3D splines are an advanced topic of study and will not be addressed in this course.

Quadrics

Particular use will be made of the library of shapes, or primitives as they are sometimes called, that are built into RenderMan. These pre-defined surfaces are based upon mathematical expressions called quadratic equations, hence their general name of **quadric surfaces**. There are seven surfaces in the library and they are illustrated on the next two pages. Each quadric has its own set of parameters that allow its form to be accurately specified. The meaning of these parameters and examples of their use are given. In addition to being described by an equation they are also surfaces of revolution. That is, they are formed by spinning a line or curve around a central axis. Most modelling programs offer these primitives because it is easy to assemble them into composite models. Unlike many renderers RenderMan does not approximate quadrics in any way and so renders them with smooth silhouettes.

Polygons

The other type of surface that will be used is a polygon - a flat shape enclosed by straight edges. Traditionally, polygons have been very important in 3D computer graphics because of the ease with which they can be

- internally represented by modellers and renderers,
- assembled into a skin or mesh that approximates a desired form, and
- rendered in a variety of ways to give the illusion of smoothness.

The straight edges of a polygon are defined by a sequence of 3D vertices each of which is specified by three numbers – its x, y and z coordinates. Since even simple polygon meshes can consist of dozens of polygons – each consisting of at least three vertices (ie. triangles), it will only be feasible for us to describe very simple surfaces.

RenderMan's Library of Quadric Surfaces

z

theta max

height

radius

x

y

Disk *height radius thetamax* (syntax)

Disk   0.5   1.0   300 (example)

z

height

theta max

radius

x

y

Cone *height radius thetamax*

Cone 1.75   1.0   270

z

theta max

zmax

zmin

radius

x

y

Cylinder *radius zmin zmax thetamax*

Cylinder   1.0   -0.5   1.0   300

z

theta max

zmax

zmin

radius

x

y

Sphere *radius zmin zmax thetamax*

Sphere   1.0   -0.5   0.8   270

z

theta max

min rad

x

y

phi min

phi max

major rad

Torus *major rad min rad phimin phimax thetamax*

Torus   1.0   0.3   90   320   300

z

radius

theta max

zmax

zmin

x

y
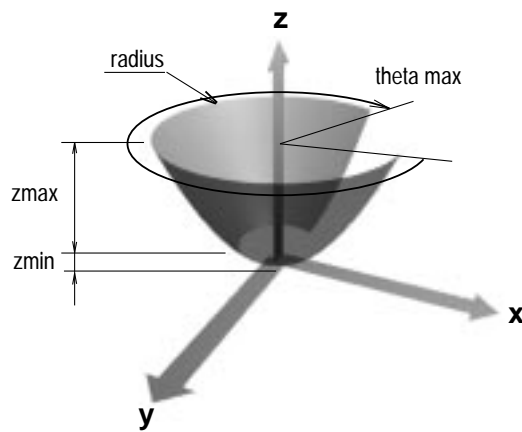
Paraboloid *radius zmin zmax thetamax*

Paraboloid   1.0   0.15   1.2   300

z

point2

x

y
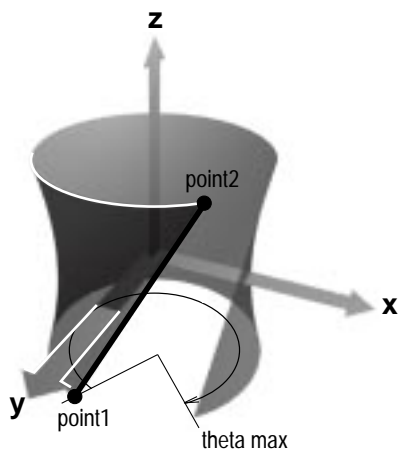
point1

theta max

Hyperboloid   *point1*   *point2*   *thetamax*

Hyperboloid -0.3 1.0 -1.0   0.7 0.7 1.0   300

Example 1 - don't forget the inside!

```
RIB

#better goblet.RIB
#adding an inside surface
Display "goblet" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

WorldBegin
    LightSource "pointlight" 1 "intensity" 40 "from" [4 2 4]
    Translate 0 0 5
    Rotate -120 1 0 0

    Surface "plastic"
    Color 1.0  0.9  0.3          #gold
    Cylinder 1 0 1.5 360         #container
    Disk 0 1 360                 #base of the container
    Cylinder 0.25 -1.5 0 360  #stem
    Disk -1.5 1 360             #base of the goblet

    Translate 0 0 1.5           #move the origin to the top of the goblet
    Sphere 1 -1 0 360           #hemi-spherical inside surface
WorldEnd
```

This example introduces the first of the library shapes – a sphere. It also uses two new RIB statements, LightSource and Surface. 3D computer graphics has developed a rich set of lighting and surface texturing techniques that can dramatically alter the appearance of an object. Although the concepts are dealt with in detail in later sections, light sources and material attributes can still be used effectively, even without elaborate explanations, to add realism to a model.

With the exception of those lines marked in italics, this file is the same as the final example of the previous section. At the end of the scene description the origin is moved to the top of the goblet and the lower half of a sphere is placed within the container by the RIB command,

carpet
cloth
cmarble
constant
finemetal
Matte
metal
paintedplastic
plastic
rmarble
rsmetal
shinymetal
spatter
stone
wood

    Sphere radius zmin zmax   thetamax

A (point) light source is oriented to high-light the curved surfaces of the goblet. The harshness of the lighting can be reduced by inserting this line,

    LightSource "ambientlight" 2 "intensity" 0.2

immediately after the first light source statement. The RIB command Surface, followed by the name of a material in the RenderMan library acts much like Color inthat all subsequent objects acquire the chosen characteristics. Although more will be said about materials and surface textures, you may like to experiment by substituting the parameter "plastic" for any one of those shown in the list given opposite. Later you will be shown how to control the characteristics of each material.

Placing objects in the world

The actions of some of the RIB statements in the first example are illustrated below. In each case the position of the x–y plane is indicated by a grid. The surface being created is shown in the heavier line weight and the parameter(s) responsible for positioning the surface in the z direction are shown in bold.

RIB

Cylinder 1 **0 1.5** 360
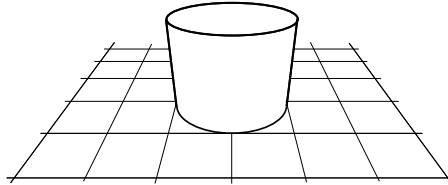
Disk **0** 1 360
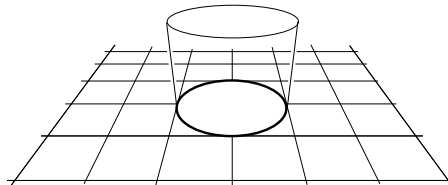
Cylinder 0.25 **-1.5 0** 360
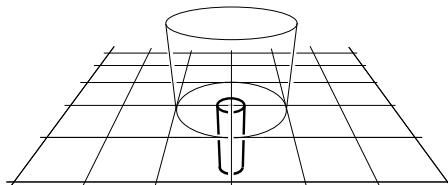
Disk **-1.5** 1 360
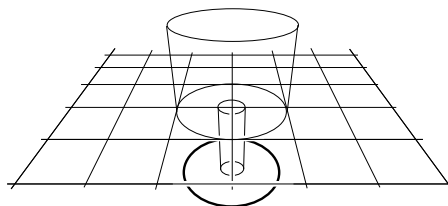
Translate 0 0 **1.5**
Sphere 1 **-1 0** 360
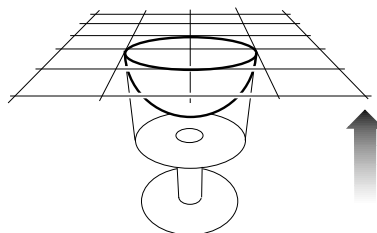
Example 2 - adding a rim and moving the camera

```
RIB

#goblet with rim.RIB
#adding a rim
Display "goblet" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

Translate 0 0 5
Rotate -120 1 0 0

WorldBegin
    LightSource "pointlight" 1 "intensity" 50 "from" [4 2 4]
    LightSource "ambientlight" 2 "intensity" 0.2

    Surface "plastic"
    Color 1.0  0.8  0.3          #gold
    Cylinder 1 0 1.5 360         #container
    Disk 0 1 360                 #base of the container
    Cylinder 0.25 -1.5 0 360  #stem
    Disk -1.5 1 360             #base of the goblet

    Translate 0 0 1.5           #move the origin to the top of the goblet
    Cylinder 0.9 -1.4 0 360
    Disk -1.4 0.9 360
    Torus 0.95 0.05 0 180 360
WorldEnd
```

Although this example demonstrates the use of a torus, its main feature is the way the transformations,

```
    Translate 0 0 5
    Rotate -120 1 0 0
```

that were previously used to rotate and move the goblet **within** the world space are now effecting the **whole** world. Remember, all RIB statements prior to WorldBegin refer to the way the world is oriented with respect to the camera. Because it makes more sense to change the camera–world relationship, as shown by the illustration on the next page, it will no longer be necessary to rotate and move individual objects to obtain a better view of them.
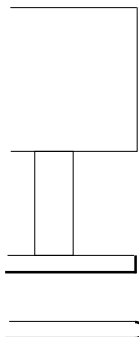
After a cylindrical liner and a flat base have been added to the inside of the goblet a rounded rim is created with the Torus statement,

```
    Torus   major rad  min rad  phimin  phimax  thetamax
```

Try to add another disk to the base of the goblet and provide it with either a

Positioning the world relative to the camera

```
Display "goblet" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1
```

display the graphic in a window titled
"goblet", 200 by 150 pixels in size,
use a camera with a 40 degree field of
vision and include rgb colour data
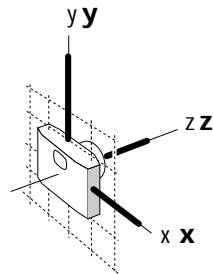
initially the origins of the camera
and the world coincide

```
Translate  0  0  5
Rotate  -120  1  0  0
```

rotate the world -120 degrees
around the x axis
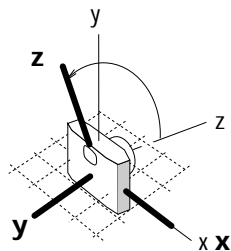
since transformations occur in
reverse order the rotation is followed
by the translation
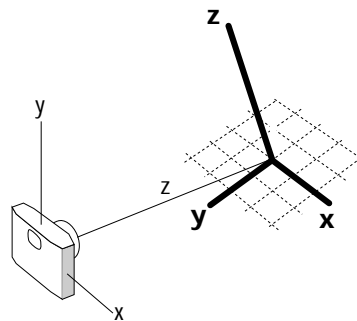
move the world 5 units along the z
axis of the camera

```
WorldBegin
  (assemble the goblet)
```

"freeze" the camera - now only use
the world coordinates
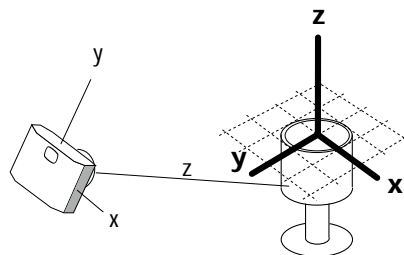
```
WorldEnd
```

scene description complete

Example 3 - anyone for coffee?

```
RIB

#coffee mug.RIB
#modifying the goblet
Display "mug" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

Translate 0 -0.5 5
Rotate -120 1 0 0
Rotate  45 0 0 1

WorldBegin
    LightSource "pointlight" 1 "intensity" 50 "from" [4 4 4]
    LightSource "ambientlight" 2 "intensity" 0.25

    Surface "plastic"
    Color 0 0 1                 #fully saturated blue
    Cylinder 1 0 1.5 360        #mug
    Disk 0 1 360                #base of the mug
    Translate 0 0 1.5           #move the origin to the top
    Cylinder 0.9 -1.4 0 360     #lining of the mug
    Disk -1.4 0.9 360           # bottom of the mug
    Torus 0.95 0.05 0 180 360   #mug rim

    Translate 0 1 -0.75          #move the origin to the back, and lower it half way down the mug
    Rotate 90 0 1 0              #rotate the origin so that the handle will be vertical
    Torus 0.6  0.1  0 360 180    #create a handle
WorldEnd
```

In this example some minor alterations to the scene have changed the goblet into a coffee mug. The statements relating to the stem and base have been removed and those shown in bold have been added or altered. However, the most important point to notice about this file is the way the world is rotated 45 degrees clockwise about the z axis before it is tipped back 120 degrees. In all the previous examples the camera was vertically aligned with the y axis of the world. If you place a comment in front of the camera's second rotation you will immediately see the effect it has on the view. In addition, the mug has been 'centred' by moving the world 0.5 units down the y axis of the camera.

Introduce a Scale statement to widen the handle as shown. The mug does not look tall enough – increase its height to 1.9 units.
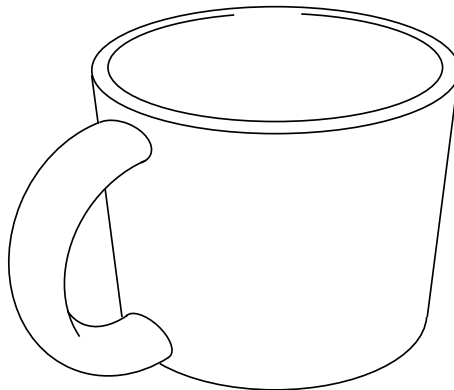
The actions of the RIB statements used in the construction of the coffee mug, example 3, are illustrated below. Unless otherwise indicated, the z axis is pointing up. The surface being created is shown in the heavier line weight and the parameter(s) responsible for positioning the suface in the z direction are shown in bold.

In the last diagram the handle has been widened by applying a scaling factor to the x coordinate. It is left as an exercise for you to determine where in the script the "Scale 2 1 1" command should be inserted.

1   Cylinder 1 **0 1.5** 360
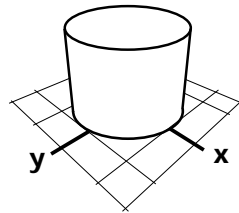
2   Disk **0** 1 360
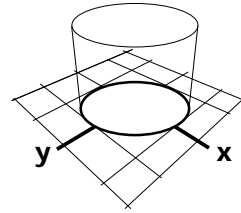
3   Translate 0 0 **1.5**
    Cylinder 0.9 **-1.4 0** 360
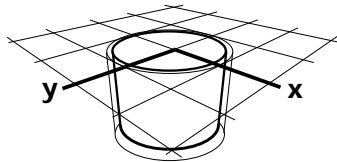
4   Disk **-1.4** 0.9 360
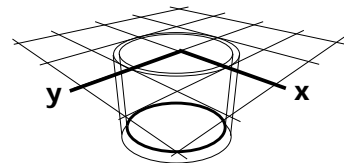
5   Torus 0.95 0.05 0 180 360
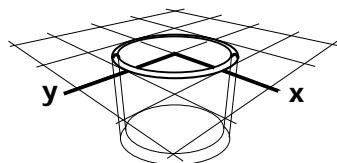
6   Translate 0 1 **-0.75**
    Rotate 90 0 1 0
    Torus 0.6 0.1 0 360 360
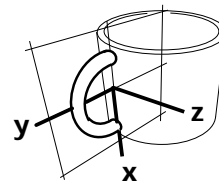
7   Scale 2 1 1
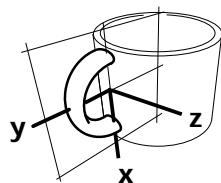
Question: why is the scaling being applied to the x axis when in this diagram it appears as if the z axis requires "stretching"?

Example 4 - the universal saucer

```
RIB

#saucer.RIB
#some tricky scaling
Display "saucer" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

Translate 0 0 7
Rotate -120 1 0 0
Rotate 60 0 0 1

WorldBegin
   LightSource "pointlight" 1 "intensity" 45 "from" [2 -3 4]
   LightSource "ambientlight" 2 "intensity" 0.15

   Surface "plastic"
   Color  0.5  0.5  1      #pale blue

   Translate  0 0 0.5
   Scale  4.4  4.4  1

   Sphere  0.5 -0.5 0 360
   Sphere  0.4 -0.4 0 360
   Torus  0.45 0.05 0 360 360
WorldEnd
```

The purpose of this example is to show the effect of using scaling and trans-
lation, as well as the importance of applying these transformations in the
correct order. Ignoring the transformations for a moment, the combination of
the two spheres and the torus simply produces a hemi-spherical "cup", 0.1
units in thickness, with a rounded rim.

The effect of the Scale statement is to stretch the "cup" into a saucer-like
object, refer to the illustration on the next page. The translation is optional in
that it does not change the form of the saucer, only its position. However, it
makes sense to lift the saucer by an amount equal to its radius so that it 'sits'
on the x–y plane, hence the translation of 0.5 units in the z direction.

The widened rim of the saucer is due to the thickness of the basic "cup" being
exaggerated by the scaling factor–like stretching a sheet of rubber. By adjust-
ing the diameter of the inner sphere, and making the necessary changes to the
parameters of the torus, a wide variety of rims can be created.

The basic "cup" can also be stretched vertically into an object reminiscent of
an egg cup–see the next page. To create this object, x and y have been scaled
by 1, therefore they remain unchanged, while the height in the z direction has
been increased by 200%. To compensate for the scaling, the translation has
been increased from 0.5 to 1 unit ie. 2 x 0.5 = 1.

As an exercise, create an egg by scaling a sphere, assign it an appropriate
colour and position it in the egg cup.

The effects of scaling and translation

no scaling or translation
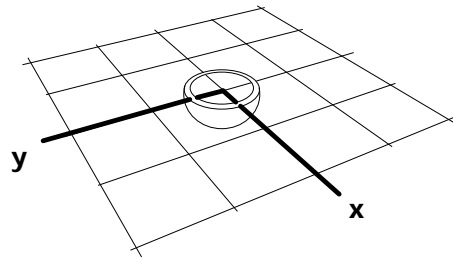
~~Translate 0 0 0.5~~
~~Scale 4.4 4.4 1~~

Sphere 0.5 -0.5 0 360
Sphere 0.4 -0.4 0 360
Torus 0.45 0.05 0 360 360

no translation
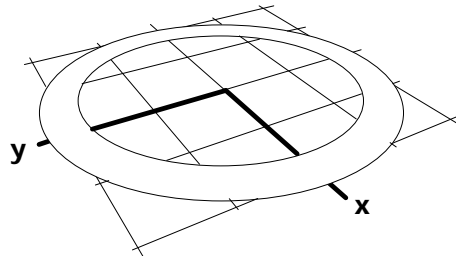
~~Translate 0 0 0.5~~
Scale 4.4 4.4 1

Sphere 0.5 -0.5 0 360
Sphere 0.4 -0.4 0 360
Torus 0.45 0.05 0 360 360

both scaling and translation applied
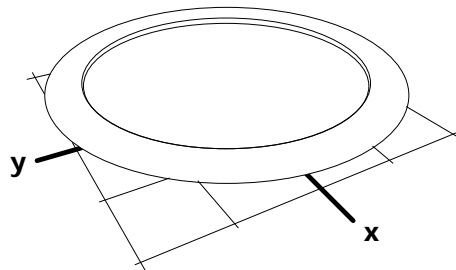
Translate 0 0 0.5
Scale 4.4 4.4 1

Sphere 0.5 -0.5 0 360
Sphere 0.4 -0.4 0 360
Torus 0.45 0.05 0 360 360

now its an egg cup!
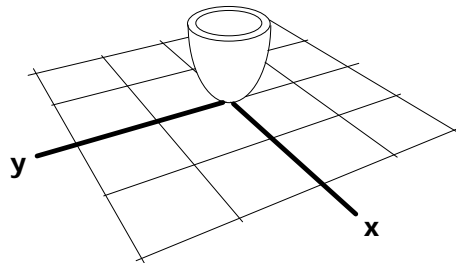
Translate 0 0 1
Scale 1 1 2

Sphere 0.5 -0.5 0 360
Sphere 0.4 -0.4 0 360
Torus 0.45 0.05 0 360 360

For reference the original x–y plane BEFORE the transformations were applied
are shown in each example.

Example 5 virtually "green"–reusable geometry

RIB

```
#eggcup with base.RIB
#copying and pasting with instancing
Display "eggcup" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1

ObjectBegin 1
   Sphere  0.5 -0.5 0 360
   Sphere  0.4 -0.4 0 360
   Torus  0.45 0.05 0 360 360
ObjectEnd

Translate 0 -.5 5
Rotate -120 1 0 0
Rotate 60 0 0 1

WorldBegin
   LightSource "pointlight" 1 "intensity" 25 "from" [2 -3 4]
   LightSource "ambientlight" 2 "intensity" 0.25

   #Egg cup top
   Color .55 .17 .11      #dark brown
   Surface "wood"
   Translate  0  0 1.0
   Scale  1  1  2
   ObjectInstance 1

   #Egg cup base
   Translate 0 0 -0.5
   Scale 1 1 0.25
   Rotate 180 1 0 0
   ObjectInstance 1
WorldEnd
```

The previous example illustrated an important point about 3D models; by making a few minor changes, to scaling for example, their geometry can form the basis of a variety of secondary models. A similiar principle can be applied within a single scene description. This example shows how several surfaces can be collected together into a single *retained* object, and conveniently reused, or *instanced*, many times. In the context of a drawing program this is like making a group, then copying and pasting it repeatedly within an illustration.

The intention to make an object from a collection of surfaces is indicated to the renderer by ObjectBegin/ObjectEnd. The number following ObjectBegin indentifies, or tags the collection for later use by a statement that places the object in the world, the number itself has no other significance,

   ObjectInstance tag

Unfortunately, transformations cannot be used between ObjectBegin and ObjectEnd. Instancing allows the renderer to work more efficiently and also helps to avoid writing tediously long RIB files. Pay particular attention to the next two pages as they explain the transformations used in this scene.

Visualising example 5

The following line drawings show the effect of applying the transformations used in example 5. At each stage, the coordinate system is represented by a one unit grid, subdivided into quarters. To fully understand the action of each group of transformations, remember they are applied,
- in reverse sequence, and
- with reference to the current coordinate system–shown as heavier lines.

The 'new' coordinate system **only** becomes current when an object is created.

RIB (fragment)

```
Translate  0  0  1
Scale  1  1  2
ObjectInstance  1
```

| untransformed | Scale 1 1 2 | Translate 0 0 1 | ObjectInstance 1 |

RIB (fragment)

```
Translate  0   0  -0.5
Scale  1  1  0.25
Rotate  180  1  0  0
ObjectInstance  1
```

| Rotate 180 1 0 0 | Scale 1 1 0.25 | Translate 0 0 -0.5 | ObjectInstance 1 |
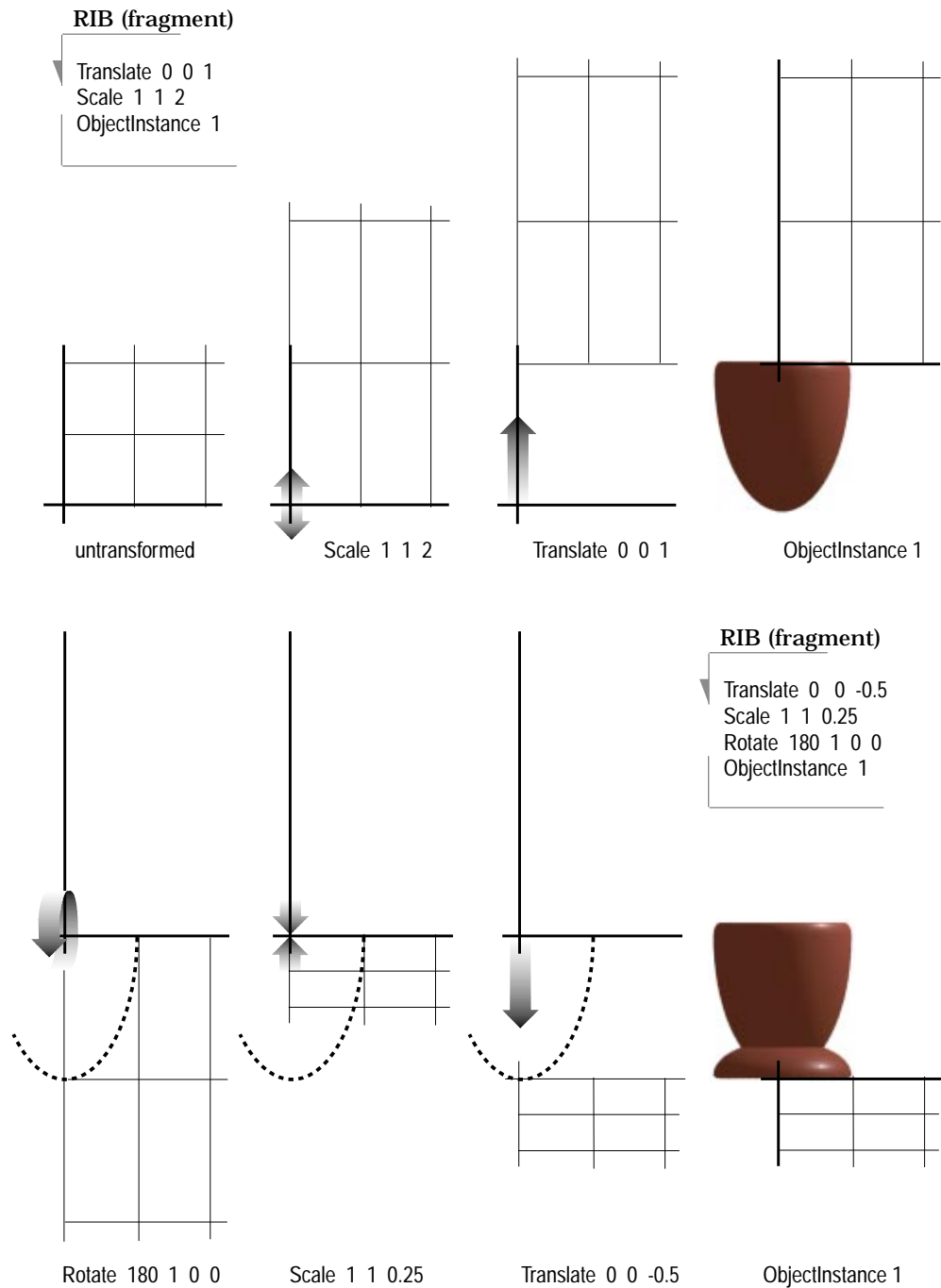
Visualising example 5 – continued

In the last example, the base of the egg cup was positioned and placed in the scene by a scaling followed by a translation.

The same effect can also be achieved by placing the translation before the scaling. However, simply reversing the two statements will not work. As the drawing below shows, the translation must be altered. In general it is better to perform a scaling BEFORE a translation, as shown on the previous page.

RIB (fragment)

```
#Egg cup base
Scale  1  1  0.25
Translate  0  0  -2
Rotate  180  1  0  0
ObjectInstance 1
```

Rotate 180 1 0 0   Translate 0 0 -2   Scale 1 1 0.25   ObjectInstance 1

-1

-2
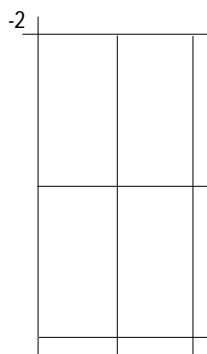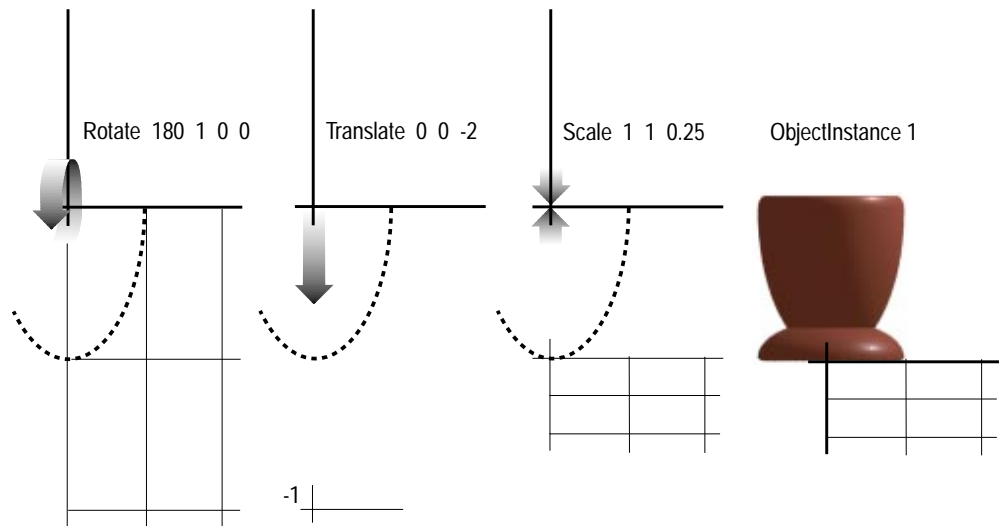
Example 6 – playing with materials

```
#egg and cup.RIB
#playing with materials

Display "eggncup.tiff" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1
ShadingRate 5

ObjectBegin 1
    Sphere  0.5 -0.5 0 360
    Sphere  0.4 -0.4 0 360
    Torus  0.45 0.05 0 360 360
ObjectEnd

Translate 0 -0.7 4
Rotate -120 1 0 0
Rotate 60 0 0 1

WorldBegin
    LightSource "pointlight" 1 "intensity" 20 "from" [2 -3 4]
    LightSource "pointlight" 1 "intensity" 8 "from" [2 3 2]
    LightSource "ambientlight" 2 "intensity" 0.15

    #Table cloth
    Color 0.87  0.71  0.51
    Surface "carpet" "Kd" 1 "nap" 0.5 "scuff" 0.5
    Disk  0  20  360

    #Top
    Color  0.55  0.17  0.11
    Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
    Translate  0  0  1.0
    Scale  1  1  2
    ObjectInstance 1

    #Egg
    Surface "spatter" "basecolor" [0.87  0.66  0.6] "sizes" 3 "spattercolor" [0.55  0.17  0.11]
    "Ks" 0.0  "Kd" 1
    Sphere 0.4 -0.4  0.4 360

    #Base
    Color 0.55  0.17  0.11
    Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
    Translate  0  0  -0.5
    Scale  1  1  0.25
    Rotate  180  1  0  0
    ObjectInstance 1
WorldEnd
```

With the exception of a sphere that has been added to model an egg, this example is essentially the same as the previous scene. Where it is different, however, is not so much in the area of "shape" as "shading". Although the first example in this section experimented with various materials, this example exploits the way most *shaders* assigned with the Surface statement can have their properties 'tuned' by a number of parameters. Each parameter has a default value that can be either accepted, which is exactly what

happened in example 1, or reset as shown. Three *surface shaders* are used in this example to give the effect of a "carpet", "wood" and "spatter". Their full specifications, as documented by PIXAR, appear on the next three pages. Although the details of surface shading will be dealt with in another section, the descriptions of each of the materials used here should give you enough information to undertake your own experiments. Most of the parameters use values that range from 0 to 1; the exceptions, at least for the materials used in this example, are

```
wood "grain"
carpet "nap"
spatter "sizes"
```

Experiment with some or all the parameters in order to appreciate the control that each provides over the appearance of a surface. In the absence of pre-computed images this is a "trial and error" process. Three techniques can be used to speed up rendering

- use a higher ShadingRate, say 20, and optionally use
- ShadingInterpolation "smooth", to reduce the blotchiness of the image,
- comment-out any surfaces that are not currently being adjusted.

ShadingRate is like a quality control adjuster; low values of around 1 or 2 give excellent results while higher values like 20 or more provide "rough and ready" snapshots. A high shading rate simply tells the renderer not to calculate the colour value for every pixel but to *sample* the pixels at whatever rate has been set. The closest comparison to 'real world' photography is choosing a high speed film with a coarse grain emulsion. Unfortunately, high shading rates generate very pixelated images, see opposite. To reduce these artefacts,

```
ShadingInterpolation  "smooth"
```

can be used to tell the renderer to average-out, or interpolate, the pixels between the samples, otherwise it simply uses a constant colour. The statement can be inserted immediately after ShadingRate. Of course you may wish to take advantage of these image "defects" to achieve a particular illustrative effect, in which case resetting ShadingInterpolation is optional. By default its set to "constant", hence the blocks of flat colour.

Bearing in mind a PAL resolution video image constists of 442,368 pixels, the careful use of these statements can have a very significant effect on the speed of rendering.

Surface Shaders

Name	**"wood" "Ka" "Ks" "Kd" "roughness" "specularcolor" "grain" "swirl" "swirlfreq" "c0" "c1" "darkcolor"**
Defaults	"Ka" 1 "Ks" 0.4 "Kd" 0.6 "roughness" 0.2 "grain" 5 "swirl" 0.25 "swirlfreq" 1 "specularcolor" [1 1 1] "darkcolor" [dependent on the surface colour] "c0" [0 0 0] "c1" [0 0 1]

Description
This shader creates a realistic-looking wood. The frequency of the wood grain can be changed with the grain parameter. The relative amount or amplitude of the turbuent swirl in the grain is controlled by the swirl paramater, and swirfreq controls the frequency of this turbulence. Low values of swirl produce more uniform looking wood, while low values of swirlfreq make the wood appear to be more knotty. Obviously these two parameters interact to a large extent. You should be careful not to set swirl too high or swirlfreq too low or the wood will become a jumbled mess.

The wood is simulated by creating a grain that is essentially composed of differently coloured concentric "cylinders" around a central axis defined by the two points c0 and c1. This axis is the z axis by default. Note that the orientation of this axis can be varied either by changing these two parameters or by doing some transformations between the call to the shader and the definition of the geometry. Either one of these approaches may make more intuitive sense in different applications.

The colour of the wood will normally consist of bands of different intensities of the surface colour. This is the most generally useful way of invoking the shader. However, for special appearances this can be changed by changing the darkcolor parameter, which controls the colour of the dark grain of the wood. The different intensity levels are actually levels of mixing between this colour and the surface colour, so setting the surface colour to red and darkcolor to white will produce red wood with white grain and various shades in between.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. roughness and specularcolor control the sharpness and colour of the specular highlight.

Bugs
This shader can have problems with aliasing.

Surface Shaders continued

Name   **"carpet" "Ka" "Kd" "scuff" "nap"**

Defaults   "Ka" 0.1 "Kd" 0.6 "scuff" 1 "nap" 5 "swirl" 1

Description   This shader produces a carpeted surface, complete with scuff-marks. The scuff parameter controls the "amount of scuff", or the relative frequency of intensity variations. Higher values produce more frequent scuffing. nap describes the "shagginess" of the carpet. Higher values make a more coarse-looking carpet.

The carpet shader makes a reasonable stab at anti-aliasing, so the actual grain of the carpet fades away with distance.

There are no specular reflections from real carpet (at least on a macroscopic scale), so the only lighting parameters are Ka and Kd, which have the usual meanings of ambient and diffuse reflective intensities, respectively.

Bugs   The way anti-aliasing is performed can cause linear artifacts in some cases.

Surface Shaders continued

Name   **"spatter" "Ka" "Ks" "Kd" "roughness" "specularcolor" "basecolor"
"spattercolor" "specksize" "sizes"**

Defaults   "Ka" 1 "Ks" 0.7 "Kd" 0.5 "roughness" 0.2
"specularcolor" [1 1 1] "basecolor" [0.1 0.1 0.5] "spattercolor" [1 1 1]
"specksize" 0.01 "sizes" 5

Description   This shader makes objects look like blue camp cookware with white paint
spatters. Actually, both the blue basecolor and the white spattercolor can be
changed if you desire.

The parameter specksize controls the size of the paint specks as you would
expect. However, there are a range of sizes of paint specks controlled by the
parameter sizes. Lower (integer) values produce smaller and more uniform
specks. Higher values produce some larger blotches and specks of many
different sizes.

The parameters Ka, Ks and Kd, have the usual meanings of ambient, specular
and diffuse reflective intensities, respectively. roughness and specularcolor
control the sharpness and colour of the specular highlight.

Bugs   This shader can have problems with aliasing.

Example 7a - making a composition the wrong way!

RIB

```
#egg and cup.RIB
#playing with materials

Display "eggncup.tiff" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1
ShadingRate 5

ObjectBegin 1
    Sphere  0.5 -0.5 0 360
    Sphere  0.4 -0.4 0 360
    Torus  0.45 0.05 0 360 360
ObjectEnd

Translate 0 -0.7 4
Rotate -120 1 0 0
Rotate 60 0 0 1

WorldBegin
    LightSource "pointlight" 1 "intensity" 20 "from" [2 -3 4]
    LightSource "pointlight" 1 "intensity" 8 "from" [2 3 2]
    LightSource "ambientlight" 2 "intensity" 0.15

    Surface "plastic"
    Color  0.5  0.5  1      #pale blue
    Translate  0 0 0.5
    Scale  4.4  4.4  1

    Sphere  0.5 -0.5 0 360
    Sphere  0.4 -0.4 0 360
    Torus  0.45 0.05 0 360 360

    #Top
    Color  0.55  0.17  0.11
    Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
    Translate  0  0  1.0
    Scale  1  1  2
    ObjectInstance 1

    #Egg
    Surface "spatter" "basecolor" [0.87  0.66  0.6] "sizes" 3 "spattercolor" [0.55  0.17  0.11]
    "Ks" 0.0  "Kd" 1
    Sphere 0.4 -0.4  0.4 360

    #Base
    Color  0.55  0.17  0.11
    Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
    Translate  0  0  -0.5
    Scale  1  1  0.25
    Rotate  180  1  0  0
    ObjectInstance 1
WorldEnd
```

In this example the description of the saucer in example 4, shown in bold, has been copied and pasted into the part of the previous RIB file that described the so-called table cloth. However, as the illustration on the next page shows, something very strange has happened to the egg cup and egg.

Because the saucer is created with a scaled coordinate system all the surfaces defined after this transformation are likewise effected, hence the Ostrich egg effect shown on the left, rather than the desired composition shown on the right!
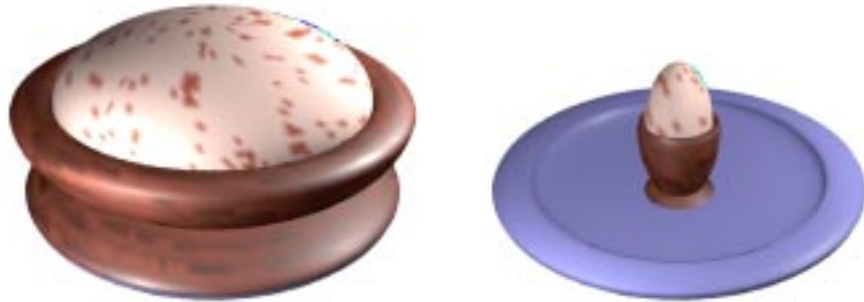
Clearly objects in a scene need to have their individual coordinate systems, or *object space*, and their surface attributes kept, in a sense, private from each other. In a RIB file there are two ways in which this can be achieved. In the following example the two principle objects, the saucer and the egg cup holding an egg, are blocked together between the statements AttributeBegin∕AttributeEnd; these instruct RenderMan to localise (keep private) the geometry **AND** the surface attributes of each object. If only the geometry needs to be kept private, and there are good reasons why this is sometimes necessary, then the TransformBegin∕TransformEnd statements are used instead.

To draw an analogy, if WorldBegin∕WorldEnd define the beginning and end of an entire theatrical play, then the AttributeBegin∕AttributeEnd behave like markers that separate one "scene" from another. Surfaces and polygons fulfill the role of "actors" with each, either separately or in collections, being assigned "costumes" represented by the attributes of Color and Surface.

In the improved RIB script on the following page, ObjectInstance has been used to insert the saucer instead of declaring three separate surfaces. None-theless, the composition still displays a modelling error–notice how the egg cup partially penetrates the saucer. However, because they are grouped together with the AttributeBegin∕AttributeEnd statements, the egg cup and egg can be raised by a single transformation (shown in bold print on the next page). Because of the curvature of the saucer a similiar error accurs if the egg cup is moved toward the rim. But even so, it is sometimes acceptable to allow objects to interpenetrate as long as the error is not too noticable.

The two methods of grouping objects together using AttributeBegin/End and TransformBegin/End are summarised on page 24.

Example 7b - making a composition the correct way

RIB

```
#egg cup and saucer.RIB
#combining objects the correct way!

Display "eggncup.tiff" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 200 150 1
ShadingRate 5

ObjectBegin 1
    Sphere  0.5 -0.5 0 360
    Sphere  0.4 -0.4 0 360
    Torus  0.45 0.05 0 360 360
ObjectEnd

Translate 0 -0.7 4
Rotate -120 1 0 0
Rotate 60 0 0 1

WorldBegin
    LightSource "pointlight" 1 "intensity" 20 "from" [2 -3 4]
    LightSource "pointlight" 1 "intensity" 8 "from" [2 3 2]
    LightSource "ambientlight" 2 "intensity" 0.15

    AttributeBegin    #Saucer
        Surface "plastic"
        Color .5 .5 1
        Translate 0 0 0.5
        Scale 4.4 4.4 1
        ObjectInstance 1
    AttributeEnd

    Translate 0 0 0.1    #raise the egg cup and egg

    AttributeBegin    #Egg cup and egg
        Color  0.55  0.17  0.11
        Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
        Translate  0  0  1.0
        Scale  1  1  2
        ObjectInstance 1

        #Egg
        Surface "spatter" "basecolor" [0.87  0.66  0.6] "sizes" 3 "spattercolor" [0.55  0.17  0.11]
        "Ks" 0.0  "Kd" 1
         Sphere 0.4 -0.4  0.4 360

        #Base
        Color  0.55  0.17  0.11
        Surface "wood" "darkcolor" [0 0 0] "swirl" 0.25 "grain" 15 "swirlfreq" 1.5
        Translate  0  0  -0.5
        Scale  1  1  0.25
        Rotate  180  1  0  0
        ObjectInstance 1
    AttributeEnd
WorldEnd
```

Example 7c - another way of grouping objects

RIB fragment

```
WorldBegin
    #lighting setup the same as before…

    Color 0.87  0.71  0.51
    Surface "carpet" "Kd" 0.8 "nap" 0.8 "scuff" 0.8
    TransformBegin        #mug
        Cylinder 1 0 1.5 360
        Disk 0 1 360
        Translate 0 0 1.5
        Cylinder 0.9 -1.4 0 360
        Disk -1.4 0.9 360
        Torus 0.95 0.05 0 180 360

        #mug handle
        Scale 2 1 1
        Translate 0 1 -0.75
        Rotate 90 0 1 0
        Torus 0.6  0.1  0 360 180
    TransformEnd

    Translate 0 0 -0.15    #lower the saucer

    Color .65 .27 .21
    Surface "wood" "darkcolor" [0 0 0] "swirl" .25 "grain" 15 "swirlfreq" 1.5
    TransformBegin        #Saucer
        Translate 0 0 0.5
        Scale 4.4 4.4 1
        ObjectInstance 1
    TransformEnd
WorldEnd
```

This example is a combination of "coffee mug.RIB" and "saucer.RIB". It shows
how the surfaces and transformations that form an object can be grouped
together using TransformBegin/TransformEnd. Like the groups formed by
AttributeBegin/AttributeEnd in the previous example, these new statements
keep the transformations for each object private, but they do so without
localizing colour and surface attributes. The image on the left shows what
happens if the transformations previously applied to the handle are not kept
"private". The image on the right is the result of the RIB given above.
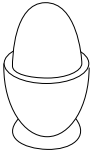
Summary of methods relating to the grouping of objects

AttributeBegin
AttributeEnd
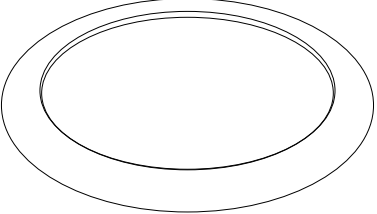
```
WorldBegin
    AttributeBegin
        (shape, transformation and shading information
         relating to the egg and egg cup)



    AttributeEnd


    AttributeBegin
        (shape, transformation and shading information
         relating to the saucer)




    AttributeEnd
WorldEnd
```
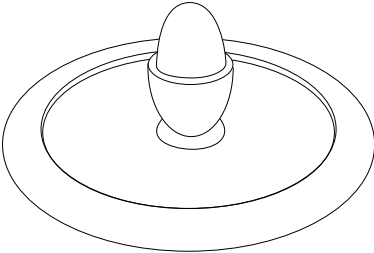
Independent objects each with
their own shape, position AND
shading attributes.

TransformBegin
TransformEnd

```
WorldBegin
    Color 1  1  0   #Yellow
    TransformBegin
        (only shape and transformation
         information relating to a saucer)
    TransformEnd

    TransformBegin
        (only shape and transformation
         information relating to a saucer)
    TransformEnd

    TransformBegin
        (only shape and transformation
         information relating to a saucer)
    TransformEnd

    TransformBegin
        (only shape and transformation
         information relating to a saucer)
    TransformEnd

    TransformBegin
        (only shape and transformation
         information relating to a saucer)
    TransformEnd
WorldEnd
```
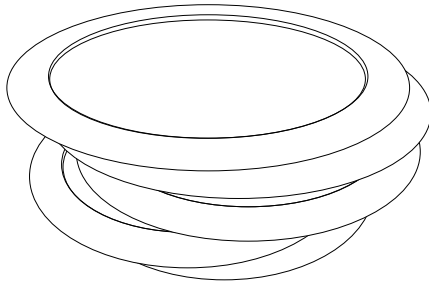
Independent objects each with
their own shape and position, BUT
all sharing a common shading
attribute which in this example
has been set to the colour yellow.

An Improved Camera

In this section the characteristics of a virtual camera will be further refined to include,

- depth of field, and
- motion blur.

In the previous chapters, RIB scripts have typically defined a camera with the following statements,

```
Display "mypicture" "framebuffer" "rgba"
Projection "perspective" "fov" 40
Format 800 800 1
```

Such a camera, however, only approximates the behaviour of a real camera. In particular it acts like a pin-hole camera in that it sharply focuses all parts of an image. Real cameras are strictly limited in their ability to simultaneously focus the images of objects placed at different distances in front of the lens – this is a physical limitation that depends solely on the ratio of the diameter of the aperture of the camera's lens to its focal length. In photography this ratio is called the "f-stop" or "f-number" of a lens. This optical limitation is aesthetically exploited by photographers when the foreground and/or the background of a composition is deliberately defocused.

Real camera's are likewise unable to sharply focus on objects in a scene if either the camera is moving, particularly if it is hand-held, or if the scene, or parts of it, are in motion. To accomodate what, in computer graphics, is called "motion blur" the specification of a synthetic camera must include the idea of an image being formed over a finite period of time ie. as if using a shutter. The 'basic' virtual camera used so far has assumed all motion is frozen by capturing an instantaneous representation of a synthetic scene.

If photo-realism is to be achieved in the digital domain then similiar facilities should be available to 3D computer graphics. By providing mechanisms to introduce, what are in effect, digital versions of "depth of field" and "motion blur", RenderMan greatly improves a designers ability to deliver photo-realistic imagery. Interestingly, some real world photographic nuances are not available in RenderMan, these include effects caused by,

- lens flare,
- optical defects such as astigmatism and chromatic aberration, and
- the mundane effects of dirty or scratched optics!

Because there is a slight difference in the way "field of vision" (fov) is defined in photography compared to RenderMan, this section also provides some examples of converting from one system to the other.

Depth of Field

The RIB statement shown below in bold italics have been inserted to demonstrate the way in which a virtual camera can form an image of a scene as if it had, like a "real" camera, a limited depth of field. The statement that sets the depth of field requires three parameters,

> DepthOfField   f-stop   focal length   focal distance

Since the field of view ("fov") has been previously specified in the RIB script (Projection "perspective" "fov" 45) it is best to use the default value of 1.0 for the focal length; RenderMan will then preserve the relationship between the height of the viewing frame, or the width if that is narrower than the height, and the field of view ie. the image scale remains constant.

Deliberately altering the focal length without making a corresponding change to the fov is like a photographer trying to control the sharpness of an image by changing both the lens AND the format of the camera on which it is mounted – a very bizarre thing to do!

RIB script

```
Display "fuzzy" "framebuffer" "rgba"
Projection "perspective" "fov" 45
Format 400 300 1
DepthOfField 2.0  1.0   5.0

Translate 0 0 5
Rotate -110 1 0 0
Rotate 70 0 0 1

WorldBegin
  Color 0 1 0   #green cone
  Cone 1.5 0.75 360

  Color 1 0 0   #red cone
  Translate -2.0 0.0 0.0
  Cone 1.5 1.0 360

  Color 0 0 1   #blue cone
  Translate 4.0 0.0 0.0
  Cone 1.5 0.25 360
WorldEnd
```
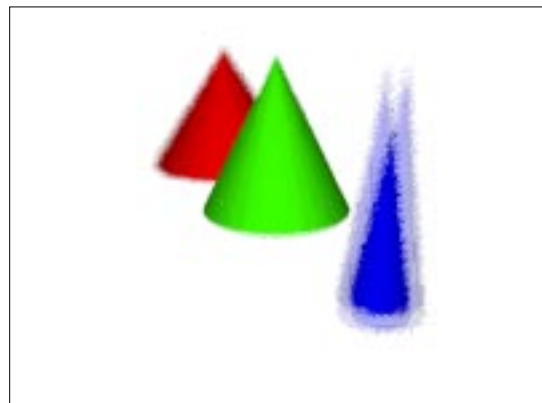

Depth of field set to

f-stop	f2.0
focal length	1.0 units
focal distance	5.0 units

The visual effect of setting other combinations of f-stop and focal distance are shown on the next page.

Depth of Field – continued
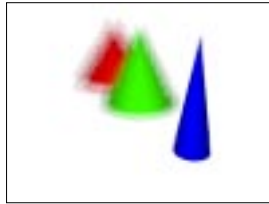
DepthOfField 2.0 1.0 3.0

ie. focus on the near cone using a moderately wide aperature lens.

DepthOfField 1.4 1.0 3.0

ie. focus on the near cone using a very wide aperature lens.
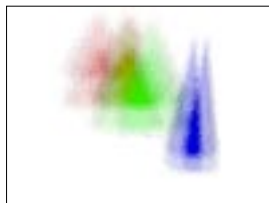
DepthOfField 1.4 1.0 2.0

ie. focus 1 unit infront of the near cone using a very wide aperature lens.
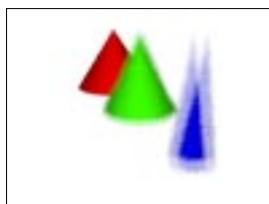
DepthOfField 2.0 1.0 7.0

ie. focus on the far cone using a moderately wide aperature lens.

Motion blur

The role of a virtual camera in photo-realistic rendering is to replicate the most important attributes of a real camera. For example an image recorded onto normal film stock or an electronic imaging device such as a CCD, is formed over a brief period of time as the result of a shutter opening and closing. Any motion occuring during this period of exposure results in the image, or parts of it, being blurred.

Since the location of every object in a virtual world, including the camera, is specified by one or more transformations, RenderMan simulates the effect of motion blur by allowing any transformation, or indeed any "physical" dimension to have several values. The first value is applied when the virtual "shutter" opens and the last one is applied when it closes. The opening and closing times of the shutter are set using the following statement,

    Shutter openTime   closeTime

Normally, openTime will be set to 0 and closeTime will be 1.

Because our RIB files are hand-written, motion blur will be controlled using just two values for any particular dimension or transformation that needs to exhibit motion. Note that attributes such as colour and shading cannot be changed during the opening and closing of the shutter. In the case of a transformation such as a rotation, motion blur would be specified as follows,

```
MotionBegin [0   1]
   Rotation  70  0 0 1    #Transformation at openTime
   Rotation  85  0 0 1    #Transformation at closeTime
MotionEnd
```

In the case of the dimension of an object being varied, such as the height of a cone, the MotionBegin/MotionEnd block might look like this,

```
MotionBegin [0   1]
   Cone 1.5  0.25  360    #Cone pointing up at openTime
   Cone -1.5  0.25 360    #Cone pointing down at closeTime
MotionEnd
```

In the example shown on the next page, motion statements have been used to introduce blur by moving the world 0.05 units from right to left parallel to the x-axis of the camera as the shutter opens and closes from time 0.0 to time 1.0. The units of time, like those of the x, y and z axes, do not refer to any particular system of measurement – they could represent seconds, hours or days.

Motion blur – continued

RIB script

```
Display "camera motion" "framebuffer" "rgba"
Projection "perspective" "fov" 45
Format 400 300 1
Shutter  0   1

MotionBegin [0 1]
  Translate 0 0 5
  Translate -0.05 0 5
MotionEnd
Rotate -110 1 0 0
Rotate 70 0 0 1

WorldBegin
  Color 0 1 0    #green cone
  Cone 1.5 0.75 360

  Color 1 0 0    #red cone
  Translate -4.0 0.0 0.0
  Cone 1.5 1.0 360

  Color 0 0 1    #blue cone
  Translate 7.0 1.0 0.0
  Cone 1.5 0.25 360
WorldEnd
```
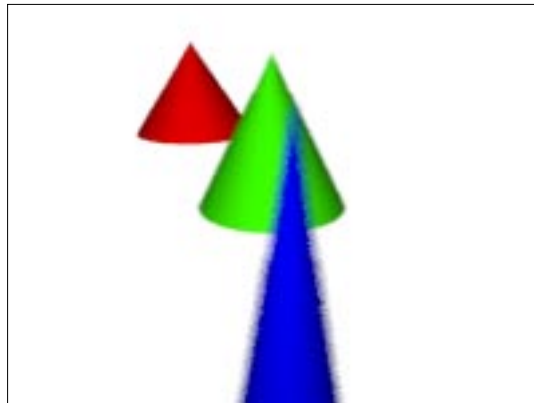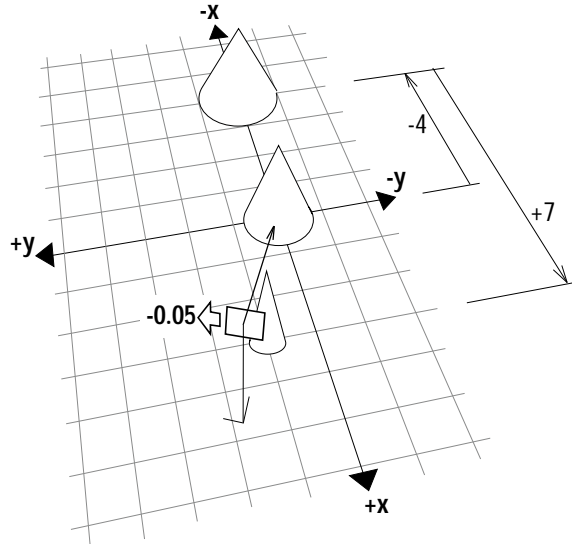

In the second example, motion blur has been achieved by rotating the camera 3° around the z-axis. Since the green cone is situated at the origin, and is itself symmetrical, it remains sharply focused. Question: why in the third example has a **linear** blur been the result of rotating the camera 90° around the z–axis?
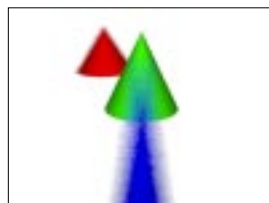

```
Translate 0 0 5
Rotate -110 1 0 0
MotionBegin [0 1]
  Rotate 70 0 0 1
  Rotate 73 0 0 1
MotionEnd
```


notice the linear blurring although a rotation was specified?
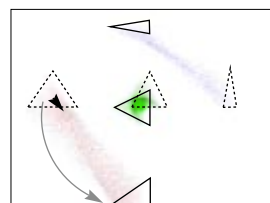
```
Translate 0 0 10
MotionBegin [0 1]
  Rotate 0   0 0 1
  Rotate 90 0 0 1
MotionEnd
Rotate -90 1 0 0
```

Field of View

In photography, the **field of view** (fov) of a lens, somewhat like the cone of vision of an eye, indicates the extent to which objects that are situated away from the direct line of sight (ie. the optical axis) can be focused on the film plane. However, photographers generally relate to the creative possibilites of a particular lens not in terms of its angular field of view – measured in degree's – but rather in terms of more general labels such as "wide-angle", "telephoto" etc. that are themselves based on the **focal length** of the lens.

In real world photography it is convenient to change the field of view either by selecting a lens with a different focal length, or as in the case of a zoom lens, by directly altering its focal length. When setting up a virtual camera with RenderMan it is more convenient to accept the default focal length (1.0 unit) and directly adjust the fov parameter using the Projection statement, for example,

```
Projection "perspective" "fov" 40
```

Unfortunately fov is measured slightly differently by RenderMan compared to normal photography. In photography fov is measured across the diagonal of the film plane, while in RenderMan it is measured across the narrower of the two sides of the image. It is particularly important to take this difference into account when trying to match the view of a virtual camera to that of a real camera, for example, when compositing images from photography and computer graphics.

field of view   field of view

"fov" in photography   "fov" in RenderMan
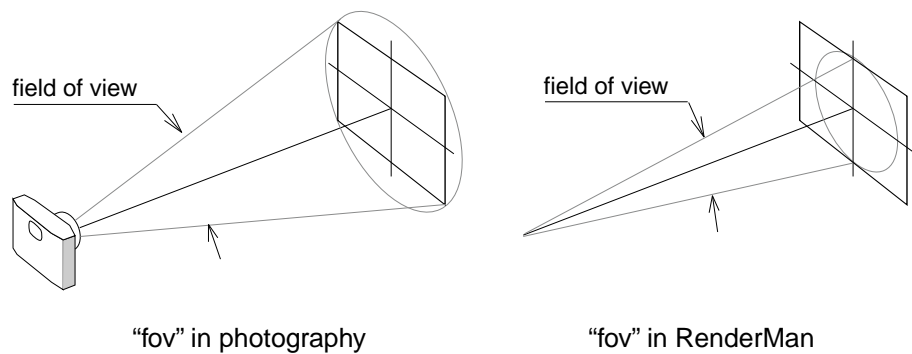
Matching a VR Camera to a Real Camera

This section provides a general over-view of the technique of compositing images generated by 3D computer graphics and photography.

When photographing the scene that will be used as a "background" the following information must be recorded:

location of the camera – its x, y and z coordinates from a known datum,
location of the 'look-at' point – its coordinates relative to the same datum,
focal length of the lens,
f-stop of the lens,
focal distance – not always the same as the 'look-at' point,
general position, orientation and intensity of the light sources, and the position and dimensions of any surfaces on, or against which any virtual 3D objects will be placed.

In the example shown below, the origin of the room ie. the datum, was defined as a point on the floor directly beneath the far corner of the computer bench. Since the camera was positioned to 'look-at' the same point, the corner of the bench is in the centre of the frame of the photograph.
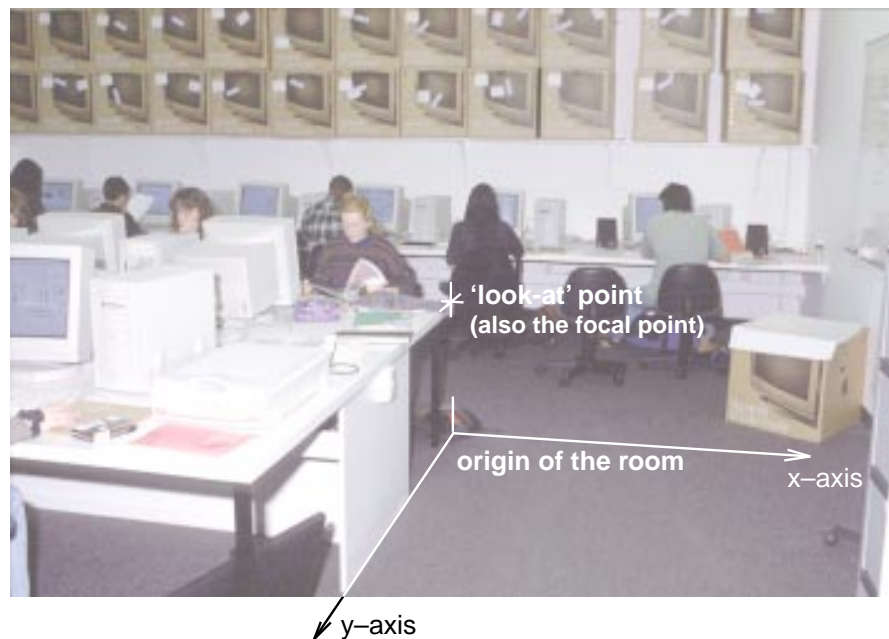
The following measurements were make relative to the origin:

camera position 940mm, 4510mm and 1520mm above the floor,
'look-at' point 730mm directly above the origin,
lens focal length, 35mm wide angle,
lens f-stop, f4,
focal distance; the lens was accurately focused on the corner of the bench,
the scene was principly lit by a flash gun 150mm to the left, and 150mm above the lens,
the computer bench is 2440mm wide.

Matching Camera's – continued

On the basis of those measurements the following RIB script was written in order to **test** the accuracy with which objects, such as cylinders for example, could be placed in a virtual space equivalent to the original room. The locations chosen were,

the origin,
the 'look-at' point and
the leading corner of the computer bench.

It was assumed the x–axis of the world space extended from left to right across the room.

RIB

```
#room.rib
Projection "perspective" "fov" 35
Display "untitled" "framebuffer" "rgba"
Format  1183  787  1

ObjectBegin 1
   Cylinder   4  0   100  360
ObjectEnd

LightSource "distantlight" 2 "intensity" 3
"from" [-150 150  0] "to" [-150 150  100]

#Camera from [940.0  4510.0  1520.0]
#Camera to [0.0  0.0  730.0]
Rotate -99.7   1 0 0
Rotate 191.8   0 0 1
Translate -940.0  -4510.0  -1520.0

WorldBegin
   LightSource "ambientlight" 1 "intensity" 0.1
   Surface "plastic"
   Color 1 0 0
   ObjectInstance 1  #cylinder at the origin

   Color 0 1 0
   Translate 0 0 730
   ObjectInstance 1  #cylinder at the lookat point

   Color 0 0 1
   Translate 0 2440 0
   ObjectInstance 1    #cylinder at the front edge of the bench
WorldEnd
```

In particular note the field of view has been set to 35˚ rather than the 'proper' value of 37.8˚ as shown on the chart on the next page. Because the actual focal length of a camera lens often deviates from its notional focal length it is always necessary to experiment with this factor until the 'test objects' apparently lie in the correct places when the rendered image is overlaid onto the photograph.

Matching Camera's – continued

Comparisons of "fov" used in 35mm photography and RenderMan
(Format must set the viewing frame to a ratio of 3:2)

focal length	photographic fov	RenderMan fov
17	104	70.4
24	84	53.1
28	75	46.4
35	**64**	**37.8**
50	47	26.9
70	34	19.5
135	18	10.1
200	12	6.9

Animation

Overview

The RenderMan interface provides a mechanism by which a sequence of image files can be produced by rendering several scene descriptions contained in a **single** RIB file. Each scene description corresponds to an individual rendered image, and in turn, each of these forms one frame of an animation. A small part of a very simple animation RIB file is shown below.

RIB script

```
Projection "perspective" "fov" 45
Format 400 300 1

Translate 0 0 5
Rotate -110 1 0 0
Rotate 70 0 0 1

FrameBegin 1
   Display "grow.001" "file" "rgba"
   WorldBegin
     Color 0 1 0    #green cone
     Cone 0.25   0.75   360
   WorldEnd
FrameEnd

FrameBegin 2
   Display "grow.002" "file" "rgba"
   WorldBegin
     Color 0 1 0    #green cone
     Cone 0.30   0.75   360
   WorldEnd
FrameEnd
:
:
: (additional FrameBegin / FrameEnd blocks)
:
:
FrameBegin 16
   Display "grow.016" "file" "rgba"
   WorldBegin
     Color 0 1 0    #green cone
     Cone 1.25   0.75   360
   WorldEnd
FrameEnd
```
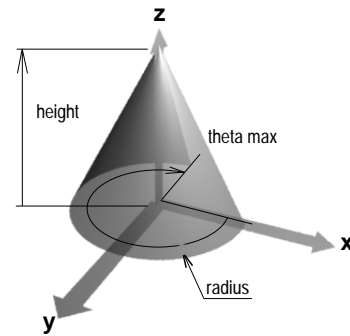
Cone *height radius thetamax*

Notice how each scene description is "bracketed" by the paired statements, FrameBegin/FrameEnd. In this example a cone is increasing in height from 0.25 to 1.25 units. Quite clearly in the context of writing RIB files with a word processor an enormous amount of tedious work ie. copying, pasting and editing, would be required to produce anything but very brief and simple animations. To enable you to experiment relatively conveniently with animation techniques you will be using a utility program called "FrameUP".

Using FrameUP

A common technique used in animation is to define a series of **key-frames** that specify the characteristics of a scene that undergoes change over a period of time. In this way an animator does not have to attend to the specific details of an animation on a frame by frame level. Unfortunately, RenderMan only deals with animations at this low level – it does not have the concept of key-framing built into its scripting language. Normally an animator would use an interactive system that would, for example, allow key-frames to be defined and then subsequently converted into long RIB files that would be processed by a renderer in the normal way. Generally animators do not read these RIB files, much less know what to do with them even if they did!

The utility software, FrameUP, has been designed to allow you to produce animations with RenderMan but at the same time to avoid the chore of producing each and every frame desciption by hand. By processing your animation file BEFORE it is passed to the renderer, FrameUP allows some "extra functionality" to be added to a RIB file. For example, the previous animation could have been written as follows,

```
 ANIMATION script

Projection "perspective" "fov" 45
Format 400 300 1
Display "grow" "file" "rgba"
Translate 0 0 5
Rotate -110 1 0 0
Rotate 70 0 0 1

Tween "from" 1 "to" 2 "frames" 16

KeyFrameBegin 1
   WorldBegin
     Color 0 1 0    #green cone
     Cone 0.25   0.75   360
   WorldEnd
KeyFrameEnd

KeyFrameBegin 2
   WorldBegin
     Color 0 1 0    #green cone
     Cone 1.25   0.75   360
   WorldEnd
KeyFrameEnd
```

The lines in bold printing highlight three of the most important additional statements supported by FrameUP.

Upon reading the Tween statement, FrameUP produces a sequence of inbetween frames, hence its name, that represent the changes that occur in going "from" keyframe 1 "to" keyframe 2 over the duration of 16 "frames".

FrameUP – continued

Any number of key frames can be used so long as each has a unique number or tag ie.

```
KeyFrameBegin 3
  WorldBegin
    Color 0 0 1                #changed to blue
    Cone 1.25  0.75  360      #height the same as keyframe 2
  WorldEnd
KeyFrameEnd
```

Pairs of key frames that will be tweened must have an identical structure – only numeric parameters are allowed to change. For example, it would be illegal to substitute the cone in key frame 2 for a sphere. If FrameUP finds any mistakes of this kind it will warn you and refuse to process the animation file. However, like RenderMan it ignores comments.

In addition, any number of Tween statements can be inserted. For example, the animation script on the previous page could have included the key frame shown above so that the cone increases in height, then changes from green to blue before returning slowly to its original height and colour, ie.

```
Tween "from" 1 "to" 2 "frames" 16
Tween "from" 2 "to" 3 "frames" 16
Tween "from" 3 "to" 1 "frames" 30
```

In each segment of the animation the changes, be they height or colouration, would change linearly ie. at a constant rate. FrameUP also allows changes to occur more gracefully ie.

```
Tween "from" 1 "to" 2 "frames" 16 "smooth"
```

In this case the rate of 'growth' of the cone would be slow at first, becoming more rapid around the 8th frame and then finally it would gradually slow down to the final frame (16). Infact, by tween'ing from key frame 1 to 2 and then back to key frame 1 using "smooth" on both sequences the cone would have a decidedly "bouncy" feel to its motion.

The way in which the tweening occurs can be further controlled by the use of a technique called "easing-in" and "easing-out" – these are sometimes referred to as "fairing-in" and "fairing-out". For example, the cone could be made to abruptly spring to its full height in the first sequence using the following statement,

```
Tween "from" 1 "to" 2 "frames" 16 "easeout" 1.0
```

On the other hand the second sequence could convey the feeling that the apex of the cone is falling as if it were under the influence of gravity ie. slow at first but becoming faster and faster,

```
Tween "from" 2 "to" 1 "frames" 16 "easein" 1.0
```

FrameUP – animated texture and displacement maps

In each case the number following "easein" and "easeout" may be set to any value between 0 and 1. A value of "easeout" 0.3, for example, would indicate that only the final 30% of the inbetweened frames would be eased-out ie. the rate of change would decrease to zero.

Finally, it should be noted that easing-in and easing-out may, to a certain extent, be combined ie.

```
Tween "from" 2 "to" 1 "frames" 100 "easein" 0.2 "easeout" 0.4
```

In this animation, 100 frames in length, only the first 20% and the last 40% of the sequence are effected by easing-in and easing-out. FrameUP (as of version 0.90) will not allow easing-in and easing-out to overlap. In other words if you attempt to set "easein" to 0.6 AND "easeout" to 0.7, FrameUP will reduce the "easeout" period to 0.4, since 60% and 40% add up to 100%. However, this restriction may change in later versions of the software.

FrameUP is able to animate texture and displacement maps in the sense that key frames can specify two different image files for corresponding pairs of MakeTexture statements, for example,

```
Tween "from" 1 "to" 2 "frames" 25

KeyFrameBegin 1
    MakeTexture "myImage.001" "myTex.tx" "periodic" "periodic" "gaussian" 2 2
    WorldBegin
      Surface "texmap" "texname" ["myTex.tx"] "maptype" 2
      Sphere 1  -1  1  360
    WorldEnd
KeyFrameEnd

KeyFrameBegin 2
    MakeTexture "myImage.025" "myTex.tx" "periodic" "periodic" "gaussian" 2 2
    WorldBegin
      Surface "texmap" "texname" ["myTex.tx"] "maptype" 2
      Sphere 1  -1  1  360
    WorldEnd
KeyFrameEnd
```

The important point to notice here is that each of the multiple images files MUST have a 3 digit extension and that the number of frames specified in the Tween statement, 25 in the example shown above, matches the number of images files to be converted to texture files with the MakeTexture statement. Of course ALL of the image files (eg. myImage.001 to myImage.025) must be in the same folder as the RIB file – FrameUP cannot create missing files by morphing images.

Because texture files are generally very large each frame of the animation uses the same name for the texture produced by MakeTexture ie. "myTex.tx", naturally any name can be used for the texture file. In this way, each frame merely over-writes the previously used texture file and thus saves disk space.

Basic Lighting

Overview

Most modelling and animation systems provide at least four types of light sources. In the RenderMan system the basic lights are referred to as "ambientlight", "distantlight", "pointlight" and "spotlight" – note these are spelt as single words. Although these light sources illuminate a scene in different ways they all share the common ability of being able to set their colour and intensity.

Colour is defined in the usual way ie. by the red, green and blue components, whilst intensity is set by a single value normally between 0 and 1. The following, for example, would form part of the specification of a light source,

```
"lightcolor" [1 1 1]
"intensity" 0.2
```

In their standard configuration, none of the basic light sources are able to cast shadows but, with the exception of an ambient light, they each have an extended version that includes this capability. However, shadow casting is an advanced topic and will be dealt with separately.

A scene may be lit by any number of light sources. Objects may share a common light source(s) or be assigned their own individual light source(s) - it just depends where in a RIB file a light source statement(s) appears. The ability to differentially illuminate the objects in a scene is unique to computer graphics – a directly equivalent situation does not exist in "real world" photography.

Defining a Light Source

The renderer creates a light source based upon information passed to it with the RIB command

```
LightSource
```

which is followed by the name of a particular type of light, for example,

```
LightSource "ambientlight" 1
```

and a number that identifies, or tags, the light source. A tag may be any number that is unique to a particular light source.

Lights are adjusted by overriding their default settings. Since in the case of the ambient light created above, specific values for its colour and intensity have not be given it would automatically have the default colour of white and an intensity of 1.0 ie maximum brightness.

Whilst lights can only be created and not destroyed it is possible to switch them OFF and ON via their tag, for example,

```
Illuminate 1 "false"
```

turns the previously created ambient light OFF. When a light is created it is automatically switched ON.

Types of Light Sources - descriptions

Ambient Lighting

An ambient light source uniformly adds colour of a certain intensity to each surface in a scene. It is generally used to increase the level of background illumination in order to soften the effects of other lights. For example, the following RIB statement creates a pale yellow ambient light,

```
LightSource "ambientlight" 1 "intensity" 0.3 "lightcolor" [1 1 0]
```

Distant Lighting

A distant light source acts in much the same way as the sun – it illuminates a scene uniformly in one direction. Objects vary in brightness according to the inclination of their surfaces; their location within the scene has no effect. The values of the "from" and "to" parameters merely specify the direction of the light source and not its 'true' location.

```
LightSource "distantlight" 1 "intensity" 1.0 "from" [2 0 4] "to" [0 0 0]
```

Point Lighting

Like an unshielded electric light bulb a point source radiates light uniformly in all directions. However, unlike the previous light sources, its intensity diminishes over distance – to be precise, brightness varies with the square of the distance. For example, a surface that is three times more distant from a point light source than another surface, only receives one ninth of the light that illuminates the nearer object. The dramatic drop in illumination over distance means that very high values for the intensity parameter are often necessary. Because a point light has a position in space but not a particular direction it does not have a "to" parameter. For example,

```
LightSource "pointlight" 2 "intensity" 30 "lightcolor" [1 1 1] "from" [0 0 9]
```

Spot Lighting

A spot light, on the other hand, has both position and direction – hence it has a "from" and a "to" parameter.

```
LightSource "spotlight" 1 "intensity" 12 "from" [2 0 4] "to" [0 0 0]
```

In addition, spot lights have an extensive range of parameters that control the way they can illuminate a scene. Like a point light their intensity falls off over distance; they have a cone of illumination which by default is set to 60 degrees and they also have control over the light fall-off that occurs at the edge of the cone as well as the distribution of light within the cone itself. As usual with RenderMan, these parameters have default settings and often there is no need to explicitly specify these values.

Types of Light Sources - examples

ambient light
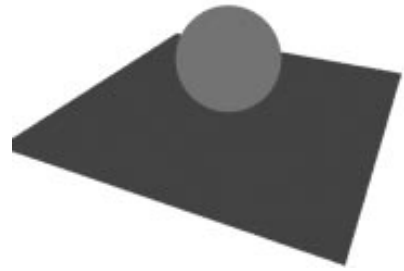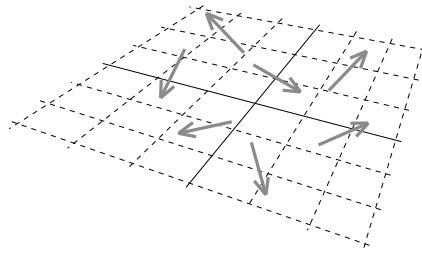
LightSource "ambientlight" 1 "intensity" 0.3

distant light
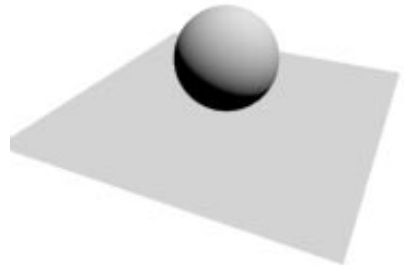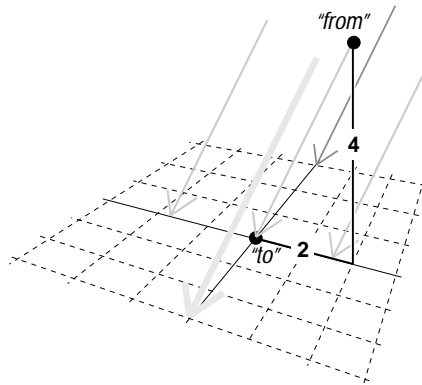
LightSource "distantlight" 1 "intensity" 1.0 "from" [2 0 4] "to" [0 0 0]
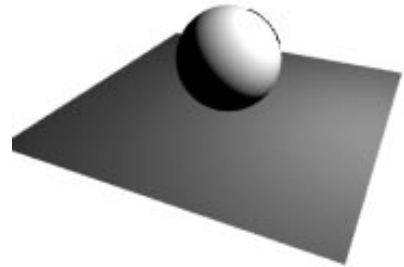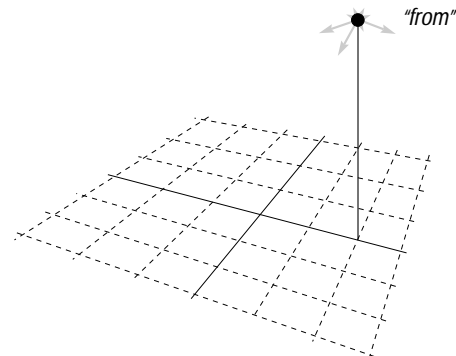
point light

LightSource "pointlight" 1 "intensity" 12.0 "from" [2 0 4]

spot light
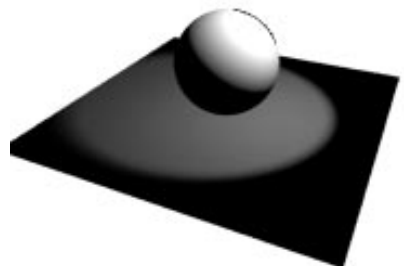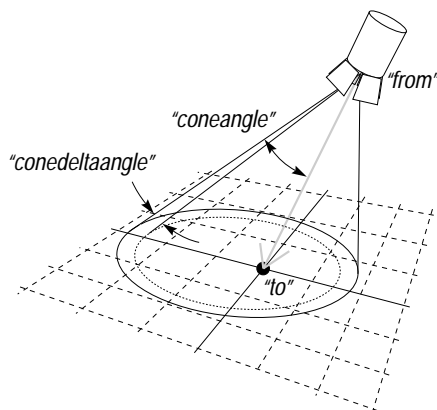
LightSource "spotlight" 1 "intensity" 12.0 "from" [2 0 4] "to" [0 0 0]

In the following RIB file the LightSource statement may be substituted by the examples shown on the previous page. The corners of the polygon have been given arbitary colours so that if you change the camera angle you will be able to orientate yourself more easily – red marks the 'positive' corner.

RIB

```
##RenderMan RIB-Structure 1.0
# Experiments with a single distant source
# 2nd Dec 1993

Display "distantlight" "framebuffer" "rgba"
Format 200 150 1
Projection "perspective" "fov" 40

ObjectBegin 1
    Polygon "P" [-3 3 0  -3 -3 0  3 -3 0   3 3 0] "Cs" [0 1 0   0 0 1   1 1 1   1 0 0]
ObjectEnd

ObjectBegin 2
    Sphere 1 -1 1 360
ObjectEnd

Translate 0 0 9
Rotate -120 1 0 0
Rotate 25 0 0 1

WorldBegin
    LightSource "distantlight" 1 "intensity" 1.0 "from" [2 0 4] "to" [0 0 0]
    Surface "matte"
    ObjectInstance 1
    Translate 0 0 1
    Color 0.8 0.8 0.8
    ObjectInstance 2
WorldEnd
```
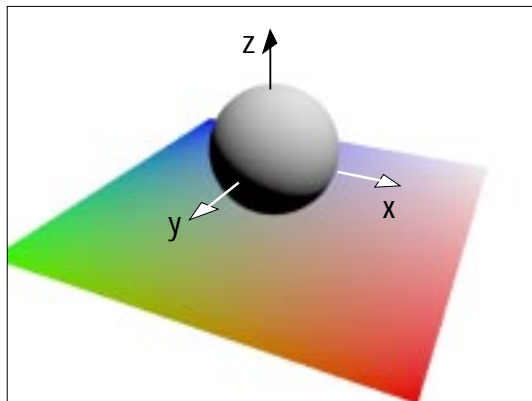
Reference

The optional settings for each of the standard RenderMan light sources are shown in italics and their corresponding default values, as well as their settable range, is also given. Each is followed by an example of the way they could be used in a RIB file.

LightSource "ambientlight"
　"intensity" default 1, range 0 to 1,
　"lightcolor" default [1 1 1], range 0 to 1 for each component.

　LightSource "ambientlight" 1 "intensity" 0.5 "lightcolor" [0.5 0.5 0.5]

LightSource "distantlight"
　"intensity" default 1, range 0 to 1,
　"lightcolor" default [1 1 1], range 0 to 1 for each component,
　"from" default [0 0 0] , unlimited range from positive to negative,
　"to" default [0 0 1], unlimited range from positive to negative.

　LightSource "distantlight" 5 "intensity" 0.5 "lightcolor" [0.5 0.5 0.5] "from" [2 0 4]
　"to" [0 0 0]

LightSource "pointlight"
　"intensity" default 1, range 0 to an unlimited upper value,
　"lightcolor" default [1 1 1], range 0 to 1 for each component,
　"from" default [0 0 0] , unlimited range from positive to negative.

　LightSource "pointlight" 2 "intensity" 25 "lightcolor" [0.2 0.5 1.0] "from" [2 0 4]

LightSource "spotlight"
　"intensity" default 1, range 0 to an unlimited upper value,
　"lightcolor" default [1 1 1], range 0 to 1 for each component,
　"from" default [0 0 0] , unlimited range from positive to negative.
　"to" default [0 0 1], unlimited range from positive to negative,
　"coneangle" default ? (30), range from 0 to ?,
　"conedeltaangle" default ? (5), range from 0 to ?
　"beamdistribution" default 2, range from 2 to ?.

　LightSource "spotlight" 4 "intensity" 12 "lightcolor" [0.2 0.5 1.0] "from" [2 0 4]
　"coneangle" 0.349 "conedeltaangle" 0.017 "beamdistribution" 3

Note: cone angle and cone delta angle are measured in radians – one degree equals 0.01745 radians.

Positioning Lights in Space

The world is orientated with respect to the camera by a sequence of rotations and translations, for example,

```
Translate  0 0 9
Rotate  -120  1 0 0
Rotate  25  0 0 1
```

… whilst lights are positioned by their "from" and "to" parameters.

The spot light shown opposite is positioned at x = -3, y = -2, z = 4 units, and is aimed at the world origin **only** if it appears within a RIB file **immediately** before or after WorldBegin, for example,

```
Display…
Projection…
Format…

Camera transformations…

LightSource "spotlight"… (here)
WorldBegin
    LightSource "spotlight"… (or here)
    Objects…
WorldEnd
```
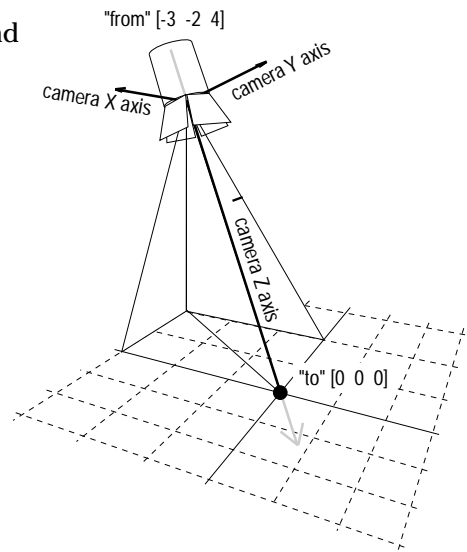
Light sources created **before** the statement WorldBegin have their "from" and "to" locations positioned within the camera coordinate system and NOT the world coordinate system. Consequently, rotations and translations applied to the camera are also applied to the lights – in effect, the lights are attached to the camera in much the same way as a flash gun can be fixed to a real camera.

Since our RIB files are 'hand made', positioning lights in the way shown above is very convenient. It is possible to perform rotations and translations on lights – they behave just like other object. But for now this added complication will be avoided.

Advanced Lighting – casting shadows
where the sun never shines!

Overview

Shadows in 'real life' are normally considered to be an integral part of scene – a natural outcome of the interplay of light and objects. In 3D graphics, however, a shadow is treated as if it were an additional element. Infact, shadow casting is an extra facility that must be explicitly 'turned on'.

When lit by one of the **basic light sources**, objects are shaded according to the orientation and optical characteristics of their surfaces. But basic light sources are unable to **cast** shadows of objects onto other objects. With the exception of an ambientlight, each of the basic light sources has a counter-part that allows shadow casting to occur. For example, instead of using a basic distant light in a scene, such as

    LightSource "distant" 1 "from" [3 3 3] "to" [0 0 0]

its shadow casting equivalent could be used ie.

    LightSource "shadowdistant" 1 "from" [3 3 3] "to" [0 0 0] "shadowname" "shadow1.tx"

For a complete description of the shadowdistant and other shadow casting light sources refer to Appendix C – Shader Reference, pages 12 to 14.

Shadow casting is not an automatic attribute of every light source because the renderer is required to perform additional calculations in order to determine the location and interaction of the shadows. This makes the rendering process slow and, in some situations, it is not always necessary to have shadows. Unlike the technique known as ray-tracing, in which the shadows are created along with every other part of an image, RenderMan's so-called scan-line renderer calculates the contribution that each "shadow casting" light source will make to the final image BEFORE it is rendered. This method of creating shadows is more efficient than ray-tracing if, for example, over part of an animation the lights and objects in a scene remain in a fixed relationship to each other. During such periods of 'static' lighting the renderer need only perform the lengthy shadow calculations on the first frame, there after it can apply the pre-calculated shadow information to all subsequent frames relatively quickly.

Pre-processing the shadows before making the final image also offers more creative control. For example, the renderer can be forced to use the "wrong" shadow information. In this way light sources can be made to cast shadows around corners, or an object can cast the shadow of an entirely different object. Such manipulations are beyond the capabilities of most ray-tracers.

For each light source that will create a shadow(s) in a scene, RenderMan requires a special "image" to be generated from the view-point of the light source. However, these special output files do not contain images as such, but store information about how far away each part of the scene is from the light source.

The following example is a very simple scene lit by a single (shadowing) distant light.
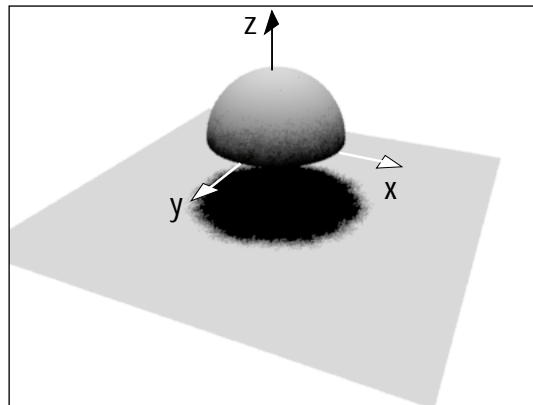
LightSource shadowdistant" "intensity" 1.5 "from" [0 0 4]
"to" [0 0 0] "shadowname" "shadow.tx"

step 1   The first step in creating shadows is the production of a depth map for each (shadow) light source. Because the output file from this step only contains depth information it is unnecessary, when defining the scene from the view point of a light, to specify the colour or surface properties of the objects themselves, for example,

```
FrameBegin 1
    Display "depth.pic" "zfile" "z"
    Format 128 128 1
    Projection "perspective" "fov" 110

    Translate 0 0 4     # equivalent to moving the view-point 4
    Rotate 180 0 1 0   # units directly above the origin of the scene

    WorldBegin
        ObjectInstance 1
        ObjectInstance 2
    WorldEnd
    MakeShadow "depth.pic" "shadow.tx"
FrameEnd
```

Notice how this first step is contained within its own frame block. This isolates it from remainder of the RIB script that produces the final full coloured rendered image.

The first frame produces two files, namely, "depth.pic" and "shadow.tx". The display statement that produces "depth.pic" uses two new parameters,

```
Display "depth.pic" "zfile" "z"
```

The inclusion of the letter "z" indicates to the renderer that it is to produce a depth map rather than a normal full colour image. Step 1 is concluded with the production of a texture file from the depth map,

```
MakeShadow "depth.pic" "shadow.tx"
```

step 2   In the second step the texture file(s) produced in step 1 is (are) used by the appropiate light source(s) to calculate the correct lighting values for each part of the scene, for example,

```
FrameBegin 2
    Display "half ball.tiff" "tiff" "rgba"
    Format 400 300 1
    Projection "perspective" "fov" 40

    Translate 0 1 8
    Rotate -120 1 0 0
    Rotate 25 0 0 1

    WorldBegin
        LightSource "shadowdistant"  1 "intensity" 1.5
        "from" [0 0 4] "to" [0 0 0] "shadowname" "shadow.tx"
        Color .5 .5 .5
        Surface "matte"
        ObjectInstance 1
        Color .5 .5 .5
        ObjectInstance 2
    WorldEnd
FrameEnd
```

There are several points to be noted in this example. Strictly speaking the RIB script, for the sake of simplicity, has produced an incorrect shadow! Distant light sources behave much like the sun – they produce shadows with parallel light. In the first frame the depth map was made using perspective projection with a field of view large enough to "see" the entire scene.

```
Projection "perspective" "fov" 110
```

Without this simplification it would have been necessary to use orthographic projection and to scale the scene in order for it to "fit" into a viewing space 1 unit by 1 unit – the dimensions of an orthographic viewing frame.

Alternatively, if the shadow version of a pointlight had been used instead of a shadowdistant it would have been necessary to produce 6 depth maps – each one corresponding to the 6 directions a point light source can radiate light ie. top, bottom, left, right, front and back! The important point is that although the final image is technically incorrect it is still CONVINCING.

Example 1
complete script

```
# Experiments with single shadows

ObjectBegin 1
    Sphere 1 0 1 360
ObjectEnd
ObjectBegin 2
    Polygon "P" [-3 3 -1  -3 -3 -1  3 -3 -1   3 3 -1 ]
ObjectEnd

FrameBegin 1
    Display "depth.pic" "zfile" "z"
    Format 128 128 1
    Projection "perspective" "fov" 110

    Translate 0 0 4
    Rotate 180 0 1 0

    WorldBegin
        ObjectInstance 1
        ObjectInstance 2
    WorldEnd
    MakeShadow "depth.pic" "shadow.tx"
FrameEnd

FrameBegin 2
    Display "half ball.tiff" "tiff" "rgba"
    Format 400 300 1
    Projection "perspective" "fov" 40

    Translate 0 1 8
    Rotate -120 1 0 0
    Rotate 25 0 0 1

    WorldBegin
        LightSource "shadowdistant" 1 "intensity" 1.5 "from" [0 0 4]
        "to" [0 0 0] "shadowname" "shadow.tx"
        Color .5 .5 .5
        Surface "matte"
        ObjectInstance 1
        Color .5 .5 .5
        ObjectInstance 2
    WorldEnd
FrameEnd
```

When using a light source that creates shadows the renderer determines if a point in a 3D scene lies within a shadow cast by another object by comparing two distances. Firstly, it calculates the true distance from the point to the light source and then it compares this value to the corresponding distance in the texture file that was produced from a depth map. If the true distance is the larger of the two then the screen pixel corresponding to the 3D point is shaded a dark colour appropriate to a shadow. Alternatively, if the true distance is smaller then the screen pixel is assigned the colour and brightness of the corresponding 3D point on the surface of the object casting the shadow.

The true distance is the greater, therefore, point "P" is in shadow.

Both distances are the same, therefore, point "P" is not in shadow.
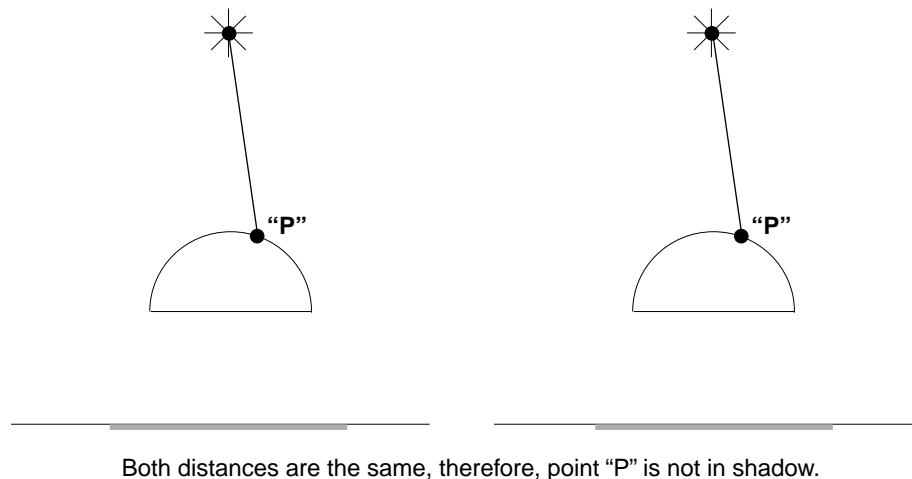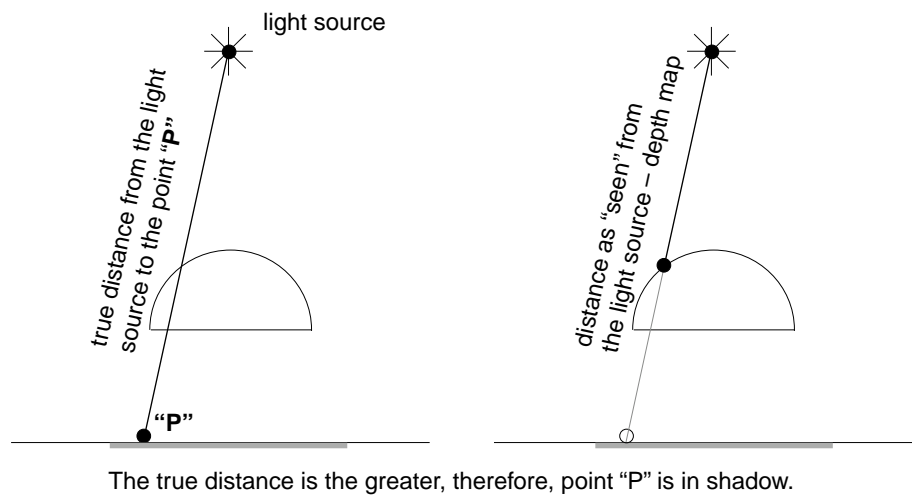
Using FrameUP to animate a scene with shadows

An example animation

The next two pages list a sample file that can be read and converted to an animation RIB file by the utility program FrameUP. Unlike the animation files found in the previous section each KeyFrameBegin/KeyFrameEnd block contains two frames. The first creates the texture file used by the shadow-distant light found in the second frame of each of the two key frames.

RIB script

```
#Example file for creating shadows using FrameUP
Option "limits" "bucketsize" [32 32]
ShadingRate 4
Display "test1""file"  "rgb"

ObjectBegin 1
    Polygon "P" [-3 3 -1  -3 -3 -1  3 -3 -1 3 3  -1]
ObjectEnd
ObjectBegin 2
    Cone 2 1 360
ObjectEnd

Tween "from" 1 "to" 2 "frames" 100 "smooth"

KeyFrameBegin 1
    FrameBegin
        Display "shadow1.pic" "zfile" "z"
        Format 512 512 1
        Projection "perspective" "fov" 90

        #Position of shadowlight 1
        Rotate -153.4   1 0 0
        Rotate -90.0    0 0 1
        Translate 2.0  0.0  -4.0

        WorldBegin
            ObjectInstance 1
            Rotate 0 1 1 0
            ObjectInstance 2
        WorldEnd
        MakeShadow "shadow1.pic" "shadowmap1.tx"
    FrameEnd
    FrameBegin
        Display "test1""file"  "rgb"
        Format 400 320 1
        Projection "perspective" "fov" 40
        Translate 0 0 12
        Rotate -120 1 0 0
        Rotate 25 0 0 1
        WorldBegin
            LightSource "shadowdistant" 1 "intensity" 2
            "from" [-2 0 4] "to" [0 0 0] "shadowname" "shadowmap1.tx"
            LightSource "ambientlight" 2 "intensity" 0.2
            Surface "plastic"
            Color 1 0 0
            ObjectInstance 1
            Color 1 1 0
            Rotate 0 1 1 0
            ObjectInstance 2
        WorldEnd
    FrameEnd
KeyFrameEnd
```

calculate the shadow information for the shadowdistant light in key frame number 1

use the shadow information…

An example Animation
– continued

calculate the shadow inform-
ation for the shadowdistant
light in key frame number 2

use the shadow information...

```
KeyFrameBegin 2
  FrameBegin
    Display "shadow1.pic" "zfile" "z"
    Format 512 512 1
    Projection "perspective" "fov" 90

    #Position of shadowlight 1
    Rotate -153.4   1 0 0
    Rotate -90.0    0 0 1
    Translate 2.0  0.0  -4.0

    WorldBegin
      ObjectInstance 1
      Rotate 360 1 1 0
      ObjectInstance 2
    WorldEnd
    MakeShadow "shadow1.pic" "shadowmap1.tx"
  FrameEnd
  FrameBegin
    Display "test1""file"  "rgb"
    Format 400 320 1
    Projection "perspective" "fov" 40
    Translate 0 0 12
    Rotate -120 1 0 0
    Rotate 25 0 0 1
    WorldBegin
      LightSource "shadowdistant" 1 "intensity" 2
      "from" [-2 0 4] "to" [0 0 0] "shadowname" "shadowmap1.tx"
      LightSource "ambientlight" 2 "intensity" 0.2
      Surface "plastic"
      Color 1 0 0
      ObjectInstance 1
      Color 1 1 0
      Rotate 360 1 1 0
      ObjectInstance 2
    WorldEnd
  FrameEnd
KeyFrameEnd
```
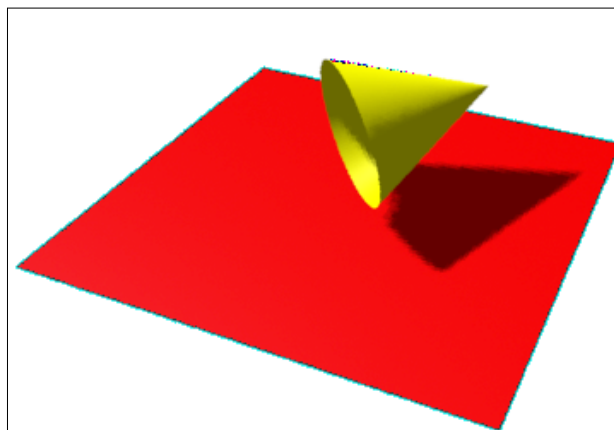

The lines printed in bold show the values that change during the animation.
The cone tumbles through 360 degrees around an axis half way between the x
and the y axes.

Importing Fragments

Overview   FrameUP allows geometry and other objects that have been saved in external files to be imported into a document using a command called Import. This statement copies the contents of an external file into a FrameUP document. For example, suppose a particularly effective collection of light sources had been used in a RIB script. The LightSource statements describing the lights could be saved in a separate file, perhaps called "spherical lighting" ie.

A sample fragment

RIB (fragment)

```
# a good set of modelling lights
# for use with rounded objects
LightSource "ambientlight" 1 "intensity" 0.1
LightSource "distantlight" 2 "intensity" 0.7 "from" [4 0 0] "to" [0 0 0]
LightSource "distantlight" 3 "intensity" 1.2 "from" [0 4 0] "to" [0 0 0]
```

This RIB 'fragment' could be could be re-used in another scene using the Import command eg.

Importing correctly

RIB

```
# using Import to insert a
# fragment of RIB script
Projection "perspective" "fov" 40
Display "square" "framebuffer" "rgba"
Format 200 200 1

Translate 1 0 23
Rotate 360 0 1 0
WorldBegin
  Import "fragment" "spherical lighting"
  Polygon "P" [-3 3 0  -3 -3 0  3 -3 0   3 3 0]
WorldEnd
```

Because FrameUP does not make any assumptions about the file that is being imported, and hence it does not check the validity of the file, it is your responsibility to ensure the contents of the imported file "make sense" in the context in which they are used. For example, it would be useless to import the lights in the manner shown below,

Importing incorrectly

RIB

```
# using Import to incorrectly insert a
# fragment of RIB script
Projection "perspective" "fov" 40
Display "square" "framebuffer" "rgba"
Format 200 200 1

Translate 1 0 23
Rotate 360 0 1 0
```

```
WorldBegin
   Polygon "P" [-3 3 0  -3 -3 0  3 -3 0   3 3 0]
   Import "fragment" "spherical lighting"
WorldEnd
```

In theory, RenderMan should allow external objects, which it refers to as 'entities', to be imported into a RIB file using a command called "Geometry". However, because this facility has not yet been implemented by PIXAR, the Import statement was added to the list of commands understood by FrameUP so that useful fragments of RIB script could be reused. The Import command also allows objects of arbitary complexity, perhaps generated by sophisticated modelling software, to be inserted into what are otherwise hand written FrameUP documents. Provided an object has been exported by a modeller as a RIB file and has been subsequently edited to remove any RIB statements that do NOT refer to the surfaces that comprise the object there should not be any rendering problems.

Fragments and objects

When making a library object, either by writing the RIB script by hand or with a modeller, it is essential to locate the local origin of the object in such a way that when the object is imported into a scene it can be positioned, rotated and scaled as conventiently as possibly. For example, in the case of a four legged chair the origin might best be located as follows,

On the other hand if the chair was a rocker it might be better to locate the origin as shown below.

When importing objects, rather than say light sources or other 'none surfaces' , it is best to insert a pair of AttributeBegin/AttributeEnd statements at the beginning and end of the file. In this way any transformations and shading used within the imported file will not have any unexpected effects on the objects added to a scene after the import command. Refer to the section Shaping Up ex.7b (page 22) for an explanation about the use of AttributeBegin/AttributeEnd.

Restrictions

Imported files cannot import copies of themselves or other fragments. For example, the following version of "spherical lighting" is illegal,

RIB (fragment)

```
# fragments cannot refer to themselves
# this will make FrameUP crash!!

LightSource "ambientlight" 1 "intensity" 0.1
LightSource "distantlight" 2 "intensity" 0.7 "from" [4 0 0] "to" [0 0 0]
LightSource "distantlight" 3 "intensity" 1.2 "from" [0 4 0] "to" [0 0 0]
Import "fragment" "spherical lighting"
```

Attributes that form part of an imported file cannot be changed unless the imported file is itself edited. It is sometimes better not to set the shading of an object within the import file. For example, suppose several chairs are required in a scene but each must be a different colour. Assuming the existance a library file called "seat" that contains the geometry defining the chair, it would be used as follows,

RIB

```
Projection "perspective" "fov" 40
Display "office seating" "framebuffer" "rgb"
Format 300 200 1

Translate  0 0 5
Rotate  -120  1 0 0
Rotate  25  0 0 1

Import "fragment" "spherical lighting"
WorldBegin
    Color  1  0  0
    Import "fragment" "seat"

    Color  1  0  0
    Import "fragment" "seat"

    Color  1  0  0
    Import "fragment" "seat"
WorldEnd
```

Appendix A – an overview of MacRenderMan

The diagram shown below represents the environment in which RIB scripts are written and rendered is shown below. Although it may appear to be complicated, working with MacRenderMan is reasonably straight forward. Initially, a RIB file is written using a word processor such as MicroSoft Word and is saved as "text only" with a **.rib** extension, for example, scene1.rib.

A small utility program called "RibWriter" can be used to write RIB scripts that correspond to a chosen camera position/angle, as well as a particular colour. The information from RibWriter can be copied and pasted into the RIB file being prepared in MS Word. "FrameUP" is described in the section dealing with Animation.

Either the vector (ie. line) or the photo-realistic renderer can be selected via "Chooser" in the same way as a network printer can be set-up for printing. The RIB file is sent to the chosen renderer via a small application called "RenderApp". If the RIB file has been written so that the rendered scene is to be displayed immediately, rather than being saved as a TIFF or PICT file, RenderApp will open a window for the rendered image.

Information about the rendering process is automatically reported via a utility called "RenderMonitor" in much the same way as the status of a network printer is displayed by "PrintMonitor".
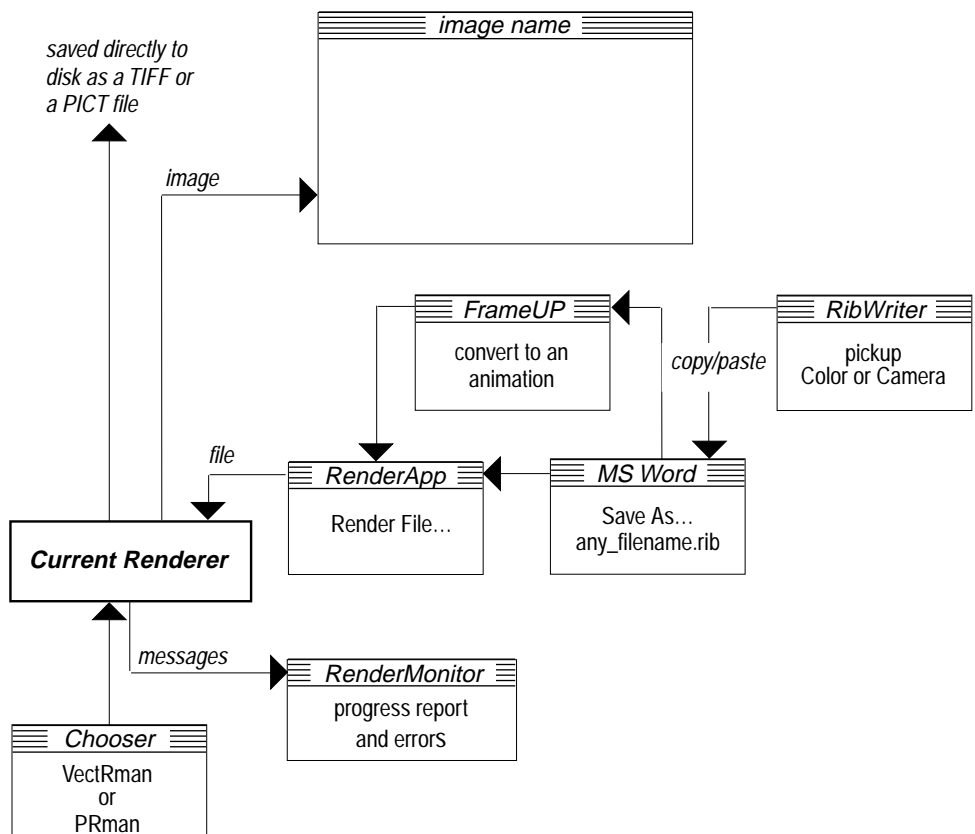
Appendix B – RenderMan Quick Reference

The reference material contained in this section is based largely on PIXAR's PhotoRealistic RenderMan Application Notes[1] #1 and #8 published in May 1990. The original "Quick Reference" was intended to be used by programmers and, therefore, contained a great deal of information that is irrelevant to those who wish to write RIB scripts directly by hand. I have tailored this reference to meet the needs of the Vcdn301 course. Several RIB statements have been omitted[2] because they relate to very advanced capabilities of the RenderMan interface. The information in this reference is of necessity very terse and is really only intended to act as a 'memory jogger'.

RIB Summary

Shape – geometric primitives

Cone
Cylinder
Disk
Hyperboloid
Paraboloid
Sphere
Torus
GeneralPolygon
PointsGeneralPolygons
PointsPolygons
Polygon

Space – transformations and grouping

Rotate
Scale
Skew
Translate
AttributeBegin/End
ObjectBegin/End
ObjectInstance
Sides
SolidBegin/End
TransformBegin/End
WorldBegin/End

Camera

Clipping
DepthOfField
Display
Exposure
Format
FrameAspectRatio
FrameBegin/End
MotionBegin/End
Perspective
Projection
Shutter

Shading

AreaLightSource
Atmosphere
Color
LightSource
MakeCubeFaceEnvironment
MakeLatLongEnvironment
MakeShadow
MakeTexture
Opacity
ShadingInterpolation
ShadingRate
Surface
TextureCoordinates

Bookkeeping

#
Declare
Option

---

1 PhotoRealistic RenderMan Application Note #1 "A Brief Introduction to the RenderMan Interface"; PhotoRealistic RenderMan Application Note #8 "RenderMan Quick Reference"

2 These include references to splines, trim curves, patches and patch meshes; transformation matrices; user defined coordinate systems; levels of detail; geometric approximation; bump mapping (not supported on the Macintosh platform); error handling; image filtering, sampling and quantization.

Shape – geometric primitives

Cone   Cone height radius thetamax parameters
Defines a partial or complete cone.
example
    Cone 0.5 0.5 270 "Cs" [1 0 0   1 0 0   1 1 1   1 1 1]

"Cs" defines colours for the parameter space, which in this example provides the cone with a red base and a white apex.

Cylinder   Cylinder radius zmin zmax thetamax parameters
Defines a partial or complete cylinder.
example
    Cylinder 0.5 0.2 1 360 "Os" [0 0 0   0 0 0   1 1 1   1 1 1]

"Os" defines opacity for the parameter space, which in this example provides the cylinder with a fully transparent base (opacity = 0,0,0) and a fully opaque top (opacity = 1,1,1).

Disk   Disk height radius thetamax parameters
Defines a partial or complete disk.
example
    Disk 1.0 0.5 270 "Os" [0 0 0   0 0 0   1 1 1   1 1 1]

Opacity is used here to give the disk a fully transparent rim and a fully opaque centre.

Hyperboloid   Hyperboloid x1 y1 z1 x2 y2 z2 thetamax parameters
Defines a partial or complete hyperboloid.
example
    Hyperboloid 1.0 -1.0 -1.0 1.0 1.0 1.0 360

Paraboloid   Paraboloid rmax zmin zmax thetamax parameters
Defines a partial or complete paraboloid.
example
    Paraboloid 0.5 0.2 0.7 270

Sphere   Sphere radius zmin zmax thetamax parameters
Defines a partial or complete sphere.
example
    Sphere 0.5 0.0 0.5 360
    "Cs" [1 0 0   1 0 0   0 0 1   0 0 1]
    "Os" [0.7 0 0   0.7 0 0   1 1 1   1 1 1]

Both opacity and colour are used for the parameter space, which in this example provides the sphere with a semi-transparent red "base" and an opaque blue "top".

Torus   Torus rmajor rmin phimin phimax thetamax parameters
                  Defines a partial or complete torus.
                  example
                      Torus 3.5 0.25 0.0 180 300

GeneralPolygon   GeneralPolygon nloops nvertices parameters
                      Defines a single convex or concave (general) planar polygon, with optional holes.
                      example
                          GeneralPolygon [3 3]
                          "P" [-1.0 -1.0 0.0 -1.0 1.0 0.0 1.0 -1.0 0.0 -0.5 -0.5 0.0   0.0 0.5 0.0
                           0.5 -0.5 0.0]

PointsGeneralPolygons PointsGeneralPolygons numLoops numVertices listVertices parameters
                          Defines several planar general polygons, with optional holes, that share vertices.
                          example
                              PointsGeneralPolygons [2 2][4 3 4 3][0 1 3 4 6 7 8 1 2 5 4 9 10 11]
                              "P" [0 0 1 0 1 1 0 2 1 0 0 0 0 1 0 0 2 0 0 0.2 0.5 0 0.7 0.7 0 1.7 0.2 0 1.2 0.5
                              0 1.7 0.7 0 1.7 0.2]

PointsPolygons   PointsPolygon numVertices listVertices parameters
                      Defines several non-concave polygons, without holes, that share vertices.
                      example
                          PointsPolygons [3 3 3][0 3 2 0 1 3 1 4 3]
                          "P" [0 1 1 0 3 1 0 0 0 0 2 0 0 4 0]

Polygon   Polygon listVertices parameters
                      Defines a single non-concave polygon with optional list of parameters that supplies
                      information about vertex normals, colour, opacity and/or texture coordinates.
                      examples
                      –no additional parameters
                          Polygon "P" [0.0 1.0 0.0   0.0 1.0 1.0   0.0 0.0 1.0   0.0 0.0 0.0]
                      –plus additional parameters relating to vertex colours
                          Polygon "P" [0.0 1.0 0.0   0.0 1.0 1.0   0.0 0.0 1.0   0.0 0.0 0.0]
                          "Cs" [1 0 0   0 1 0   0 0 1 1 1 1]
                      –plus additional parameters relating to vertex opacity
                          Polygon "P" [0.0 1.0 0.0   0.0 1.0 1.0   0.0 0.0 1.0   0.0 0.0 0.0]
                          "Os" [1 1 1   0 0 0   0.5 0.5 0.5   0.2 0.5 0.8]

Shape – transformations and grouping

Rotate

Rotate angle dx dy dz
Turns the object space so that it is rotated by "angle" degrees around the given axis prior to a shape being defined.
example
    Rotate 90 0 1 0

Scale

Scale sx sy sz
Stretches or compresses the object space so that it is scaled along the x, y and z axes prior to a shape being defined.
example
    Scale 0.5 1.0 1.0

Skew

Skew angle dx1 dy1 dz1 dx2 dy2 dz2
Shears the object space so that it is skewed by "angle" degrees along the x, y and z axes prior to a shape being defined.
example
    Skew 45 01 0 1 0 0

Translate

Translate dx dy dz
Shifts the origin of the object space so that it is translated along the x, y and z axes prior to a shape being defined.
example
    Translate 0.0 1.5 0.0

AttributeBegin/End

AttributeBegin/AttributeEnd
Forms a group of shapes, transformations and surface attributes.
example
    AttributeBegin
        some attributes such as color 1 0 0
    AttributeEnd

ObjectBegin/End

ObjectBegin identifier/ObjectEnd
Defines a collection of shapes as a "retained" object that can be inserted, or instanced, within a scene.
example
    ObjectBegin 4
        object 1
        object 2…
    ObjectEnd

ObjectInstance

ObjectInstance identifier
Inserts, or instancies, a previously retained collection of shapes as a single object.
example
    ObjectInstance 4

Sides
Sides sides
Defines subsequent shapes as single sided or double sided.
example
> Sides 2

SolidBegin/End
SolidBegin operation/SolidEnd
Defines a collection of shapes as a "solid" object according to the rules of constructive solid modelling ie. union, intersection and difference.
example
> SolidBegin "union"
>> define two or more objects to be joined together as a single object
>
> SolidEnd

TransformBegin/End
TransformBegin/TransformEnd
Forms a group of shapes and transformations but IGNORES surface attributes.
example
> TransformBegin
>> some transformations ex. Rotate
>> some shapes
>
> TransformEnd

WorldBegin/End
WorldBegin/WorldEnd
Freezes the characteristics of the camera and marks the beginning of a world description.
example
> WorldBegin
>> scene description
>
> WorldEnd

Camera

Clipping	Clipping near far
	Sets the near and far clipping planes along the direction of view.
	example
	Clipping 0.1 1000

DepthOfField	DepthOfField fstop focallength focaldistance
	Parameters to simulate the depth of field.
	example
	DepthOfField 22 1 26.7

Display	Display name type mode parameters
	Chooses a display by name and sets the type of output being generated.
	examples
	Display "filename" "file" "rgba"
	Display "filename" "zfile" "z"
	Display "windowname" "framebuffer" "rgba"

Exposure	Exposure gain gamma
	Controls the sensitivity and non-linearity of the exposure process.
	example
	Exposure 1.5 2.3

Format	Format xresolution yresolution pixelaspectratio
	Sets the horizontal and vertical resolution in pixels of the image to be rendered.
	example
	Format 400 300 1

FrameAspectRatio	FrameAspectRatio ratio
	Ratio sets the ratio of the width to height of the desired image.
	example
	FrameAspectRatio 1.333

FrameBegin/End	FrameBegin/FrameEnd
	Marks the beginning and end of a frame of animation.
	example
	FrameBegin 1
	scene description for this frame
	FrameEnd

MotionBegin/End	MotionBegin t0 t1…tn-1/MotionEnd
	Marks the beginning and end of motion
	example
	MotionBegin [0 1]
	transformation information at time 0
	transformation information at time 1
	MotionEnd

MotionBegin/End   MotionBegin t0 t1…tn-1/MotionEnd

Marks the beginning and end of motion

example

    MotionBegin [0 1]

        transformation information at time 0

        transformation information at time 1

    MotionEnd

Perspective   Perspective fov

Sets the camera to give a perspective view.

example

    Perspective 90

Projection   Projection name parameters

Sets the type of projection and activates the camera coordinate system ie. the world coordinate system is only active between WorldBegin and WorldEnd.

example

    Projection "perspective" "fov" 40

Shutter   Shutter opentime closetime

Sets the times at which the shutter opens and closes.

example

    Shutter 0 1

Shading

AreaLightSource AreaLightSource name int parameters

Creates an area light and makes it the current light source. Each subsequent object is added to the list of surfaces that define the area light.

example

    AreaLightSource "finitelight" 1 "decayexponent" 0.5
    AreaLightSource "glowlight" 2 "color" [0.5 0 0] "intensity" 0.6

Atmosphere Atmosphere name parameters

Sets the currently active atmosphere shader.

examples

    Atmosphere "fog" "background" [0.2 0.2 0.3] "distance" 39.4
    Atmosphere "depthcue" "background" [0.2 0.2 0.3] "mindistance" ?
    "maxdistance" ?

Color Color red green blue

Sets the colour that will be applied to subsequent objects.

example

    Color 0.2 0.3 0.9

LightSource LightSource name sequencenumber parameters

Creates a non-area ie. infinitely small, light source, turns it on, and adds it to any other lights previously created.

example

    LightSource "ambient" 2 "intensity" 10

MakeCubeFaceEnvironment

MakeCubeFaceEnvironment px nx py ny pz nz texturename fov filter swidth twidth parameters

Converts six images in a standard picture file (for example a TIFF file) representing six viewing directions into an environment map.

example

    MakeCubeFaceEnvironment "foo.x" "foo.nx" "foo.y" "foo.ny" "foo.z" "foo.nz"
    "foo.env" 95 "gaussian" 2.0 2.0

MakeLatLongEnvironment

MakeLatLongEnvironment picturename texturename filter swidth twidth parameters

Converts an image in a standard picture file (for example a TIFF file) representing a latitude-longitude map whose name is picturename into an environment map called texturename.

example

    MakeLatLongEnvironment "long.tiff" "long.tx" "gaussian" 2 2

MakeShadow

    MakeShadow picturename texturename parameters

    Convers a depth image file into a shadow map.

    example

        MakeShadow "shadow.tiff" "shadow.tx"

MakeTexture   MakeTexture picturename texturename swrap twrap filter swidth twidth

    Converts an image in a standard picture file (eg. TIFF) into a texture file.

    examples

        MakeTexture "globe.tiff" "globe.tx" "periodic" "periodic" "gaussian" 2 2

        MakeTexture "globe.tiff" "globe.tx" "black" "black" "gaussian" 2 2

        MakeTexture "globe.tiff" "globe.tx" "clamp" "clamp" "gaussian" 2 2

    In the first example the image will if necessay repeat horizontally and vertically. In the second example the image will be mapped once and will be surrounded by black. While in the last example, the colour of the pixels image at the extreme edge of the image will be "smeared" outward if there is enough space available on the object being texture mapped.

Opacity   Opacity c1 c2 c3

    Sets the opacity to the colour channels c1, c2, c3, like the use of Color, subsequent objects are set to these levels of opacity.

    example

        Opacity 0.5 1.0 1.0

ShadingInterpolation   ShadingInterpolation type

    Controls how the values are interpolated ie. estimated, between shading samples.

    examples

        ShadingInterpolation "constant"

        ShadingInterpolation "smooth"

ShadingRate   ShadingRate size

    Sets the number screen pixels, across and down the image, the renderer will skip between making its shading calculations – large numbers give fast but coarse images. The skipped pixels are either shaded with constant or "smoothed" colour – see above.

    example

        ShadingRate 10

Surface   Surface name parameters

    Sets the current surface shader, subsequent surfaces acquire the 'look' of the chosen surface.

    example

        Surface "wood" "roughness" 0.3 "Kd" 1.0

TextureCoordinates   TextureCoordinates s1 t1 s2 t2 s3 t3 s4 t4

    Sets the current set of texture coordinates.

    example

        TextureCoordinates 0.0 0.0   2.0 -0.5   -0.5 1.75   3.0 3.0

Bookkeeping

any text upto the **end of a line** is a comment

Enables notes to be included in a RIB file and ensures these will be ignored by the renderer.

examples

    # this is a comment
    Color 1 0 0 #this is another comment

Declare   Declare name declaration

Declares a non-standard parameter.

example

    Declare "centrepoint" "uniform float"

Option   Option name parameterslist

Allows any pre-set option to be set from within a RIB file.

example

    Option "limits" "bucketsize" [24 24]
    Option "limits" "texturememory" [1024]

Appendix C – Shaders Reference

This reference provides information about the basic shaders that support RenderMan. The data has be compiled from "The RenderMan Companion" by Steve Upstill and the PIXAR document, "MacRenderMan Shaders" published (August 1990) as part of "MacRenderMan Developers Stuff".

Like 'plug-ins' for the Adobe image processing software PhotoShop, an almost unlimited number of (potential) shaders can be added to a RenderMan environment – this document only describes the essential ones.

Shader Summary

LightSource
ambientlight
distantlight
pointlight
spotlight
pointnofalloff
shadowdistant
shadowpoint
shadowspot

Atmosphere
depthcue
fog

Displacement
cloth
dented
diaknurl
droop
emboss
filament
sinknurl
threads

Surface
blue_marble
carpet
checker
cmarble
glass
glassbal
glow
matte
metal
plastic
rmarble
rsmetal
rubber
screen
show_st
show_xyz
sinknurl
spatter
stippled
stone
texmap
txtplastic
wood
transparent_texture
eroded

Surface Shaders

blue_marble

"blue_marble" "Ks" "Kd" "Ka" "roughness" "txtscale" "specularcolor"

"Ks" 0.4 "Kd" 0.6 "Ka" 0.1
"roughness" 0.1
"txtscale" 1
"specularcolor" [1 1 1]

This shader gives a surface a very delicate marble-like appearance. It gives the visual impression of turbulent fluid flow, as if the marble had been formed by molten coloured rocks. The txtscale parameter scales the turbulence.

carpet

"carpet" "Ka" "Kd" "scuff" "nap"

"Ka" 0.1 "Kd" 0.6
"scuff" 1 "nap" 5 "swirl" 1

This shader produces a carpeted surface, complete with scuff-marks. The scuff parameter controls the "amount of scuff", or the relative frequency of intensity variations. Higher values produce more frequent scuffing. nap describes the "shagginess" of the carpet. Higher values make a more coarse-looking carpet.

The carpet shader makes a reasonable stab at anti-aliasing, so the actual grain of the carpet fades away with distance.

There are no specular reflections from real carpet (at least on a macroscopic scale), so the only lighting parameters are Ka and Kd, which have the usual meanings of ambient and diffuse reflective intensities, respectively.

This way anti-aliasing is performed can cause linear artifacts in some cases.

checher

"checker" "Kd" "Ka" "frequency" "blackcolor"

"Kd" 0.5 "Ka" 0.1
"frequency" 10
"blackcolor" [0 0 0]

This shader imposes a checker-board pattern over a surface. The frequency parameter sets how many times the pattern is to repeat itself within the texture space of the surface. Blackcolor sets the colour to be used for the pattern.

cmarble   "cmarble" "Ka" "Ks" "Kd" "roughness" "specularcolor" "veining"

"Ka" 0.1 "Ks" 0.4 "Kd" 0.6
"roughness" 0.1
"specularcolor" [1 1 1]
"veining" 1

This shader produces a marble that consists of coloured veins on a white background with some greyish mottling. The vein colour is determined by the current surface colour. The parameter veining controls the frequency of the veins in the marble; higher values produce more and narrower veins.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

glass   "glass" "Ka" "Ks" "Kd" "roughness" "specularcolor" "envname" "Kr"
glassbal   "glassbal" "Ka" "Ks" "Kd" "roughness" "specularcolor" "envname" "Kr" "eta"

"Ka" 0 "Ks" 0.6 "Kd" 0
"roughness" 0.025 (for glass) or 0.2 (for glassbal)
"specularcolor" [1 1 1]
"envname" "your environment map"
"Kr" 0.5 "eta" 0.6

The glass shader makes an object appear to be made out of transparent and possibly coloured glass. No refraction is attempted, so the glass appears to be thin, but reflections are simulated using the environment map given with envname. The environment map's reflective intensity Kr can be controlled to fine-tune the appearance; a lower-intensity reflection may look better on very dark glass. Although you can use the shader even if you don't have an environment map (just ignore envname), it will look more like transparent plastic then glass.

You can use the shader for either clear glass or coloured glass by setting the surface colour (clear glass has color [1 1 1]). The transparency of the glass is controlled only by the surface colour; if you want to make the glass less transparent you should make the colour darker.

The glassbal shader can be used to make some objects look like solid glass. It simulates the refraction seen through a sphere turning objects seen through the glass upside-down and backwards. The shader needs the environment map given with envname to this refraction. It will not work without an environment map. Because the shader simulates refraction as if the object is a sphere, it works well on curvy objects like teapots (and spheres), but will not look correct on flat objects or cylinders.

The colour of the object can be set with the surface colour, just as with glass. The transparency is somewhat different here, however, because the camera is not really seeing "through" the object. Therefore, the surface opacity should always be set to [1 1 1], and the relative intensity of the "refraction" will again depend on the surface colour.

The parameter eta is the relative index of refraction of the atmosphere to the glass. By default, this is the standard value of air (1.0) relative to glass (1.66).

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

glassbal needs an environment map; glass looks bad without one.

glow

"glow" "attenuation"

"attenuation" 2

This shader imparts a glow to a surface. The glow is brightest when looking directly at the glowing object, and falls off rapidly nearby.

matte

"matte" "Ka" "Kd"

"Ka" 1 "Kd" 1

A matte surface exhibits only diffuse reflection, because it scatters light uniformly with no preferred direction. That makes the apparent brightness of such a surface independent of the direction from which it is viewed.

metal

"metal" "Ka" "Ks" "roughness"

"Ka" 1 "Ks" 1
"roughness" 0.25

Very similiar to the matte surface shader except that this surface allows specular reflections to occur ie. reflections are concentrated around the mirror direction. Roughness controls the concentration of the specular highlight, a high roughness value giving a more diffuse reflection.

plastic

"plastic" "Ka" "Ks" "Kd" "roughness" "specularcolor"

"Ka" 1.0 "Ks" 0.5 "Kd" 0.5

"roughness" 0.1
"specularcolor" [1 1 1]

This shader models a plastic material as a solid medium with microscopic coloured particles suspended within it. The specular highlight is assumed to be reflected directly off the surface, and the surface colour is assumed to be due to light entering the medium, reflecting off the suspended particles, and re-emerging. This explains why the colour of the specular reflection is different from the surface.

rmarble

"rmarble" "Ka" "Ks" "Kd" "roughness" "specularcolor" "veining"

"Ka" 0.1 "Ks" 0.4 "Kd" 0.6
"roughness" 0.1
"specularcolor" [1 1 1]
"veining" 1

This shader produces a marble that consists of red veins on a white background with some greyish mottling. The parameter veining controls the frequency of the veins in the marble; higher values produce more and narrower veins.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

rsmetal

"rsmetal" "Ka" "Ks" "Kr" "roughness"

"Ka" 1.0 "Ks" 1.0 "Kr" 1.0
"roughness" 0.1

This shader uses random data for the reflection instead of requiring an environment map. The parameter Kr controls the intensity of the reflection. This shader is not recommended for simple spheres, it produces a chrome-plated look on objects with more complex curvature.

rubber

"rubber" "Ka" "Kd" "txtscale"

"Ka" 1.0 "Kd" 1.0
"txtscale" 1.5

This shader is similiear to a matte shader except that it includes small amounts of white dust in the surface of the rubber. The amount of dust is controlled by the parameter txtscale.

screen   "screen" "Ks" "Kd" "Ka" "roughness" "density" "frequency" "specularcolor"

                            "Ks" 0.5 "Kd" 0.5 "Ka" 0.1
                            "roughness" 0.1
                            "density" 0.25 "frequency" 20
                            "specularcolor" [1 1 1]

                            This shader produces a wire-frame appearance. The frequency parameter
                            controls how many grid lines there are in the surfaces texture space; the
                            default produces 20 grid lines per surface. The density parameter controls the
                            portion of the surface that is opaque. The default 0.25 means that the "wires"
                            will cover 25% of the texture space.

show_st   "show_st"
                            no parameters

                            For each point on a surface, this shader sets its red and green colour equal to
                            the texture coordinates at that point. Transparency cannot be set with this
                            shader - all surfaces are set to be opaque.

show_xyz   "show_xyz" "xmin" "ymin" "zmin" "xmax" "ymax" "zmax"

                            "xmin" -1 "ymin" -1 "zmin"-1
                            "xmax" 1 "ymax" 1 "zmax" 1

                            This shader converts points within a bounding box, given by the parameters,
                            into red, green and blue values.

spatter   "spatter" "Ka" "Ks" "Kd" "roughness" "specularcolor" "basecolor"
                            "spattercolor" "specksize" "sizes"

                            "Ka" 1 "Ks" 0.7 "Kd" 0.5 "roughness" 0.2
                            "specularcolor" [1 1 1] "basecolor" [0.1 0.1 0.5] "spattercolor" [1 1 1]
                            "specksize" 0.01 "sizes" 5

                            This shader makes objects look like blue camp cookware with white paint
                            spatters. Actually, both the blue basecolor and the white spattercolor can be
                            changed if you desire.

                            The paramter specksize controls the size of the paint specks as you would
                            expect. However, there are a range of sizes of paint specks controlled by the
                            parameter sizes. Lower (integer) values produce smaller and more uniform
                            specks. Higher values produce some larger blotches and specks of many
                            different sizes.

The parameters Ka, Ks and Kd, have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. roughness and specularcolor control the sharpness and colour of the specular highlight.

This shader can have problems with aliasing.

stippled

"stippled" "Ka" "Ks" "Kd" "roughness" "specularcolor" "grainsize" "stippling"

"Ka" 0.1 "Ks" 0.3 "Kd" 0.8
"roughness" 0.3
"specularcolor" [1 1 1]
"grainsize" 0.01
"stippling" 0.2

This shader makes objects appear to be made of plastic with lots of little bumps, as computer keyboards, camera surfaces, stucco and many other objects. This is done by making the surface appear to have intensity variation in small grains or granules. The parameter grainsize controls the size of these granules, and stippling controls the relative variation in intensity of the granules; larger values produce a rougher looking surface.

This shader makes a fairly good attempt at anti-aliasing itself, so the granules should appear to fade out with distance in a way similiar to a 'real' stippled surface.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

stone

"stone" "Ka" "Ks" "Kd" "roughness" "specularcolor" "scale" "nshades" "exponent" "graincolor"

"Ka" 0.2 "Ks" 0.9 "Kd" 0.8
"roughness" 0.3
"specularcolor" [1 1 1]
"scale" 0.02 "nshades" 4
"exponent" 2
"graincolor" [0 0 0]

This shader makes objects look like they are carved our of grainular stone, like granite, by making "crystals" of varying intensity and colour. The parameter scale controls the size of the "crystals", or grains; larger values make larger grains. This is the only parameter that most users will want to change.

The parameter nshades is the number of unique intensity levels found in the

grains. Higher values of this will produce less "simplistic" looking stone. Setting nshades equal to 3 will produce stone that looks remarkably like the spattered-paint fake stone Zolatone. The exponent parameter controls the distribution of intensity levels; higher values push the intensities toward the darker end (more toward graincolor, as described below).

The "intensity" levels are actually levels of mixing two colours, the surface colour and the graincolor parameter. Since graincolor is black by default, the different colours normally are in fact different intensities of the surface colour. However, if you want red-and-green speckly stone for some reason, you could do this by setting the surface colour and graincolor appropriately, but you should probably set nshades to something pretty low to avoid getting lots of weird colours between red and green.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

This shader can have problems with aliasing.

texmap

"texmap" "Ka" "Ks" "Kd" "roughness" "specularcolor" "texname" "maporigin" "xaxis" "yaxis" "zaxis" "maptype "s1" "t1" "s2" "t2" "s3" "t3" "s4" "t4"

"Ka" 1.0 "Ks" 0 "Kd" 1.0
"roughness" 0.25
"specularcolor" [1 1 1]
"texname" "name of texture file"
"maporigin" [0 0 0]
"xaxis" [1 0 0] "yaxis" [0 1 0] "zaxis" [0 0 1]
"maptype" 3 (ie. no projection)
"s1" 0 "t1" 0   "s2" 1 "t2" 0
"s3" 0 "t3" 1   "s4" 1 "t4" 1

This shader texture maps a surface. The name of the texture file is given by texname. The parameters maporigin, xaxis, yaxis, zaxis, maptype, s1-4 and t1-4 are passes directly

Note that because the t component of a texture map is displayed on a monitor as increasing downward, textures mapped onto surfaces can easily appear to be upside-down. You should be careful to orient your coordinate system correctly when using projections for texture mapping. For example, if you are using planar projection you may want to have the y axis of the projection plane pointing down.

The maptype parameter indicates the following types of projection:
0 planar,

1 cylindrical,
2 spherical,
3 no projection, and
4 automap.

In the case of spherical mapping the parameters maporigin, xaxis, yaxis and zaxis describe the coordinate system of the projection sphere. The maporigin is naturally the center point of the sphere. The xaxis, yaxis and zaxis are points describing the 3 coordinate axes relative to maporigin. The texture map wraps around the "equator" of the sphere such that the seam is located on the positive x-axis of the sphere.

The surface is texture-mapped as if it were painted with the image in the file. Normal shading techniques are then used to render the surface, as specified in the normal way.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. Roughness and specularcolor control the sharpness and colour of the specular highlight.

txtplastic

"txtplastic" "Ks" "Kd" "Ka" "roughness" "specularcolor" "mapname"

"Ks" 0.5 "Kd" 0.5 "Ka" 1
"roughness" 0.1
"specularcolor" [1 1 1]
"mapname" "name of a texture file"

This shader is based on the "plastic" surface shader. The parameter mapname allows an image previously converted to a texture file to be mapped onto a surface.

wood

"wood" "Ka" "Ks" "Kd" "roughness" "specularcolor" "grain" "swirl" "swirlfreq"
"c0" "c1" "darkcolor"

"Ka" 1 "Ks" 0.4 "Kd" 0.6 "roughness" 0.2
"grain" 5 "swirl" 0.25 "swirlfreq" 1
"specularcolor" [1 1 1]
"darkcolor" [dependent on the surface colour]
"c0" [0 0 0] "c1" [0 0 1]

This shader creates a realistic-looking wood. The frequency of the wood grain can be changed with the grain parameter. The relative amount or amplitude of the turbuent swirl in the grain is controlled by the swirl paramater, and swirfreq controls the frequency of this turbulence. Low values of swirl produce more uniform looking wood, while low values of swirlfreq make the wood

appear to be more knotty. Obviously these two parameters interact to a large extent. You should be careful not to set swirl too high or swirlfreq too low or the wood will become a jumbled mess.

The wood is simulated by creating a grain that is essentially composed of differently coloured concentric "cylinders" around a central axis defined by the two points c0 and c1. This axis is the z axis by default. Note that the orientat- ion of this axis can be varied either by changing these two parameters or by doing some transformations between the call to the shader and the definition of the geometry. Either one of these approaches may make more intuitive sense in different applications.

The colour of the wood will normally consist of bands of different intensities of the surface colour. This is the most generally useful way of invoking the shader. However, for special appearances this can be changed by changing the darkcolor parameter, which controls the colour of the dark grain of the wood. The different intensity levels are actually levels of mixing between this colour and the surface colour, so setting the surface colour to red and darkcolor to white will produce red wood with white grain and various shades in between.

The parameters Ka, Ks and Kd have the usual meanings of ambient, specular and diffuse reflective intensities, respectively. roughness and specularcolor control the sharpness and colour of the specular highlight. This shader can have problems with aliasing.

transparent_texture "Ks" "Kd" "Ka" "roughness" "specularcolor" "texname" "traname"

"Ks" 0 "Kd" 1.0 "Ka" 1.0 "roughness" 0.1
"specularcolor" [1 1 1]
"texname" ["name of the texure file to be used for texturing"]
"traname" ["name of the texure file to be used for transparency"]

This shader uses two texture files. Like the shader texmap, the file associated with "texname" is used as a texture map – except that this shader provides no control over the type of projection used. The other texture file, associated with "traname", controls the transparency of the surface(s) to which this shader is assigned. The gray scale values of "traname" alter the level of transparency of the shaded surface. Black pixels of the image make the corresponding parts of a surface fully transparent while white pixels renders the surface opaque.

eroded "eroded" "Ks" "Ka" "Km" "roughness"

"Ks" 0.4 "Ka" 0.1 "Km" 0.3
"roughness" 0.25

This shader erodes a "plastic-like" surface in such a way that parts of it are worn down to be transparent. The Km parameter controls the magnitude of the erosion.

LightSource Shaders

ambientlight
"ambientlight" "intensity" "lightcolor"

"intensity" 1
"lightcolor" [1 1 1]

An ambient light source supplies light of the same colour and intensity to all points on all surfaces

distantlight
"distantlight" "intensity" "lightcolor" "from" "to"

"intensity" 1
"lightcolor" [1 1 1]
"from" [0 0 0] "to" [0 0 1]

Unlike an ambient source, a distant source casts its light in only the direction defined by the from and to parameters. Otherwise the output light is the same as an ambient light.

pointlight
"pointlight" "intensity" "lightcolor" "from"

"intensity" 1
"lightcolor" [1 1 1]
"from" [0 0 0]

A point light source is the converse of a distant light. It radiates light in all directions, but from a single location. It has a from parameter, but no to.

spotlight
"spotlight" "intensity" "lightcolor" "from" "to" "coneangle" "conedeltaangle" "beamdistribution"

"intensity" 1.0
"lightcolor" [1 1 1]
"from" [0 0 0] "to" [0 0 1]
"coneangle" randians (30) "conedeltaangle" radians (5) "beamdistribution" 2

This shader reproduces the lighting effect of a spot light.

The intensity of the light varies from 0 (off) to any positive value (usually 1) representing the light at full intensity. The lightcolor parameter is an RGB triple representing the colour of light emitted by the source.

The from and to parameters specify the direction in which the light is shining. The coneangle and conedeltaangle parameters specify the distribution of the light as a cone-shaped beam, whose intensity falls off with the angle from the center to the cone. The falloff from the cone center is a "square-law" falloff (cosine of this angle raised to the power of 2) by default, but can be changed to a higher (or lower) power by setting the beamdistribution parameter.

pointnofalloff

"pointnofalloff" "intensity" "lightcolor" "from"

"intensity" 1.0
"lightcolor" [1 1 1]
"from" [0 0 0]

This is a point light shader without an inverse square intensity falloff. It is useful for lighting a scene when you don't want to go through the process of calculating the large intensity usually required to get a normal point light to look the way you want it. It is also useful for producing uniform lighting from objects inside a scene, without the "pooling" normally associated with point light sources.

In addition, some modelling systems, for example, work only with point light sources. In such a system some point lights may be placed far away from the scene to simulate distant lights, and this shader is a good choice in such circumstances.

The intensity of the light varies from 0 (off) to any positive value (usually 1) representing the light at full intensity. The lightcolor parameter is an RGB triple representing the colour of light emitted by the source.

The from parameter represents the position of the light source in space.

shadowdistant

"shadowdistant" "intensity" "lightcolor" "from" "to" "shadowname" "samples" "width"

"intensity" 1.0 "lightcolor" [1 1 1]
"from" [0 0 0] "to" [0 0 1]
"shadowname" (name of a shadow map)
"samples" 16
"width" 1

This is a normal distant light with an optional shadow map parameter given with shadowname. If a shadow map is not supplied the light behaves like a normal distant light source.

The paramter samples controls the sampling rate for filtering the shadow map. Higher values will produce less noisy-looking shadows, but will take

significantly longer. You can produce a (very noisy) test shadow very rapidly by setting samples to 1.

The width parameter controls "overfiltering" in the s and t directions. Higher values will give shadows more blurry edges, which can be used either as an effect or to hide the jagged edges or a low-resolution shadow map.

The intensity of the light varies from 0 (off) to any positive value (usually 1) representing the light at full intensity. The lightcolor parameter is an RGB triple representing the colour of light emitted by the source.

The from and to parameters specify the direction in which the light is shining.

shadowpoint

"shadowpoint" "intensity" "lightcolor" "from" "sfpx" "sfnx" "sfpy" "sfny" "sfpz" "sfnz" "samples" "width" "shadowname"

"intensity" 1.0
"lightcolor" [1 1 1]
"from" [0 0 0]
"sfpx" "sfnx" "sfpy" "sfny" "sfpz" "sfnz" (6 shadow maps)
"samples" 16
"width" 1
"shadowname"

This is a point light source that can cast shadows in all directions. To do this, you must supply 6 shadow maps, sfpx, sfnx, sfny, sfpz and sfnz for the positive and negative x, y and z directions respectively. This is very similiar to the idea of creating an environment map from 6 cube-face images. If any of the cube faces are not supplied, the shader will behave as a normal point light in those directions. For best results, the field of view for shadow images should be greater than 90 degrees (95 recommended).

The paramter samples controls the sampling rate for filtering the shadow map. Higher values will produce less noisy-looking shadows, but will take significantly longer. You can produce a (very noisy) test shadow very rapidly by setting samples to 1.

The width parameter controls "overfiltering" in the s and t directions. Higher values will give shadows more blurry edges, which can be used either as an effect or to hide the jagged edges or a low-resolution shadow map.

The intensity of the light varies from 0 (off) to any positive value (usually 1) representing the light at full intensity. The lightcolor parameter is an RGB triple representing the colour of light emitted by the source.

The from parameter specifies the position of the light source in space.

shadowspot   "shadowspot" "intensity" "lightcolor" "from" "to" "coneangle" "conedeltaangle"
"beamdistribution" "shadowname" "samples" "width"

"intensity" 1.0
"lightcolor" [1 1 1]
"from" [0 0 0] "to" [0 0 1]
"coneangle" randians (30) "conedeltaangle" radians (5) "beamdistribution" 2
"shadowname" "name of the shadow file"
"samples" 16 "width" 1

This light is a spotlight with an optional shadow map parameter
shadowname. If a shadow map is not used the light is a normal spotlight.

The paramter samples controls the sampling rate for filtering the shadow
map. Higher values will produce less noisy-looking shadows, but will take
significantly longer. You can produce a (very noisy) test shadow very rapidly
by setting samples to 1.

The width parameter controls "overfiltering" in the s and t directions. Higher
values will give shadows more blurry edges, which can be used either as an
effect or to hide the jagged edges or a low-resolution shadow map.

The intensity of the light varies from 0 (off) to any positive value (usually 1)
representing the light at full intensity. The lightcolor parameter is an RGB
triple representing the colour of light emitted by the source.

The from and to parameters specify the direction in which the light is shining.

The coneangle and conedeltaangle parameters specify the distribution of the
light as a cone-shaped beam, whose intensity falls off with the angle from the
center to the cone. The falloff from the cone center is a "square-law" falloff
(cosine of this angle raised to the power of 2) by default, but can be changed to
a higher (or lower) power by setting the beamdistribution parameter.

Displacement Shaders

cloth

"cloth" "freq" "depth"

"freq" 500
"depth" 0.02

This shader produces a cloth-like perpendicular weave pattern. The freq parameter changes the frequency of the "threads" (higher values mean the threads are closer together), and depth controls the height of the threads. The surface aliases pretty fiercely, but real cloth actually produces a somewhat similiar effect, so it looks fairly realistic.

dented

"dented" "Km"

"Km" 1.0

This shader produces a dented surface. The amount of denting is controlled by Km.

diaknurl

"diaknurl" "maporigin" "xaxis" "yaxis" "zaxis" "freq" "depth" "width" "radius" "zmin" "dampzone"

"maporigin" [0 0 0]
"xaxis" [1 0 0] "yaxis" [0 1 0] "zaxis" [0 0 1]
"freq" 10 "depth" 0.25 "width" 0.05
"radius" (refer to notes)
"zmin" "zmax" (refer to notes)
"dampzone" 0

This shader cuts a diamond knurl pattern into a cylindrical object. The parameters maporigin, xaxis, yaxis, and zaxis are used to do a cylindrical projection.

The freq parameter gives the number of grooves to cut per unit length along the z axis; higher values give closer grooves. The depth parameter controls the depth of the grooves, and the width parameter controls the width of the grooves. In order to render a correctly diamond-shaped pattern, the radius of the cylindrical object must be given with the radius parameter.

By default, the diamond knurl will be rendered along the entire length (along the z axis in shader space) of the cylinder. However, you can set minimum and maximum bounding z values with zmin and zmax parameters. The surface will not have knurl pattern cut outside these boundaries. In addition, you can make the knurl smoothly fade out instead of abruptly stopping at

these boundaries by setting the dampzone parameter. This parameter controls the width of the zone in which the depth of the grooves goes to zero. This zone is inside the zmin and zmax boundaries.

Remember to set the displacement bounds attribute when using this shader.

This shader can experience severe aliasing.

droop

"droop" "Km"

"Km"

This shader droops or sags a surface downward as if under the influence of gravity. 'Downward' for this shader means moving a surface in negative y.

emboss

"emboss" "Km" "texname"

"Km" 0.03
"texname" "name of a texture file that will control the embossing"

This shader embosses a surface according to an image given with the parameter texname. Pale areas of an image used for the texture file will push the surface "inward". The magnitude of the displacement is controlled by the paramter Km.

filament

"filament" "frequency" "phase" "width"

"frequency" 5.0
"phase" 0
"width" 0.3

This shader turns a cylinder into a spiral light-bulb filament. The filament can be brought to a point by applying the same shader to two cones at the ends of the cylinder. The frequency and phase parameters are identical to those of the "threads" shader.

sinknurl

"sinknurl" "maporigin" "xaxis" "yaxis" "zaxis" "freq" "depth" "zmin" "zmax" "dampzone"

"maporigin" [0 0 0]
"xaxis" [1 0 0] "yaxis" [0 1 0] "zaxis" [0 0 1]
"freq" 100 "depth" 0.005
"zmin" "zmax" (see notes)
"dampzone" 0

This shader cuts a sinusoidal knurl grooves along the length of a cylindrical object. The paramters maporigin, xaxis, yaxis and zaxis are used to do a cylindrical projection.

The freq parameter gives the number of grooves to cut around the circumference of the object. By default this number is quite high, but low numbers produce a shape like a classic Greek column. The depth parameter controls the depth of the grooves.

By default, the diamond knurl will be rendered along the entire length (along the z axis in shader space) of the cylinder. However, you can set minimum and maximum bounding z values with zmin and zmax parameters. The surface will not have knurl pattern cut outside these boundaries. In addition, you can make the knurl smoothly fade out instead of abruptly stopping at these boundaries by setting the dampzone parameter. This parameter controls the width of the zone in which the depth of the grooves goes to zero. This zone is inside the zmin and zmax boundaries.

Remember to set the displacement bounds attribute when using this shader.

This shader can have problems with aliasing.

threads

"threads" "maporigin" "frequency" "depth" "phase" "zmin" "zmax" "dampzone"

"maporigin" [0 0 0]
"frequency" 5 "depth" 0.1 "phase" 0
"zmin" "zmax" (refer to notes)
"dampzone" 0

This shader cuts a right-handed thread into a cylindrical object using the parameters maporigin, xaxis, yaxis, and zaxis to do a cylindrical projection.

The parameter freq gives the number of threads per unit length along the cylinder. The depth parameter controls the depth of the thread, and phase rotates the threads around the z axis of the cylinder. A value of 0 means no rotation and 1 means 360 degree rotation. This can be used to match threads from different cylinders.

By default, the threads will be rendered along the entire length (along the z axis in shader space) of the cylinder. However, you can set minimum and maximum bounding z values with zmin and zmax parameters. The surface will not have a thread pattern cut outside these boundaries. In addition, you can make the knurl smoothly fade out instead of abruptly stopping at these boundaries by setting the dampzone parameter. This parameter controls the width of the zone in which the depth of the grooves goes to zero. This zone is inside the zmin and zmax boundaries.

Remember to set the displacement bounds attribute when using this shader ie.

Attribute "bound" "displacement" [1.5]

This shader has problems with aliasing. The ShadingRate usually needs to be set to quite a low number.

Atmosphere Shaders

depthcue

"depthcue" "mindistance" "maxdistance" "background"

"mindistance" 0
"maxdistance" 1
"background" [0 0 0]

This atmospheric shader linearly adds the background colour according to the distance between the camera and a surface. No background colour is added if the surface is less than mindistance away. The background colour eliminates the surface colour entirely for points farther than maxdistance. In between, the two colours are mixed.

fog

"fog" "distance" "background"

"distance" 1
"background" [0 0 0]

This atmospheric shader is somewhat more realistic for emulating atmospheric absorption. It assumes that the attenuation of the surface colour in the fog is never complete, as it is in the depth-cue shader.

Appendix D
Project 1   Separating shape from shading

Overview

The purpose of this project is to explore the creative potential of displacement and texture maps; it is also intended to underscore the distinction between shape and shading – between the underlying geometry of an object and its outward visual apperance. Through this project you will learn how to

- create displacement and texture maps,
- control an abstraction called "parameter space"
- apply these maps to a quadric surface, and
- make a sphere as visually interesting as the surface of a **natural** object.

A simple object has been deliberately chosen for this project in order to focus your attention on controlling the process of shading, rather than shaping, an object.

You are to analyse two natural objects in terms of their suface attributes, these are to include such features as variations in colour, bumpiness, shininess (reflectivity), diffuseness and transparency. *In short, how they interact with light.* You will apply the characteristics of your chosen natural surfaces to two spheres. The natural objects, whose surfaces you are attempting to "re-create", need not be spherical. Your images will be judged by the extent to which they portray spherical versions of the original objects – no matter how inconguent that might be, for example, a spherical banana, leaf or toe-nail!

Submission

Your submission will consist of two **sets** of files. Each rendered image must be accompanied by

- the RIB file you wrote to create the finished image, and the
- the original images used for the texture and displacement maps.

We will aim to create high resolution images suitable for recording to 35mm transparency film via our LFR film recorder. The fully rendered images should have an alpha channel and be saved in the TIFF file format with a resolution of 1166 by 800 pixels. All images should be LZW compressed; this can be done using RenderApp or PhotoShop.

It is left for you to decide the resolution of the TIFF images that you will use for the texture and displacement maps. In general each "source" image should not exceed 5MB in size; again these should be compressed using LZW. Do NOT submit the texture files.

The files relating to each sphere must be located in their own folder ie. at least three files in total. These should be placed in a folder clearly identified with your name and should be copied to the archive server.

Project 2   Combining the 'real' and the 'imaginary'

Overview

In this project you will combine a computer generated scene with a photographic image of the interior of a building or other architectural space. The computer generated scene should consist of a simple object or objects viewed by a virtual camera set to match the characteristics of the camera that recorded the photograph ie. position, orientation, focal length and f-stop.

The project will be completed in two separate phases. In the first, the objects in the synthetic scene will be modelled, viewed and the resulting image composited with the photograph. At this stage NO particular attention will be given to lighting the synthetic scene or assigning realistic surface attributes to the models. Emphasis will be on

- matching the viewpoint of the virtual and 'real' camera,
- matching the scale, placement and orientation of the models with the scene portrayed in the photograph.

In the second phase, light sources will be added to the synthetic scene in order to match the illumination of the models with the 'real' interior. Appropriate surfaces will be assigned to the models and the refined virtual scene will again be viewed and composited with the photograph. Compositing will be done using PhotoShop and the synthetic scenes will be created using hand written RIB files rendered with RenderMan.

The aesthetics underpinning the final composition can be anything from the literal to the sarcastic; the synthetic objects can be modelled to be entirely appropriate to their photographic "home" or may be startlingly incongruent with the context provided by the photographic image.

Background

The traditional separation between computer generated imagery and "live action" recording is now almost a thing of the past. The integration of imagery derived from the inner representations of a computer system and scenes captured through standard photographic techniques is challenging our notions of what is cinematically real and believable. What makes "special effects" special is often not so much what is added in the post-production process–a dinosaur here, or a dinosaur there–but by what is subtly altered or removed. This might be the judicious removal of items that were thought to be out of camera when the original footage was filmed. It might be the background, props or even the actors themselves that an editor deems superfluous to a scene.

Research is well underway on the generation of photo-realistic synthetic actors and more especially, as in the case of the research by Thalman et al, synthetic actresses.

Before the turn of the century movie stars, or indeed, any type of performer, may literally be even more pre-processed than the current crop of charicatured symbols emanating from the film, television and music industries.

With the advent of sophisticated VR systems, probably at the beginning of the next century, the propagation and wide acceptance of, perhaps even demand for, synthetic actors over 'real' ones will be paralleled on personal/home VR systems.

The spread and wide acceptance of synthetic actors compared to real ones may become driven not so much by what mass audiences wish to watch (passively) on film and video as by what they will become devoted to as a result of interacting with personal/home VR systems. In a 'media world' dominated by male fantasies, the enlightening promise of VR, like that of television before it, seems set to become subservient to financial returns and corporate market forces. For those in society who at one time or another are unable to 'test reality', the combination of seamless image synthesis and the immersive experiences of VR may lead to behaviour that will, by comparison, make the current debate about the link between criminality and explicit violence on the television and cinema seem like a trivial linguistic parlour game.

But how difficult is it to combine the real and the imaginary; the photographic and the synthetic? This project is designed to encourage you to explore the technical and aesthetic issues involved in synthesing imagery.

Readings   Nan–o–sex and Virtual Seduction
Siggraph Panel Session
Computer Graphics Proceedings of SIGGRAPH 93

Computer Graphics in Visual Effects – Course Notes Siggraph '93
Charles Gibson

Computer Graphics in Visual Effects, Cost Effective Special Effects
Section 4 – Course Notes Siggraph '93
Ricard Hollander

A Once and Future War
Jody Duncan
Cineflex issue 47, August 1991

An Integrated Control View of Synthetic Actors
D. Boisvert, N. Magnenat-Thalmann and D. Thalmann
New Advances in Computer Graphics
Proceedings of CG International 1989

Project 3   Three Dimensional Icons for a Graphical User Interface

Background

All modern interfaces use small pictograms called icons to help users interact with a computer system. The icons, generally much smaller than 100 pixels square, are used

- as a system of static **signage** to enable a users to orientate themselves within an imaginary electronic space, somewhat like highway signs and road markings,
- to dynamically provide a response, or feed-back, to support the users sense of **interaction**.

An icon as a *sign* can represent

- an **object**, such as a "recycler" for destroying files, fig 1,
- a **concept** or an abstraction – a body of knowledge such as an historical data base stored on a CD, fig 2,
- an **action**, for example a check box that controls colour printing, fig 3.
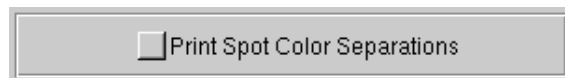
fig. 4
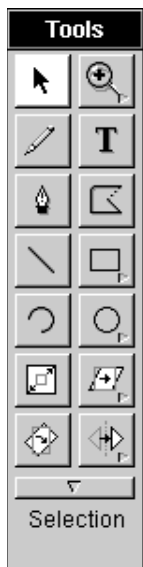
fig. 1

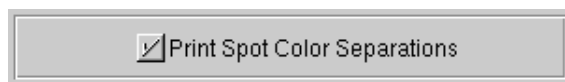fig. 2

fig. 3 check box OFF
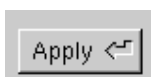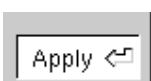
fig. 3 check box ON

fig. 5 button UP

fig. 6 button DOWN

Icons are used extensively to represent storage devices, hard disks, diskettes and CD's, the directory structure of the file system and the files themselves. Often dozens of icons are used within a single application as 'covers' for buttons or the cells in tool palettes, fig 4. Such buttons and cells often have two graphics associated with them, one for the button's dormant state and the other to indicate to the user that some change has or is about to occur. For example, a button might normally have an icon that represents it's "UP" condition which is subsequently replaced by another icon to give the illusion of the button having a "DOWN" condition when a user presses it, figs 5 and 6.

The majority of interfaces used at present are two dimensional even though, as can be seen by the figures on the previous page, there is often an implied depth. Interactive information systems, such as desktop computers, multimedia CD's, interactive television, information kiosks and "games" consoles will move away from using solely two dimensional elements in their user interface, such as 'flat' icons and buttons. Instead, three dimensional representations combined with various transformations (animations) will enliven, what many people now see as the bland uniformity that exists from one interface design to another.

The best designs will undoubtedly be those that use three dimensional graphics in a way that improves communication rather than those that have gratuitously applied 3D techniques. For example, an icon that represents some information on a hard disk might, when selected by a user, undergo a 3D transformation that conveys an animated "explanation" of what it represents, rather than, as is now the case, merely undergoing a colour inversion to show it has been selected. How often, when faced with the task of learning a new software package, have you wondered about the function a particular item in a tool palette? Using animated graphics, the icons in a palette might "act-out" their function, or divide or otherwise "decompose" into several more easily understandable 'sub-icons'.
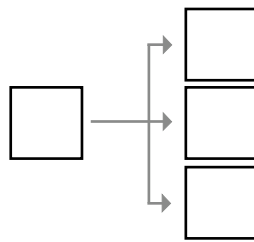
Sub-division is one way in which an icon could decompose to provide more information to a user. For example, an icon representing the category "New Zealand" on a CD dealing with "Tourism in the Pacific" might divide into two or three sub icons representing, "sights and sounds", "adventure" and "relaxation", when selected by a user. The area of interface design posed several problems. For example, can 3D graphics help in sustaining an illusion that users are entering an informational space and not just browsing a computerised version of a travel agents notice board?

If the icon, "New Zealand", was three dimensional it's way of decomposing might be to "unwrap" as if it were a box being opened out flat. But should each of the new sub-icons be three dimensional? If so then what is to prevent the user becoming hopelessly lost? What conventions should be adopted for the icons – how many levels of decomposition can the user be expected to keep track of. How can a user be certain that an icon has no further sub-levels and what visual cues should be provided to enable a user back-track to a previous level? After all the icons only represent the information, they are not the information themselves.
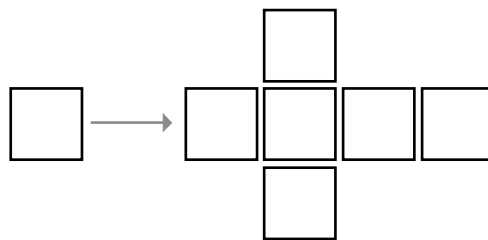
Design Brief

This third and final project of the course is concerned with employing 3D computer graphics animation to some element of a real or imaginary graphical user interface. You are to design and animate a 3D icon(s). However, because the project is primarily concerned with 3D computer animation and not the issue of interface design as such, you are NOT required to design an interface in which the icon(s) could be used. However, you are strongly advised to be mindfull of the questions raised in the previous paragraph.

There are no restrictions on the colour depth of your graphics ie. you may break the artificial colour barriers that are normally imposed on this work, and the content of what your icon, or system of icons represent is entirely your decision. However, the following restrictions apply,

- maximum screen space is 300x300 pixels – look upon this as your "stage" and the icons as the "actors",
- maximum duration of any single animated segment such as a transformation, a transition or other effect is 50 frames.

Submissions,

- at least ONE "demo" animation of your 3D icon/button in the form of a QuickTime movie,
- all texture, transparency, displacement maps and rib files used in the production of the animation(s), and finally
- an A3 "concept board", mounted or unmounted, that shows the development of your ideas.

More specific details about the time and place of the submission will be made available later.