

Rayshade-M Manual

Contents

- Preface3**
- 1 Introduction.....4**
 - 1.1 A Simple Example.....4
- 2 Running Rayshade.....6**
 - 2.1 The Input File.....6
 - 2.1.1 Generating a Script File6
 - 2.2 Images6
 - 2.3 Antialiasing.....7
 - 2.4 The Ray Tree.....7
 - 2.5 Selecting Antialiasing and Tree Depth.....7
- 3 The Object Editor.....9**
 - 3.1 The Wireframe View.....9
 - 3.2 The Crosshair.....9
 - 3.3 The Object List.....9
 - 3.4 View Filters10
 - 3.5 Creating New Lights and Objects.....10
 - 3.6 Modifying an object.....11
- 4 Specifying a View.....12**
 - 4.1 Camera Position12
 - 4.2 Field of View.....13
 - 4.3 Depth of field.....13
 - 4.4 Screen Options.....13
- 5 Light Sources.....15**
 - 5.1 Light Source Types.....15
 - 5.2 Shadows.....16
- 6 Object Definition17**
 - 6.1 The World Object.....17
 - 6.2 Primitives.....17
 - 6.3 Aggregates.....18
 - 6.4 CSG in Rayshade-M.....19
- 7 Surfaces and Atmospheric Effects20**
 - 7.1 Surface description.....20
 - 7.2 Atmospheric Effects.....20
 - Transformations.....22
- 8 Texturing.....23**
 - 9.1 Texturing functions.....24
 - 9.2 Mapping Functions.....25
- A Script Language Quick Reference26**
- B Animation.....30**
- Bibliography.....31**

Preface

A public domain ray tracer written in C is available. Implement this on a Mac IIx and experiment with a variety of acceleration techniques as well as a system to construct interesting scenes for ray tracing.

Rayshade is a program for creating ray-traced images. It reads a description of a scene to be rendered and produces a colour image corresponding to the description. *Rayshade* was designed to make it easy to create nice pictures. It was also meant to be flexible, easy to modify, and relatively fast.

Rayshade-M is a specially adapted version of *rayshade* which takes advantage of the capabilities that the Macintosh range of computers offers. This enhanced version allows the user to construct and render scenes with greater ease through visual feedback from a built-in object editor. This makes ray tracing more accessible and less time consuming. Additionally *rayshade-M* can be used in conjunction with other versions since it reads and writes compatible script files.

Rayshade-M requires a Macintosh fitted with a maths co-processor, 32-bit colour QuickDraw, a 13-inch (640x480) monitor and preferably 8 Megabytes of free RAM. Images are stored internally at 24-bit colours. If the monitor is not capable of supporting this many colours, then a best match picture is displayed. Images are saved in PICT format which enables rendered scenes to be used in other Macintosh applications such as word-processors, art-programs etc.

Chapter 1

Introduction

This manual attempts to cover all of the features found within *rayshade-M* as opposed to *rayshade*. A fair level of technical knowledge is assumed, both of the Macintosh operating system, and of the features and techniques commonly associated with ray-tracing. This manual places much more emphasis on the user-interface of *rayshade-M* than the script language, since to all intents and purposes, the script has been made invisible to the user.

For further information of the syntax of the *rayshade* language, the user may consult the quick reference appendix in this guide or the manual written by Craig Kolb which accompanies the original version of *rayshade*.

1.1 A Simple Example

Rayshade provides default values for the camera, screen and lighting, so it is easy to render a simple example.

Run the application.

Rayshade is a 24-bit ray tracer so make sure your monitor is set to display the highest number of colours possible.

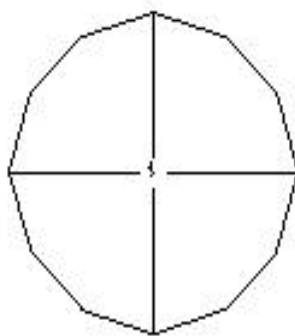
Select New from the menu

The editor window will be opened. This contains three view areas representing the top, front and side views of the scene. At the moment these views will be blank apart from a cross-hair, a letter t and a letter o. The letters stand for observer and target, and have default values which do not need to be changed for this example.

Click on the Create button.

This will open the object/light creation dialogue box.

Select sphere from the pop-up menu and then click on the Create Object button. A new sphere with radius 1 will be drawn, centred on the cross-hair.



Select Preview from the menu.

Preview is the fastest way to display a rendered picture, since it does not draw shadows or attempt to antialias the picture.

You should have an image similar to this:



Chapter 2

Running Rayshade

The time required to render an image is not constant. It can take anywhere between seconds and weeks of rendering, and this depends on the speed of the Macintosh, the complexity of scene and how 'good' you want the final image to be.

Usually it best to carry out a series of test renderings on a scene, progressively refining the settings, before rendering the final image.

This chapter describes the basic operations of *Rayshade-M* and some of the options that control how a scene can be rendered.

2.1 The Input File

The scene description read by *rayshade-M* consist of a number of keywords, each followed by a set of arguments. These keywords can be thought of as commands that direct *rayshade-M* to do various things, such as create objects, set the eye's position, and change the objects position.

The user-interface of *rayshade-M* means that the inner workings of the scene description language have been hidden from the user. It is still possible to view or even change the contents of such a script using a standard Macintosh text editor, so long it is capable of saving 'TEXT' type files.

Note: The original *rayshade* automatically runs the C preprocessor, so that directives such as `#define`, `#include` and `#ifdef` are allowed in the scripts. Since this preprocessor is not present in *rayshade-M*, all of these must be removed before the script is parsed.

2.1.1 Generating a Script File

Rayshade-M is capable of generating script files based on the scene which the user is currently editing. This script is compatible with all versions of *rayshade*, which means it is possible to use the Macintosh editor as a scene development tool and then run the script constructed from the scene, on a more powerful machine.

To save a scene, select *Save* from the menu, a file selector will allow you to select the name of the output script before it is generated.

There are limitations to the current version of the script generator: all animation data, comments, and user-defined variables which are contained in the input script will not be generated into the output script. All named instances, surface definitions, transformations and textures are saved however.

Typically, the machine-generated script will be slightly larger than the input script, since all floating point numbers are generated to 5 decimal places, whether or not this level of precision is required.

2.2 Images

The final result of running a script through *rayshade-M* is an image file. In the original *rayshade* these images were saved in either the *Utah Raster RLE* format or in the generic *mtv* format. *Rayshade-M* however stores its images in PICT format. This means in order for images

between the two versions of Rayshade to be exchanged, a graphics tool such as *Adobe PhotoShop* must be used to convert between the different image formats.

When an image has been rendered, the user can save this by selecting the **Save Picture** menu option. If the scene has animation data within it, the user will be asked to set the frame range before rendering. Each frame will be automatically saved once it is completed.

2.3 Antialiasing

If an image is not sampled at the proper rate, aliasing will occur. This is usually apparent as 'jaggies' within the image. *Rayshade* has several methods of countering this problem.

The first is jittering, which adds a random offset to the position of each ray as it is fired into the scene. This has the effect of reducing stepping, however it does introduce noise which can be unwelcome.

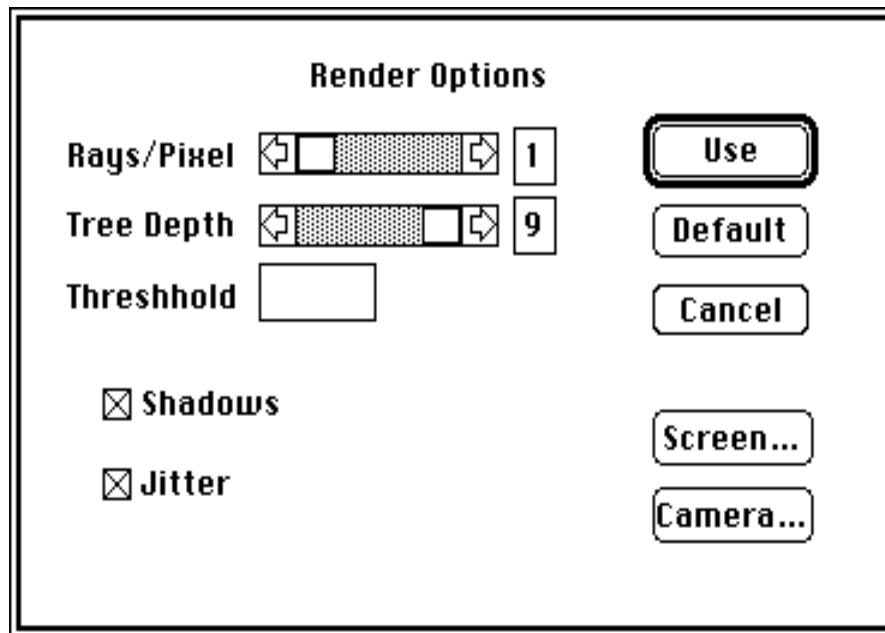
The second method is adaptive sampling. This begins by sampling each pixel on the current scanline once. For each pixel on the scanline, the contrast between it and its four immediate neighbours is computed. If this contrast is greater than a user-specified maximum in any colour channel, the pixel and its neighbours are all supersampled by firing an additional $\text{numsamples}^2 - 1$ rays through those pixels that have not already been super sampled. This process is repeated for the current scanline until a pass is made without a pixel being supersampled.

2.4 The Ray Tree

When ray tracing a scene, reflected or transmitted rays may strike other reflective or transparent objects. Further reflected or transmitted rays will be spawned and so on. This hierarchy of rays is known as a ray tree. Care must be taken to control the depth of this tree: If it is allowed to grow too deeply, the program may spend a great deal of time, computing rays that have an insignificant effect on the final picture; if the tree is not allowed to grow far enough, degradation of image quality may be result, e.g. Transparent objects appear solid or dark.

2.5 Selecting Antialiasing and Tree Depth

Antialiasing and tree-depth are set from the **Render Options...** menu option. This will the dialogue box below:



By default the antialiasing is set at 3^2 jittered samples. The rays per pixel can range anywhere between 1 to 8^2 . The maximum level of super-sampling takes considerably longer than the shorter level and should only be used for the final image.

The tree depth can be set between 0 and 9 levels of recursion. At recursion level 0, only a primary ray is fired, and no further rays are spawned. This leads to transparent objects appearing solid and no reflections or shadows. The highest tree-depth is 9, which can be a lot slower in scenes with many reflective or transparent surfaces.

The threshold is the minimum intensity amount that a ray must contribute to the image if it is to be spawned further. Normally if a ray does not contribute more than 1% to the intensity of the final image, it is not worth spawning it any further.

The **Preview Picture** menu option has preset rendering options 1 ray per pixel, 0 tree-depth, no jittering, and no shadowing. Preview mode is useful to check the "correctness" of the scene before proceeding with the more expensive normal rendering.

Chapter 3

The Object Editor

Rayshade-M is unique in the *rayshade* family in that it is possible to create a scene from within the application. This means it is very easy to design scene without any knowledge of how the *rayshade* script language works. Specific information on object, surface, texture and light editing can be found in their own chapters. This chapter is used to describe the how the editor works.

3.1 The Wireframe View

The object editor provides a visual representation of what a scene will look like when it is rendered. This is done via three panels which give a front, side and top view of the scene. Each object is represented by a wireframe equivalent which means that each view can be drawn fast.

The wireframe representation of an object uses surface information to determine its colour, and uses the objects transformations to determine its position and shape. For instance a sphere which has been affected by a (2,0.5,0.5) scaling, will be represented correctly, i.e. stretched.

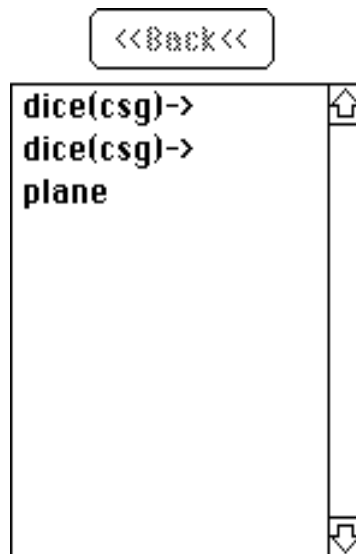
Also shown on the view are the camera and light positions. The observer is represented by a letter *o*, and facing at the target *t*. Each light source is shown as a letter within a circle. The letter is the type of the light source. e.g. E for extended.

It is possible to zoom in and out of a scene using the zoom buttons situated in the spare quarter of the window. To zoom in, click on the + button. To zoom out, click on the - button.

3.2 The Crosshair

The crosshair is used to determine where new objects are placed. Repositioning the crosshair simply involves clicking on the desired destination. The coordinates of the crosshair are shown in the bottom left of the window. When a new object is created, the centre of the object will appear at the crosshair, making it possible to place objects with great accuracy. Additionally when you wish to make an object transformation, the crosshair positions is used to insert default values into the rotation axis fields.

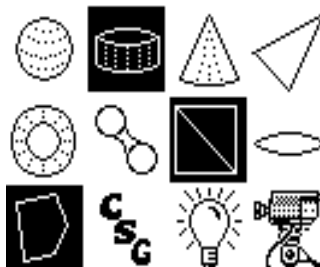
3.3 The Object List



The object list contains a textual representation of the World. It works in a similar way as a disk filing system: consider aggregates the same as folders, and primitives the same as files. By doubling clicking on aggregates, you will go down to next level in the hierarchy. Double clicking on primitives will allow you to modify their characteristics.

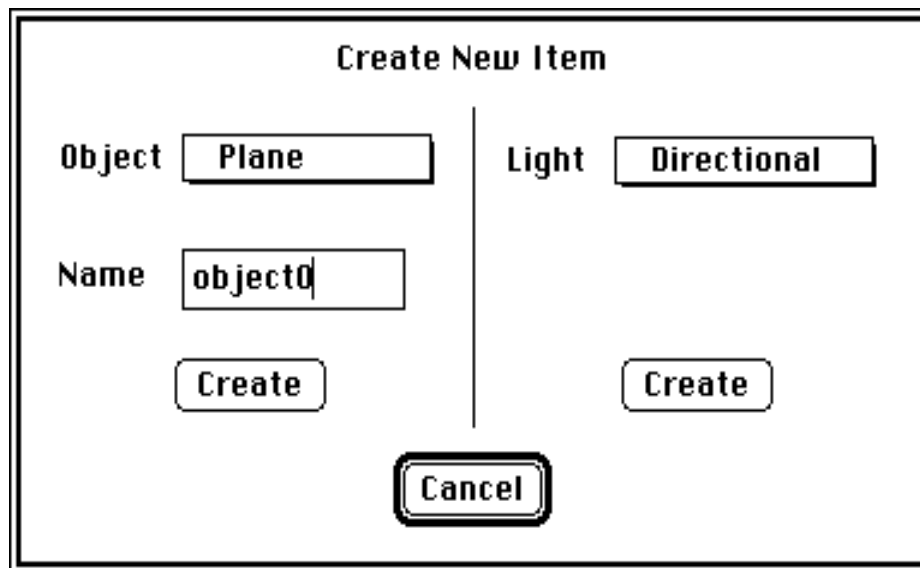
To return to an object's parent, click on the <<**Back**<< button. Since the *World* object has no parent, the button is disabled at the top tree level.

3.4 View Filters



In a complicated or cluttered scene it is often too confusing to be able pick out the shape of particular objects. Filters allow a way of doing this by masking out certain object types or other information. The filters are Boolean switches, so that clicking twice on an icon will return it to its original state. Activated filters are highlighted.

3.5 Creating New Lights and Objects



The 'Create New Item' dialog box is divided into two columns by a vertical line. The left column contains an 'Object' label with a text box containing 'Plane', a 'Name' label with a text box containing 'object0', and a 'Create' button. The right column contains a 'Light' label with a text box containing 'Directional' and a 'Create' button. A 'Cancel' button is centered at the bottom of the dialog.

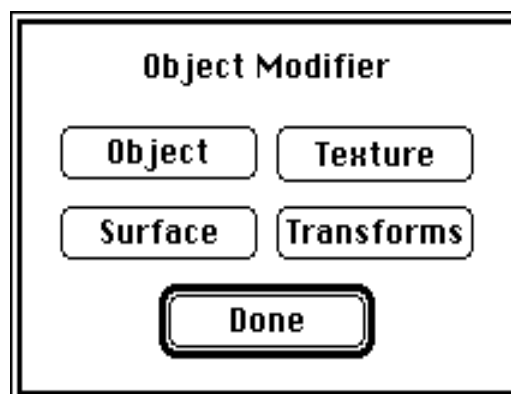
To create a light object, move the crosshair to the desired position for the new item. Click on the **Create** button. The **Create New Item** dialogue will be displayed.

If you are creating an object, select the name and type of object, and then click on on the object creation button. A new object will appear at the current level of the object list, and it will also be drawn in wire-frame. Since the dimensions, surface and texture of this new object may not be those desired, it may be necessary to modify the object by double clicking on it.

If you are creating a light, select the type and click on the light creation button. A new light of the desired type will appear at the crosshair. The values associated with the light can be changed by going to the light editor.

Note: Light and object types which have been greyed out means they are currently not supported by the editor. This does not stop the user from adding them manually to the scene script is they wish though.

3.6 Modifying an object



The 'Object Modifier' dialog box contains four buttons arranged in a 2x2 grid: 'Object', 'Texture', 'Surface', and 'Transforms'. A 'Done' button is centered at the bottom of the dialog.

Each object has information which can be changed by either double-clicking on the object's name, or clicking on the **Modify** button with the object selected. Transformations, textures and surfaces properties are all hierarchical, so if the object being edited is an aggregate, all changes will affect objects that it contains.

The user can change an objects properties from this menu, and when finished select **Done**.

Chapter 4


Specifying a View

Rayshade uses a camera model to describe the geometric relationship between the objects to be rendered and the image that is produced. This relationship describes a perspective projection from world space onto the image plane.

The geometry of the perspective projection may be thought of as an infinite pyramid, known as the viewing frustum. The apex of the frustum is defined by the camera's position, and the main axis by the "look" vector. The four sides of the pyramid are differentiated by their relationship to a reference "up" vector from the camera's position.

The image ultimately produced by *rayshade* may then be thought of as a projection of the objects to the eye onto a rectangular screen formed by the intersection of the pyramid with the plane orthogonal to the pyramid's axis. The overall shape of the frustum (the lengths of the top and bottom sides compared to left and right) is described by the horizontal and vertical fields of view.

4.1 Camera Position

Camera Options

Observer		Target		Up Vector	
X	<input type="text" value="0.000"/>	X	<input type="text" value="0.000"/>	X	<input type="text" value="0.000"/>
Y	<input type="text" value="20.000"/>	Y	<input type="text" value="0.000"/>	Y	<input type="text" value="0.000"/>
Z	<input type="text" value="8.000"/>	Z	<input type="text" value="3.500"/>	Z	<input type="text" value="1.000"/>

Aperture

Focal distance

Field of Vision°

Cancel

OK

The three basic camera properties are its position, the direction in which it is pointing, and its orientation. All of these values can be set from the Camera Options dialogue. The Observer position is where the camera "eye" is located. This is represented on the editor by the letter *o*. The target position is the location the camera is looking at. This represented on the editor by a letter *t*. The upvector is orientation of the camera.

The default values are designed to provide a reasonable view of a sphere of radius 2 located at the origin.

4.2 Field of View

Another important choice to be made is that of the field of view of the camera. the size of this field describes the angles between the left and right sides and the top and bottom sides of the frustrum.

By default the horizontal field of view is 45 degrees. An option for the vertical field of view is not provided in the editor since it is not necessary.

4.3 Depth of field

It is usual to use a "pinhole" camera to render a scene. In this mode the camera's aperture is a single point and all light rays are focused on the image plane.

Alternatively, one may widen the aperture in order to simulate depth of field. In this case, rays are cast from various places on the aperture disk towards a point whose distance from the camera is equal to the focus distance. Objects that lay in the focal plane will be in sharp focus. The farther an object is from the image plane, the more out-of-focus it will appear to be. A wider aperture will lead to greater blurring of objects that do not lay in the focal plane. When using a non-zero aperture radius, it is best to use jittered sampling in order to reduce aliasing.

4.4 Screen Options

Screen Options

Resolution

☒ 50 X 50

☐ 100 X 100

☐ 200 X 200

☐ Custom

50

X

50

Magnification

☒ 1x1

☐ 2x2

☐ 4x4

☐ 8x8

☐ 16x16

☐ 32x32

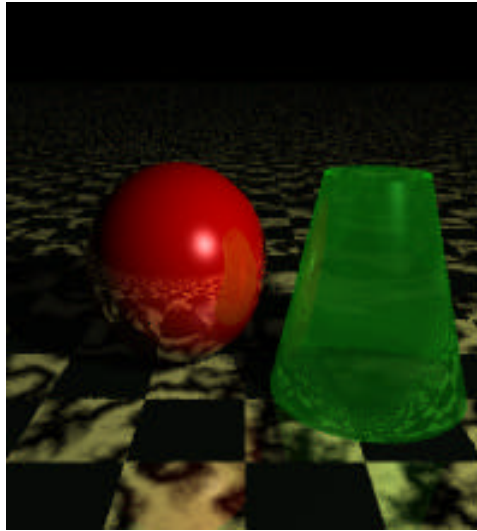
Cancel

Ok

This is where the size and magnification of the image are selected. Selecting larger resolutions will mean that *rayshade-M* will take longer to produce an image. the magnification option is provided mainly as a way of being able to see very small images. For instance if you want to render a 50x50 image at 200x200 resolution then it can be achieved by setting the magnification level to 4x4. The pixels will appear 'chunky' however this still allows a general idea of the final version would have looked like. Compare the, magnified version in (a) with the final version in (b).



(a)



(b)

Chapter 5

Light Sources

The lighting in a scene is determined by the number, type, and nature of the light sources defined in the input file. Available light sources range from simple directional sources to more realistic but computationally costly quadrilateral area light sources. Typically, you will want to use point or directional light sources while developing images. When final renderings are made, these simple light sources may be replaced by the more complex ones.

No matter what type of light source you use, you will need to specify its intensity. Intensity is defined by either a red-green-blue triple indicating the colour of the light source, or a single value that is interpreted as the intensity of a "white" light. In current versions of *rayshade*, the intensity of the light does not decrease as one moves farther from it.

If you do not define a light source, *rayshade* will create a directional light source of intensity 1.0 defined by the vector (1.,-1.,1.). This default light source is designed to work well when default viewing parameters and surface values are being used.

You may define any number of light sources, but keep in mind that it will require more time to render images that include many light sources. It should also be noted that the light sources themselves will not appear in the image, even if they are placed in the frame.

5.1 Light Source Types

The amount of ambient light present in a scene is controlled by a pseudo light source of type ambient.

There is only one ambient light source; its default intensity is 1,1,1. If more than one ambient light is defined, only the last instance is used. A surface's ambient colour is multiplied by the intensity of the ambient source to give the total ambient light reflected from the surface.

Directional are described by a direction alone, and are useful for modelling light sources that are effectively infinitely far away from the objects they illuminate.


Point sources are defined as a single point in space. They produce shadows with sharp edges and are a good replacement for extended and other computationally expensive light sources.

Spotlights are useful for creating dramatic localized lighting effects. They are defined by their position, the direction in which they are pointing, and the width of beam of light they produce.

Extended sources are meant to model spherical light sources. Unlike point sources, extended sources actually possess a radius, and as such are capable of producing shadows with fuzzy edges (penumbra). If you do not specifically desire penumbras in your image, use a point source instead.

The shadows cast by extended sources are modelled by taking samples of the source at different locations on its surface. When a source is partially hidden from a given point in space, that point is in partial shadow with respect to the extended light source, and the sampling process is usually able to determine this fact.

Quadrilateral light sources are computationally more expensive than extended light sources, but are more flexible and produce more realistic results. This is due to the fact that an area source is approximated by a number of point sources whose positions are jittered to reduce aliasing. Because each of these point sources has shading calculations performed individually, area sources may be placed relatively close to the objects it illuminates, and a reasonable image will result.

Each light source type is represented by its first letter surrounded by a circle, for instance an extended light source is represented by the symbol .

5.2 Shadows

Shadow determination is made by tracing rays from the point of intersection to each light source. These "shadow feeler" rays can add substantially to overall rendering time. This is especially true if extended or area light sources are used.

Shadow determination can be toggled on and off from the Render Options dialogue window.

Chapter 6

Object Definition

Rayshade-M is capable of rendering all of the object types found in the Unix counterpart. There are restrictions in *rayshade-M* which have been taken in the interests of saving time, but still preserving the overall flavour of the program. Please note that in the most part these restrictions usually only prevent the user from creating/manipulating the objects from within the editor, but this does not stop them from being added to the script by hand.

Each object primitive can be altered from its own dialogue box which gives primitive specific information. Additionally the user can also define the texture and surface of the object from here as well.

6.1 The World Object

The scene is all contained within one object called the World. This is an aggregate list type, and is represented on-screen by the hierarchical world-list.

The current level is displayed as a list on the screen. Doubling clicking on a composite object such as a list or csg will move down one level in the tree. Clicking on the <<**Back**<< button will move one level up the tree.

To edit a primitive object, just double click on it. If the object is an instance of defined type, then changing this object will change ALL instances of this object elsewhere.

6.2 Primitives



Blob

A blob is a superimposed exponential density distribution, constructed from sub-objects called meta-balls. Each meta-ball is defined by its position, radius and strength. The blob overall is also governed by a threshold value. *Rayshade-M* allows the user to create blobs which may contain up to thirty meta-balls.



Box

A box is an axis aligned volume contained between two opposite corner points.



Sphere

A sphere is defined by a position and a radius. An ellipsoid can be created from a sphere by scaling the object in the x, y and z coordinates.



Torus

A torus is defined by an outer and inner radius, a centre position and an up vector.



Triangle

There are two types of triangle:

Flat Triangle - Described by 3 vertices

Phong-shaded Triangle - Described by 3 vertices and the normals at each vertex. Phong shaded triangles interpolate the vertex normals over the surface, resulting in less faceting.



Polygon

A polygon may consist of one or more vertices which are described in an anti-clockwise direction. *Rayshade-M* set a limit of a 30 sided polygon, however it is unlikely that this will pose a problem.



Height field

A height field is a grid of altitude points stored within a file. The surface of the height field is constructed from joining points to form triangles. No editing or creation of heightfields is allowed in the current version of *rayshade-M*, since they are a number of problems:

- a) The height field file stores data in the native floating point format for the host computer. This means that heightfields are not portable between computers with different floating point representations.
- b) Writing code for the manipulation of height fields would have been time-consuming and would have distracted the focus of programming from more important tasks.



Plane

A plane is described by a point on the surface and a normal to the surface.



Cylinder

Cylinders are defined by a radius and a top and a bottom.

Note: Cylinders are not capped. A capped cylinder can be created by covering each end with a disc.



Cone

A cone is defined by a top and bottom radius and a top and bottom position. The inclusion of a top radius enables cones to be truncated. The method with which cones are stored internally means that the end with the largest radius is regarded as the base. This means that sometimes the cone dialogue editor may switch the fields for the base and apex.

Note: It is not possible to have a base and apex radius that are equal since the cone will never converge. If the two radii need to be equal, use the cylinder primitive instead.



Disc

A disc is defined by a centre, a radius and a surface normal. Discs can be used to 'cap' cylinders. By using scaling transformations it is possible to create ellipsoid discs.

6.3 Aggregates

Aggregate objects are constructed from sub-objects. Double-clicking on an aggregate in the object list will reveal its contents.

Aggregates allow the user to group objects together in meaningful ways, which often makes the task of scene construction much simpler. For instance, if a group of primitives need to be moved together, it is better to have them contained within a list and then transform the list.

Instances

Object instances are used to reference predefined objects. For example it is possible to define an object called *Chair* and have more than instance of *Chair* positioned around another object called *Table*. When instances are deleted, the actual referenced object still remains intact.

List

A list aggregate contains all the objects held within it. Bounded and unbounded objects are dealt with separately, so objects such as planes which are unbounded will always appear last in the list.

Grid

A grid is an array of size X by Y by Z which contains an object for each position within the array. When the user creates a grid, it will be filled by dummy objects. To insert a object in its place, click on the create button. After an object has been created you will be asked where in the grid you would like it to be placed.

6.4 CSG in Rayshade-M

Constructive Solid Geometry allows complex objects to be constructed from primitives using simple Boolean operations. *Rayshade* is capable of performing union, difference or intersection between two different sub-objects. These sub-objects can be primitives or aggregates, so it is possible to nest csgs within each other. *Rayshade-M* has no support for the creation or manipulation of csg types, however it will still parse them, and display their contents. No attempt is made to compute the resultant object from a csg operation, e.g. a csg containing a box and a cylinder will be drawn as a box and cylinder.

Chapter 7

Surfaces and Atmospheric Effects

Surfaces are used to control the interaction between light sources and objects. A surface specification consists of information about how the light interacts with both the exterior and interior of the object. For non-closed objects such as polygons, the "interior" of an object is the "other side" of the object's surface relative to the origin of the ray.

Rayshade usually ensures that a primitive's surface normal is pointing towards the origin of the incident ray when performing shading calculations. Exceptions to this rule are transparent primitives, for which *rayshade* uses the direction of the surface normal to determine if the incident ray is entering or exiting the object. All non-transparent primitives will, in-effect, be double-sided.

7.1 Surface description

Surface Properties

Ambience	<div></div>	Specular Power	<div>25.00</div>
Diffuse	<div></div>	Attenuation	<div>1.000</div>
Specular	<div></div>	Refractive Index	<div>1.150</div>
Diffuse Transmission	<div></div>	Reflectivity	<div>1.000</div>
Specular Transmission	<div></div>	Transmittance	<div>0.300</div>
		Translucency	<div>0.000</div>
<div><input type="checkbox"/> Shadows</div>		<div>Use</div>	
		<div>Cancel</div>	

Clicking on the surface button within the object modification dialogue will bring up window which allows the surface properties of the object to be changed. Colours are shown to represent the values stored for ambience, diffuse, specularity, body and translucency. When a new object is created all of these properties are given default values.

This editor additionally allows the user to change the reflectivity, specular power, transparency, extinction coefficient and refractive index.

The ambience value associated with an object's surface is used colour it when it is displayed in wire-frame.

7.2 Atmospheric Effects

Rayshade allows various atmospheric effects to be applied to the scene.

Atmosphere

This specifies the refractive index for the atmosphere in which the scene is defined.

Fog

Is specified by its colour and its thinness. As the ray travels further from the view plane its colour becomes that specified.

Mist

Mist is similar to fog, however it is also affected by altitude. Mist is given a colour and thinness, and additionally A base altitude and scale.

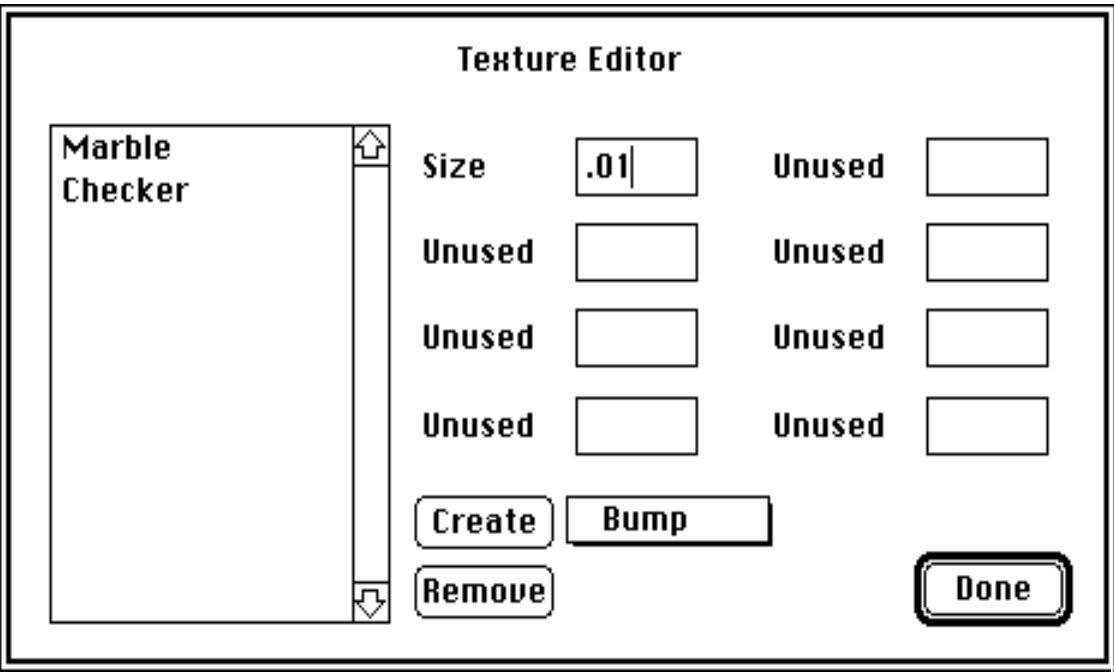
Chapter 8

Transformations

It is possible to offset, rotate, or scale an object. This makes it possible for sphere to be turned into an ellipsoid, or two instances of the same object to be in different positions. Transforms work in a hierarchical fashion, starting with the node objects and working backwards. Therefore if a list is 'squashed', all of its children will appear 'squashed' also. Transformations are shown by *rayshade-M*, therefore it is possible to use transformed instances within a scene.

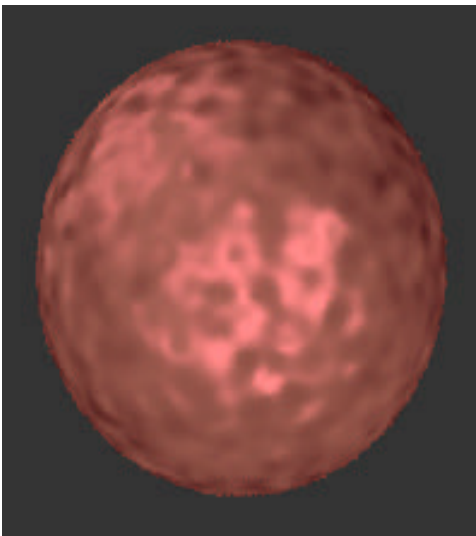
Chapter 9

Texturing



An object may have none or more textures. These can be altered from within the texture dialogue box. Textures are affected by scaling transformations and each texture for an object is applied in turn, these means it is possible to have an object affected by more than one texture, say wood and cloud for instance.

Note: Image mapping is not yet supported by the editor or by the script generator, however it still possible to use image mapping if it is done from the input script. Please note that *rayshade-M* still uses a raw format for images and not PICT or the RLE format found in the *Utah Raster Graphics Toolkit*.



An example of texture mapping

9.1 Texturing functions

Blotch *Blendfactor* <*Surface*>

This produces a blotchy looking surface, which is based on mixing the objects original surface with a new surface. The *blendfactor* of this mix, determines which surface predominates the object. A blend-factor of 0 will give a 50-50 mix, higher values give more of the default surface.

Bump *scale*

This applies a random bump map to the surface of an object. The point of intersection is passed to a noise function which returns a normalized vector which is then added to the normal vector at the point of intersection. Bump mapping can be scaled to alter the 'height' of each bump.

Note: that Bump is only a texture mapping. i.e. if you were to look at a plane using bump-mapping from the side it would still be a flat object.

Checker <*Surface*>

This applies a checkerboard texture to an object. A second surface is defined in the checker definition, and this is the one that is applied when a point of intersection falls within an 'odd' cube of 3-dimensional space.

Cloud *scale H lamda octaves cthresh lthresh tscale*

FBm *offset scale H lambda octaves thresh [colourmap]*

FBm stands for fractional Brownian motion, and it is used to scale the diffuse and ambient components of an objects surface. The function has several parameters which control its results.

<i>Scale</i>	Is used to scale the value returned by the fBm function.
<i>Offset</i>	Is the minimum value that can be returned by the fBm function
<i>H</i>	Is the Holder exponent (0.5 works quite well)
<i>Lambda</i>	Is used to control lacunarity, specifying the frequency differences between successive samples of the fBm function. (2.0 will suffice)
<i>Octaves</i>	Is the number of octaves (samples) to take of the fBm basis function, between 5 and 7 usually works well.
<i>Thresh</i>	Is used to specify the lower bound on the fBm output/

FBmBump

This is similar to fBm but is applied as a bump map, rather than affecting the colours of an object.

Gloss *glossiness*

Gives reflective surfaces a glossy appearance. A value of 1 results in perfect mirror-like reflections, while a value of 0 results in extremely fuzzy reflections. For best results, jittered sampling should be used to render scenes that make use of this texture.

Marble

This creates a marbled texture. The function produces a veined marble intensity map which is applied to the surface of the object. An optional colour-map can be provided which returns colours instead of intensity values, making it possible to have unusually coloured marble.

Sky *scale H lambda octaves cthresh lthresh*

This is similar to the *fBm* function. Rather than modifying the colour of a surface, this texture modulates its transparency. The parameter *cthresh* is the value of the *fBm* function above which is totally opaque. Below *lthresh*, the surface is totally transparent.

Stripe *<Surface> size bump <Mapping>*

This applies a "raised" stripe pattern to the surface. The surface properties used to colour the stripe are those of the given surface. The width of the strip, as compared to the unit interval is given by *size*. The magnitude of *bump* controls the extent to which the bump appears to be displace from the rest of the surface. If negative, it will appear to stand out of the surface.

Wood

The wood texture produces a brown/green wood appearance to an object. It is normally desirable to scale the texture since the feature size is approximately 0.01 of a unit.

Image

Images can be used to modify the characteristics of a surface. *Rayshade-M* has no editor support for image mapping, and all instances of image texture will be not be included if a script is generated. It is still possible to use images but this must be specified from the script.

9.2 Mapping Functions

Mapping functions allow the application of two-dimensional textures in a variety of ways by transforming a three-dimensional point on a surface into a u-v pair which is used to deduce the texture value that should be applied to that point.

map *uv*

This uses the *uv* (inverse mapping) method associated with the object that was intersected in order to map from 3D to determine texturing coordinates.

map linear [*origin vaxis uaxis*]

This uses a linear mapping method. A 2D texture is transformed so that its *u* axis is given by *uaxis* and its *v* axis by *vaxis*. The texture is projected along the vector defined by the cross product of the *u* and *v* axes, with the (0,0) in texture space mapped to origin.

map cylindrical [*origin vaxis uaxis*]

This uses a cylindrical mapping method. The point of intersection is projected onto an imaginary cylinder, and the location of the projected point is used to determine the texture coordinates. If given, *origin* and *vaxis* define the cylinder's axis, and *uaxis* defines where *u*=0 is located.

map spherical [*origin vaxis uaxis*]

This uses a spherical mapping method. The intersection point is projected onto an imaginary sphere, and the location of the projected point is used to determine the texturing coordinates in a manner identical to that used in the inverse mapping method for the sphere primitive. If given, the centre of the projection is *origin*, *vaxis* defines the sphere axis, and the point where the non-parallel *uaxis* intersects the sphere defines where *u* = 0 is located.

Appendix A

Script Language Quick Reference

File:

<Item> [<Item> ...]

Item:

<Viewing>
<Light>
<Atmosphere>
<RenderOption>
<ObjItem>
<Definition>

ObjItem:

<SurfDef>
<ApplySurf>
<Instance>
<ObjDef>

Viewing:

eyep Xpos Ypos Zpos	/* Eye position (0 -10 0) */
lookp Xpos Ypos Zpos	/* Look position (0 0 0) */
up Xup Yup Zup	/* "up" vector (0 0 1) */
fov Hfov [Vfov]	/* FOV in degrees (horizontal=45) */
aperture Width	/* Aperture width (0) */
focaldist Distance	/* focal distance (eyep - lookp) */
shutter Speed	/* Shutter speed (0 --> no blur) */
framelength Length	/* Length of a singelf frame (1) */
screen Xsize Ysize	/* Screen size */
window Xmin Ymin Xmax Ymax	/* Window (0 0 xsize-1 ysize-1) */
eyesep Separation	/* eye separation (0) */

SurfDef:

/* Give a name to a set of surface attributes. */
surface Name <SurfSpec> [<SurfSpec> ...]

Surface:

/* Surface specification */
<SurfSpec> /* Use given attributes */
Surfname [<SurfSpec> ...] /* Use named surface */
cursurf [<SurfSpec> ...] /* Use cur. surface */

SurfSpec:

ambient R G B	/* Ambient contribution */
diffuse R G B	/* Diffuse color */
specular R G B	/* Specular color */
specpow Exponent	/* Phong exponent */
body R G B	/* Body color */
extinct Coef	/* Extinction coefficient */
transp Ktr	/* Transparency */
reflect Kr	/* Reflectivity */
index N	/* Index of refraction */
translu Ktl R G B Stpow	/* Translucency, transmit diffuse, spec exp */
noshadow	/* No shadows cast on this surface */

Effect:

mist R G B Rtrans Gtrans Btrans Zero Scale
fog R G B Rtrans Gtrans Btrans

Atmosphere:

atmosphere [Index] <Effect> [<Effect>...]

ApplySurf:

applysurf <Surface>

Instance:

<Object> [<Transforms>] [<Textures>]

Object:

Primitive
Aggregate

ObjDef:

/* define a named object */
name Objname <Instance>

Primitive: /* Primitive object */

plane	[<Surface>]	Xpos Ypos Zpos Xnorm Ynorm Znorm
disc	[<Surface>]	Radius Xpos Ypos Zpos Xnorm Ynorm Znorm
sphere	[<Surface>]	Radius Xpos Ypos Zpos
triangle	[<Surface>]	Xv1 Yv1 Zv1 Xv2 Yv2 Zv2 Xv3 Yv3 Zv3 /* flat-shaded */
triangle	[<Surface>]	Xv1 Yv1 Zv1 Xn1 Yn1 Zn1 Xv2 Yv2 Zv2 Xn2 Yn2 Zn2 Xv3 Yv3 Zv3 Xn3 Yn3 Zn3 /* Phong */
polygon	[<Surface>]	Xv1 Yv1 Zv1 Xv2 Yv2 Zv2 Xv3 Yv3 Zv3 [Xv3 Yv4 Zv4 ...]
box	[<Surface>]	Xlow Ylow Zlow Xhi Yhi Zhi
cylinder	[<Surface>]	Radius Xbase Ybase Zbase Xapex Yapex Zapex
cone	[<Surface>]	Rbase Xbase Ybase Zbase Rapex Xapex Yapex Zapex
torus	[<Surface>]	Rswept Rtube Xpos Ypos Zpos Xnorm Ynorm Znorm
blob	[<Surface>]	Thresh Stren Rad Xpos Ypos Zpos [Stren Rad X Y Z ...]
heightfield	[<Surface>]	Filename

Aggregate:

Grid
List
Csg

Grid:

grid X Y Z <ObjItem> [<ObjItem> ...] **end**

List:

list <ObjItem> [<ObjItem> ...] **end**

Csg:

union <Object> <Object> [<Object> ...] **end**

intersect <Object> <Object> [<Object> ...] **end**
difference <Object> <Object> [<Object> ...] **end**

/* CSG will only work properly when applied to closed objects, e.g.:
 * sphere, box, torus, blob, closed Aggregate, other Csg object
 */

Transforms: /* Transformations */
translate Xtrans Ytrans Ztrans
scale Xscale Yscale Zscale
rotate Xaxis Yaxis Zaxis Degrees
transform A B C
 D E F
 G H I
 [Xt Yt Zt]

Textures:
texture <TextType> [Transforms] [<Texture> [Transforms] ...]

Texture:
checker <Surface>
blotch Scale <Surface>
bump Bumpscale
marble [Colormapname]
fbm Offset Scale H Lambda Octaves Thresh [Colormapname]
fbmbump Offset Scale H Lambda Octaves
wood
gloss Glossiness
cloud Offset Scale H Lambda Octaves Cthresh Lthresh Transcale
sky Scale H Lambda Octaves Cthresh Lthresh
stripe <Surface> Width Bumpscale
image Imagefile [<ImageTextOption> [<ImageTextOption> ...]]

ImageTextOption:
component <SufComp>
range Lo Hi
smooth
textsurf <Surface>
tile U V <Mapping>

SurfComp:
ambient
diffuse
reflect
transp
specular
specpow

Mapping:
map uv
map cylindrical [Xorigin Yorigin Zorigin Xup Yup Zup Xu Yu Zu]
map linear [Xorigin Yorigin Zorigin Xv Yv Zv Xu Yu Zu]
map spherical [Xorigin Yorigin Zorigin Xup Yup Zup Xu Yu Zu]

Light:
light R G B <LightType> [noshadow]
light Intensity <LightType> [noshadow]

LightType:
ambient

point	Xpos Ypos Zpos
directional	Xdir Ydir Zdir
extended	Xpos Ypos Zpos Radius
spot	Xpos Ypos Zpos Xat Yat Zat Coef Thetain Thetaout
area	Xorigin Yorigin Zorigin
	Xu Yu Zu Usamples
	Xv Yv Zv Vsamples

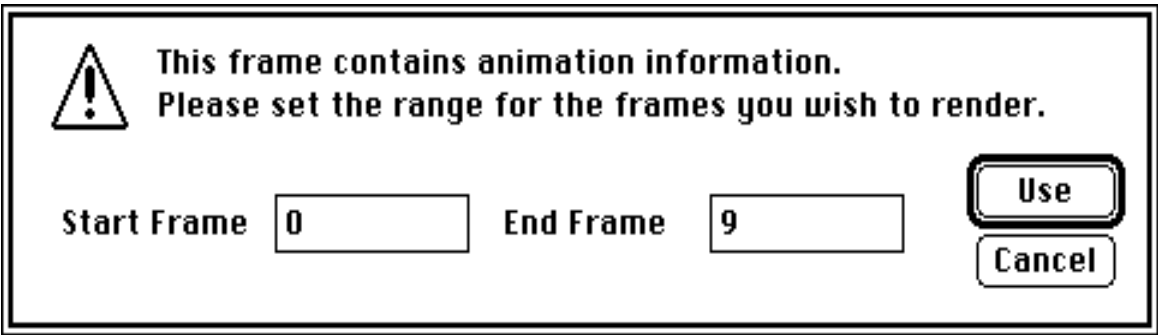
RenderOption:	
samples	Nsamp [jitter nojitter]
background	R G B
outfile	Filename
frames	Nframes
starttime	Time
contrast	R G B
maxdepth	Depth
cutoff	Factor
report	[verbose] [quiet] [Freq] [Statfile]
shadowtransp	

Definition:

define	Name Expr
---------------	-----------

Appendix B

Animation



The object editor contains no support for animation. Animation information contained within a script file may be lost if the user changes the scene or generates an output script. It is still possible to render animations but a warning will be displayed before proceeding, allowing the user to set the frame range if they wish.

Bibliography

- APPL85** Apple Computer Inc., *Inside Macintosh Volumes I-V*, Addison-Wesley, Reading, Massachusetts, 1985.
- BUCK91** Buck, D.K., *DKBTrace User's Manual*, 1991
- FOLE90** Foley, J.D., A. van Dam, S.K. Feiner and J.F. Hughes, *Computer Graphics, Principles and Practice*, Addison-Wesely, Reading, Massachusetts, 1990
- GLAS89** Glassner, A.S., *An Introduction to Ray Tracing*, Academic Press, Oval Road, London, 1989
- KERN88** Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Prentice Hall, New Jersey, USA, second edition 1988
- KOLB91** Kolb, C.E., *Rayshade User's Guide and Reference Manual*, 1991
- SYMA91a** Symantec Corporation, *THINK-C User Manual*, Symantec Corporation, Cupertino, California, 1991
- SYMA91b** Symantec Corporation, *THINK-C Standard Libraries Reference*, Symantec Corporation, Cupertino, California, 1991
- WANG88** Wang, P.S., *An Introduction to Berkely UNIX*, Wadsworth International, Belmont, California, 1988

Programmers Notes

Preface

This section of the report intends to cover the technical, programming aspect of the project. It will attempt to show what has had to be done in order to convert *rayshade* from its Unix origins into a Macintosh application, what limitations have been imposed on the conversion, and what acceleration techniques have been employed to make creating pictures faster. It will also show the differences in programming styles between the new code and the old. Finally a 'wish-list' of features that would be desirable to make *rayshade-M*.

Changes Required

Many parts of rayshade have required changing to better fit the Macintosh environment. For the most part, these changes affect the input and output functions, and not the ray tracing code. For the most part, any changes that have been made to the original code are invisible to the rest of the program. For instance, the memory allocation module has been rewritten so that more than one script could be loaded into rayshade, but all of the procedure calls and parameters have been left exactly the same.

In addition to making the original code work, I have added new code for script generation, structure deleting, object and scene editing, and PICT format image generation.

This chapter details some of the areas which required moderate to extensive work during the course of this project.

Memory Allocation

Rayshade is a Unix command-line tool. Only one script is expected for each execution of the program, and all memory freeing was handled by system functions when the program terminated.

The Macintosh environment does not allow for this style of programming. Most applications are capable of cleaning-up after the user wishes to close a project, and then be ready to load a new one.

In order to make *rayshade* capable of loading more than one script, it was necessary to add some house-keeping code that reinitialised structures and reset pointers within the code. The simplest way of doing this was to change the memory allocation procedures so that each allocation was 'remembered'. When a reset was required, the remembered allocations could then be freed. Not only is this method quicker, but it is also less inclined to 'leaks' that might occur if the program tried to free the memory by recursively descending memory, freeing each node in turn.

Code has been written to delete sections of the object tree by recursion, however this is used for deleting single items from the scene, rather than dismantling everything at once.

Image Format

The original *rayshade* by default used the *Utah Raster ToolKit RLE* image format. Since this was not a widely available standard on the Macintosh, it was necessary to change the code so that images are save in PICT format. The image is saved to file from a 24-bit buffer, and not from the output window. This enables rayshade to be used with any type of monitor and still save the same level of quality of images.

There was not enough time left to change the image mapping to PICT format, so that image mapping still requires that the input image be in *mtv* raw format.

Script File Handling and Generation

Macintosh applications tend to steer away from script based approach used by many Unix tools, instead they adopt more graphical, intuitive methods. To hide the script from the user, a series of dialogue boxes were created which provide a way to edit objects, lights, textures, transforms and surfaces without editing the script directly through something like a text editor.

Since the built-in object editor allows the user to create, modify and delete objects within the scene, it was necessary to provide some method of saving this new scene to disk. Since *rayshade* parsed a script, it was natural for the output be a script as well.

A script generator had to be created that worked by recursively traversing the object tree, generating each object in turn, complete with any transformations, surfaces or textures it might have. This script file is human readable and takes advantage of any named surfaces or objects that may have been declared, however no attempt has been made to add animation information into the output script, nor any references that are made in the object tree to other files.

Rendering

Since the Macintosh user expects some form of visual feedback it was desirable to render a scene directly to a window rather than to a file. An 24-bit image buffer is created which used to store the picture.

Rayshade has no method for displaying a picture. Instead they are rendered directly to file. On the Macintosh it is essential to get some form of feedback, so I decided to draw the picture to the screen instead. The user is still able to save the image, but this is done from a menu option, rather than automatically.

All of these dialogues have been integrated together into an object editor. The object editor additionally allows the user to preview the scene in wireframe, design new objects and view the contents of the object tree in list form.

Program Modifications

This section lists all of the major modifications / additions the have been made to the *rayshade* code:

All superfluous `#defines`, `#ifdef` which cater for other computer platforms are removed from code. This makes the program clearer to understand. Additionally, the source and header files have all `#include` directives modified to account for the different file structure on the Macintosh, e.g. `#include "libray/libcommon/ray.h"` is changed to `#include "ray.h"`.

Code is written for library functions which are standard on Unix systems, but not included with THINK C, e.g. `hypot`, `rand`, `bcopy`.

Simplifying of some C statements that THINK C compiles incorrectly. For instance in `blob.c` the line

```
*tmpc++ += c[j];  
needed to be changed to  
*tmpc += c[j];  
tmpc++;
```

Some static variables are made global, so that they could be accessed by the object editor, e.g. Surface and Light definition list pointers.

Memory allocation module is completely rewritten, so that the location of all allocations is remembered. This enables the object tree to be freed in a simple loop, rather than by recursively freeing each element of the tree.

A macintosh event handler and menu bar is added so that *rayshade* renders pictures from a menu selection.

The picture generation code is modified so that *rayshade* renders directly into a window, instead of a file. Images have been fixed so that they are not rendered upside-down (as they were previously). The image is stored in a 24-bit offscreen buffer, which is used to refresh the screen when necessary.

PICT image generation code is added. This replaces the generic image file or RLE format which *rayshade* normally uses. PICT images are generated from the 24-bit offscreen buffer and not from the screen.

Main.c is rewritten so that the ray-tracer is called only when selected from the menu.

Three dialogue boxes called Screen options, Camera Options and Render Options replace the command-line parameters that were required until now.

Raytracer is made reentrant so that scenes can be rendered more than once at a time. More than one script can also be loaded into *rayshade* without restarting the program each time.

An object editor is added to *rayshade*, using an orthographic wireframe representation of the scene. An object list allows the user to traverse the contents of the scene. Each object primitive requires its own wireframe code generator.

A cross-hair and zoom system is created. The zoom allows the user to focus on interesting parts of the scene. The crosshair is used when creating new objects which are centred on the current crosshair position.

Object filter buttons are added so that it is possible to mask certain types of object from the scene. In a particularly complex scene it is possible to hide objects, thus making the scene more understandable and faster to draw.

The window rendering code is changed so that it is possible to render with up to 32x32 'pixel' size.

Object dialogues for each primitive type are created. Heightfields do not yet have a dialogue box.

A surface editor is created. This dialogue box uses the Macintosh ColorPicker in determining which surface colours to use.

Code to detect for animation is added. The user is able to set the frame range of an animation before rendering begins.

Object instance alert boxes are added. These warn users if they are about to edit object instances.

Rendering code is changed to allow the user to abort rendering at any time.

A texture editor is added. One or more textures can now be applied to objects.

A transformation editor is created. Objects, or groups of objects can now be scaled, rotated or translated.

A script generator is added.

Known Bugs in Rayshade-M

There are some bugs in Rayshade-M which I was unable to repair in time to meet the deadline. For the most part these can be avoided, however it worth noting that none of them are too major:

Scaled textures sometimes appear 'speckled'. This also manifests itself when the ray tree depth is large.

Script files will only load from the Rayshade folder.

Reentrant code sometimes fails.

Rayshade-M terminates if it is given an incorrect input file, or if an error occurs during rendering.

Rayshade-M programming techniques

Rayshade-M has been designed to cause as little change to the underlying ray tracing code as possible. This has meant that there is a different style between the original code and Macintosh controller.

THINK C is an ANSI C compiler, however *rayshade* has been written in K&R C. Now attempt has been made to add prototyping to the function declarations of the original code, however all the of the new code is ANSI style.

The original code is highly object orientated. Each object, light and texture has its own Methods structure which holds pointers to functions that are used for common operations such as intersections tests. To really merge the new features of *rayshade-M* into the rest of the code, it would require extending the Methods structures to include function procedure pointers for all of the additional functions each element now carries out. In the case of object primitives, the methods structure would need to be extended to include::

- Drawing a dialogue box
- Creating a wire frame representation of an object
- Creating a 'prototype' object
- Deleting an object (recursively in the case of aggregates)
- Generating a script command for the objcv.

Choosing this object orientated approach would have meant major interference to the original code. This would result in extra effort being required when upgrade patches became available. (Six patches were released during the development of this project).

The other approach, which was chosen for this project, does not involve object-orientation. Instead it uses a more conventional switch statement to determine the correct procedure to call for an operation, e.g.:

```
void OpenObjectDialog(Geom *obj) {
    switch(ObjectType(obj)) {
        case SPHERE:
            doSphereDialog(obj);
            break;
        case CONE:
            doConeDialog(obj);
            break;
```

```

        case CYLINDER:
            doCylinderDialog(obj);
            break;
    }
}

```

Rather than the more compact, object orientated method:

```
(obj->methods->drawobjectdialog)(obj);
```

Wishlist

There have been many things that I have wanted to add to this project, but simply didn't have the time to do. Many of these new features would certainly fill some of the inconsistencies that I am aware exist at the moment. Others would actually increase the usefulness of the program.

- Speeding up the refresh on the wireframe drawing
- ANSI Prototyping of original code
- Wireframe preview
- Fractal mountain generation using height fields
- Height field editing
- Importing PICT files as image maps
- Animation support
- More consistency
- Texture map preview
- More utilisation of Macintosh interface
- Instance creation
- Extrusion and Lathe tools
- Surface patch generation
- Distortion tools
- Leakproof memory deallocation code
- Better integrated code - perhaps object orientated