RayGL Users Manual

Quick Start Guide

1 Download the RayGL object and header files

2 Include the RayGL library in your makefile

3 Include raygl.h in all source code

4 Include raygldefs.h in all source code (this must be **AFTER** the raygl.h include)

5 Add a call to rayglFrameBegin("/tmp/movie") prior to the first glBegin

6 Add a call to rayglFrameEnd() prior to swapping buffers

7 Compile and run your program

8 Find .sdl files in /tmp

9 Invoke POV-Ray, povray +P -UV +w800 +h600 /tmp/movie000000001.sdl

Detailed Guide

1. Preparing your code for use with RayGL

1.1 Code Organization

RayGL requires that the code for drawing objects is separated from the code for setting up the scene. The setup for the scene consists of positioning and aiming the lights, positioning and aiming the camera, and generating any textures that may be used. The setup for the scene must be done before any objects are drawn to the buffer.

1.2 Encapsulating a frame

rayglFrameBegin should be called at the beginning of each frame prior to drawing objects, and after the scene is setup. rayglFrameEnd should be called at the end of each frame after the last object has been drawn to the buffer. If the setup for the scene is modified between the call to rayglFrameBegin and rayglFrameEnd, the changes may not be applied consistently to the frame, may only affect the next frame, or may cause an error. Changes made to the scene after calling rayglFrameEnd will affect the next frame. Calling rayglFrameBegin or rayglFrameEnd multiple times per frame will cause an error.

RayglFrameBegin takes a string representing a file name as a parameter. This string will be the base name of the files that will be output by RayGL. The SDL files output by RayGL vary in size depending on the complexity of the scene being rendered and the method used to describe the objects in OpenGL. Files may range anywhere from a few kilo bytes to several mega bytes. When choosing a path for the files, consider that a smooth video display will require tens of frames per second. Assuming a moderately complex scene with files of size 2MB, to represent the SDL files for one minute of this scene at 20 frames per second, 2.4GB of storage will be required.

1.3 Including RayGL in your code and Makefile

To use RayGL, the RayGL header files need to be included in each file that makes calls to OpenGL. raygl.h must be included before raygldefs.h. The first header

contains declarations for the RayGL functions, and the second header contains preprocessor commands that alias a subset of the OpenGL API to RayGL.

The RayGL library also needs to be linked to your project, to do this you will need to add it to your Makefile. Add the directory containing RayGL headers and library to a macro, and add the RayGL object file to the shell line for your target. For example:

```
CC=g++
FLAG=-c -Wall
... various other macros
RAYGL=../raygl/
movie: movie.o
$(CC) $(RAYGL) $(CFLAGS) -o $@ raygl.o
```

2. Using RayGL

2.1 Generating and Rendering Basic SDL

Once each frame has been encapsulated within a rayglFrameBegin and rayglFrameEnd, the minimum requirements for generating SDL files have been met. After the program has been compiled with RayGL, running the program will generate new SDL files in the location that was specified in the call to rayglFrameBegin. These files can then be used to render images with POV-Ray. To render an image invoke the following on the command line:

~$ povray +P -UV +w800 +h600 <file>A brief explanation of the options used:

+P pause when done, this option causes the result to be displayed after rendering

-UV disables the vista buffer, the vista buffer is an optimization that can interfere with RayGL

+w800 +h600 set the resolution of the output to 800 by 600, smaller resolutions render faster

2.2 Lighting

If lighting is enabled in the OpenGL and all of the light sources have been defined prior to calling rayglFrameBegin, light sources will be added to the SDL file. However, there is an issue with the translation between the lighting model used for OpenGL and the model used for POV-Ray. OpenGL defaults to have no attenuation, lights do not fade over distance. POV-Ray uses attenuation in all of its lighting equations and RayGL does not have a good way to interpret the values from OpenGL into POV-Ray. To manually adjust the attenuation factor, the functions setFadeDistance and setFadePower have been provided. These values may need to be adjusted to match the size of the scene and distance between objects to get the desired effect.

2.3 Texturing

To provide support for texturing in POV-Ray, RayGL relies on the image_map to transfer bitmapped textures. The current version of RayGL is limited to the image types that are supported by POV-Ray. These types are: gif, tga, iff, ppm, pgm,

png, jpeg, tiff, and bmp. RayGL makes use of texture objects to manage texturing. In normal OpenGL operation a list of texture names would be created with glGenTextures, a 2D texture would be defined by a call to glTexImage2D or gluBuild2DMipmaps which in turn calls glTexImage2D, then the texture would then be bound to a texture name by calling glBindTexture, and when the texture was required it would be recalled by calling glBindTexture. Using textures in RayGL follows this same pattern with one additional step. RayGL defines a struct called Image in raygl.h, this struct contains meta data about the texture that is needed for translating to POV-Ray. After generating a list of texture names and before defining a texture using glTexImage2D, the texture must be loaded into an Image struct, and this Image struct must be passed to the function rayglCustomLoadedTexture. The field texName does not need to be specified, this value will be generated by the call to glTexImage2D.

Translating between the texture coordinates in OpenGL and the texture coordinate used in POV-Ray presents another issue. To get correctly mapped textures for most objects will require adjustments to the scale, position, and shape of the texture by using rayglScaleTexture and rayglTranslateTexture, and rayglTextureType. These functions are defined in the API. It should be noted that these functions alter the state of the texture translation, once they are invoked they affect all subsequent textures until they are invoked again.

3. Considerations

3.1 Scale

Try to decide on a scale to use when drawing the objects. Integrating all of your objects into one scene will be difficult if they are all of different scales. RayGL may require some tweaking of its lighting variables for large (100+) size objects. Resolving scale issues may not be as easy as calling glScale for each object as this may affect other object transformations.

3.2 Camera

Decide on a method for handling your camera, RayGL requires that the camera be setup for the scene prior to drawing any objects. To ease integration, draw all objects at the origin until multiple objects need to be displayed at once. Providing a method to skip to a specific point in the animation will make it considerably easier to edit the animation. Because RayGL causes OpenGL programs to slow down, mapping setRenderPov to a key is very useful while editing.

3.3 Counter Clockwise Rule

Follow the counter clockwise rule. OpenGL determines the front face of an object by following your vertices and assuming you specified them in counter clockwise order. If you do not follow this rule, your objects will appear to draw correctly, however they will cause major problems when it comes time to integrate them with the rest of the scene. RayGL does not support clockwise objects, these objects will be drawn facing the wrong direction or inside out.

3.4 Lighting

RayGL requires that all light sources that are present in the scene be specified before drawing objects. This is because all of the light sources are considered during the ray tracing calculations; there is no way to know beforehand which lights will affect which objects.

RayGL provides a pgm texture loader, however if you want you can use your own texture loader, you will need to provide the necessary information to the image meta objects in order for RayGL to be able to provide texture support. See the API section on rayglCustomLoadedTexture.

3.5 Texturing

Applying 2D textures to 3D objects in POV-Ray is an art form, tweaks to texture position, scale, and normals may need to be performed. Expect to spend some time adjusting for this.

3.6 Error Detection

There is a difference between an OpenGL program that runs and an OpenGL program that runs correctly. Be sure to check for errors in your OpenGL by using glGetError. RayGL does not call glGetError because it is a destructive call, once it is called the error flag is reset to zero. RayGL cannot be expected to work for OpenGL programs that have errors. One example is calling glBegin twice before making a call to glEnd, for some implementations of OpenGL your program may continue to work, however this puts RayGL in an unrecoverable state.

3.7 Display Lists

RayGL does not currently include support for display lists. While display lists are a good way to organize objects that will be used multiple times and a much more efficient way to run OpenGL, the slowdown inherent in writing complex SDL files to disk eliminates the need for efficiency in the OpenGL.

3.8 Vista Buffer

If your scene is getting unexpected errors rendering in POV-Ray, try turning the vista_buffer off by adding -UV to your command line. The vista buffer is an optimization technique used in POV-Ray to decrease the render time by subdividing the image into 4 quadrants, unfortunately, by default the optimization only improves scenes with many object (10+), and may cause errors as a result of changes in the camera vector.

3.9 Planar Polygons

Be certain that all polygons are planar. The OpenGL specification does not guarantee that non-planar polygons will render, this is another example of a situation where OpenGL will continue to run with errors for some implementations. POV-Ray is unable to render non-planar polygons, SDL files containing non-planar polygons will report an error and fail to render.

3.10 Commenting

Use rayglInject to help track down problems. This function can inject comments directly into the SDL file, in the case of a non-planar polygon, this is a good way to determine which object the polygon belongs to.

3.11 Performance

For scenes that render too slowly in POV-Ray, adding a skysphere by means of rayglInject can have a dramatic effect on performance.

API

Active Calls

Active calls are RayGL function calls that are explicitly made by the programmer. These function are unique to RayGL, they exist to provide RayGL with meta data about a scene that is difficult or impossible to intuit from arbitrary OpenGL. For example, without relying on a specific windowing system or buffering method, the beginning and end of a frame cannot be determined. rayglFrameBegin and rayglFrameEnd allow the user to explicitly mark the beginning and end of a frame. These markers signal when to start and stop writing to an SDL file.

Frame markers

This section consists of 4 functions, rayglFrameBegin, rayglFrameEnd, setRenderPov, and getRenderPov. rayglFrameBegin and rayglFrameEnd define the beginning and end of a frame block. OpenGL code specified outside of a frame block will not be displayed. They are the only function calls required to create a valid POV-Ray SDL file. Without using them, SDL files will not be created.

SDL file creation can be overridden by calls to setRenderPov. Attempting to use other RayGL functions outside of a frame block without setting renderPov to false will cause a segmentation fault if those functions write to the file. To check the current state of renderPov, the getRenderPov function has been provided.

rayglFrameBegin

void rayglFrameBegin(char* fileName)

This function creates a file to write ray code to, and then writes out the basics setup for camera and lighting. The camera and lighting for the scene should be setup in OpenGL prior to calling it. The file is closed by a matching call to rayglFrameEnd().

Parameters:

char* filename: this is the filename that will be used for the SDL output files

rayglFrameBegin

void rayglFrameBegin(char* fileName, GLint eyeX, GLint eyeY, GLint eyeZ, GLint centerX, GLint centerY, GLint centerZ, GLint wX, GLint wY, GLint wZ)

This fully parameterized version of rayglFrameBegin is used when displaying in orthographic mode.

Parameters:

char* filename: this is the filename that will be used for the SDL output files

GLint eyeX, eyeY, eyeZ: these value represent the coordinates of the viewer

GLint centerX, centerY, centerZ: these values represent the coordinates that the viewer is looking at

GLint wX, wY, wZ: given a vector formed by connection the eye and center coordinates, these values modify that vector by rotating about the x, y, and z axis

rayglFrameEnd

void rayglFrameEnd()

This function is used to signal the end of a frame, it closes the SDL file, and must be called before starting a new frame. This could be abstracted away into the logic of rayglFrameBegin in future versions. For increased stability it may be

desirable to call this function if a rayglFrameBegin finds the file handle is still open.

setRenderPov

void setRenderPov(GLboolean on)

Allows user to turn SDL file writing on and off. The ability to disable RayGL may make it easier to preview certain sections of a scene for projects that don't allow the user to jump directly to a given frame.

Parameters:

GLboolean on. The state that renderPov is being set to.

getRenderPov

GLboolean getRenderPov()

Allows the user to detect if RayGL is enabled.

Texture manipulation

This section consists of 5 calls, rayglTexture, rayglDisableTexture, rayglScaleTexture, rayglRotateTexture, and rayglTextureType. rayglTexture and rayglDisableTexture are deprecated. rayglTexture enables the user to pass a string containing a POV-Ray style texture specification directly to the SDL file. This was useful before OpenGL textures were working, but is not in the spirit of RayGL. rayglDisableTexture sets RayGL to its default state, using OpenGL textures in the SDL file.

rayglScaleTexture, rayglRotateTexture, and rayglTextureType are all used to adjust the layout of textures. There are some problems with the conversion between OpenGL texture mapping and POV-Ray texture mapping. Ideally, the mapping details would be detected automatically, until that is perfected, these functions allow the user to manually tweak the state of the texture mapping.

rayglTexImage2D, rayglCustomLoadedTexture, and raygluBuild2DMipmaps are used to work with the image meta data and prepare texture as image objects to be bound to geometric objects. The meta data structure that stores the image information is described below. The sizeX and sizeY values are not required; these values represent the vertical and horizontal width of the texture but are not presently used. The texName stores the texture name assigned by calling texImage2D or gluBuild2DMipMaps, the texFileType is the file extension representing the format of the file to be loaded (POV-Ray supports a limited number of formats, the selection varies by operating system), the texFileName field stores the full filename including the path to the file if it is not located in the working directory. The data field stores the binary data, finally, the toString field is an embedded function that displays the image information on the command line.

```
struct Image {
        unsigned long sizeX;  unsigned long sizeY; //the dimensions of the texture
        GLuint texName; //a texture name assigned by OpenGL
        char* texFileType;  char* texFileName; //the file format and full file name
        uchar* data; //the texture data
         void toString(){cout<<"name: "<<texFileName<<" texName: "<<texName<<" Size x,y"
        <<sizeX<<","<<sizeY<<endl;}  //a function to assist with debugging
}
```

rayglTexture

void rayglTexture(char * texture)

This function changes texturing to use POV-Ray textures and takes a string representing the SDL specification for the texture to be used. Contributed by Rick Fillian.

Parameters

char * texture. This string will be added to the texture specification of each object written to the SDL file while it is in effect.

rayglDisableTexture

void rayglDisableTexture()

This function changes texturing to use OpenGL textures, this is the default state of RayGL. Contributed by Rick Fillian

rayglScaleTexture

void rayglScaleTexture(GLdouble texX, GLdouble texY, GLdouble texZ)

This function adjusts the scale of a texture. It is used for texturing non-unit size polygons. Hopefully this will be able to be automated later. Default value is 1; this is the correct setting for a 1X1 square. This is global, when rendering a texture to the SDL file, the current value is checked. Reset it for each object or group of objects as needed.

Parameters

GLdouble texX, texY, texZ. These are the scale values in the x, y, and z dimensions for the texture.

rayglTranslateTexture

void rayglTranslateTexture(GLdouble texX, GLdouble texY, GLdouble texZ)

This function translates the texture across the surface of the object it is being drawn on.

Parameters

GLdouble texX, texY, texZ. These represent the offset of the texture in the x, y, and z dimensions.

rayglTextureType

void rayglTextureType(GLint texTypeIn)

This function changes the way that POV-Ray maps a texture to an object. The texture map can be distorted to fit many different forms. The default value is 0; this represents the conventional mapping of a texture to a flat plane. Other texture types can be useful for translating texture objects between quadric and Glut objects. This change is global and the current setting will affect an object when it is written to the SDL file. Using the wrong type when creating an object can result in oddly misshapen texture objects.

Parameters

GLint texTypeIn. This is an integer mapping to the following texture modes:

0 = planar, 1 = spherical, 2 = cylindrical, 3,4 undefined, 5 = torus

rayglCustomLoadedTexture

void rayglCustomLoadedTexture(Image* image)

This function provides support for customized texture loaders, if some method other than the provided PGM IO functions is used to load textures from a file into memory, then this function should be called to create the meta data needed to specify the texture file in POV-Ray.

Parameters

Image* image. An image struct as defined in raygl.h, this struct stores the image for later use.

Commenting and code injection

This section defines rayglInject. The rayglInject function was added by request of the students. Its initial purpose was to allow students to inject comments directly into the SDL output file, and for a while it was also being used as an aid to debug RayGL. However, since the input to RayGL is a trusted source, no checking on the input to rayglInject is done. As a result, some students have used it to inject POV-Ray code directly into the SDL. In other words, this allows the students to add certain POV-Ray only objects and textures directly into their SDL from OpenGL.

rayglInject

void rayglInject(char* inject)

The primary purpose of this function is to provide the user with a way to insert

comments into their SDL files. The size of these files varies in magnitude from 100 lines to 1,000,000 lines depending on the complexity of the objects in the scene and the methods used to define them in OpenGL. POV-Ray recognizes C style commenting. Calls to rayglInject are only appropriate when a SDL file is open for writing.

Parameters

char* inject. This parameter contains a pointer to the string that is to be inserted into the SDL file.

Lighting

The default setting for OpenGL's lighting models does not factor in for light attenuation over distance or in relation to the intensity of the light sources. This is not a problem in OpenGL because it does not handle reflections. This does not translate well to POV-Ray's lighting model, an enclosed scene without attenuation would be completely lit in this model, and no shadows would be visible regardless of the location of the light source or sources. Another problem encountered here is that scale has no effect on lighting when using OpenGL's default settings. When determining a scale early in their projects, student groups don't take this effect into account. Given a scene rendered at two different scales in OpenGL, 1X1 And 1000X, using the default OpenGL lighting model will result in the same lighting in both renders. However in POV-Ray these two scenes would be rendered in very different ways due to attenuation. These functions have been

provided to help the students to adjust to this while maintaining their original OpenGL lighting and scene scale.

rayglFadeDistance

void setFadeDistance(GLfloat distance)

This adjusts the distance at which light begins to fade. It should be relative to scene size. This affects the way in which all lights in the scene act.

Parameters

GLfloat. The distance at which light begins to fade. The default value is 100.

rayglFadePower

void setFadePower(GLfloat power)

This adjusts the change in the rate that light fades with distance. It represents the power that distance is raised to in the lighting equation. The value usually ranges between 1.4 and 2.0 where 1.4 tends to make for a good looking scene, and 2.0 is a close approximation of reality. This is also global and affects all lights. The default is 1.4, it is a compromise between 1.0, linear attenuation, and 2.0 realistic attenuation.

Parameters

GLfloat power. The change in the rate at which light fades. Default value is 1.4.

Passive calls

These functions are not explicitly called by the code. Instead, they act to intercept similarly named OpenGL functions. The raygldefs header file contains C preprocessor directives that search and replace occurrences of specific OpenGL function calls with RayGL function calls before compile time. The passive calls capture the original OpenGL parameters and translate them into SDL; then they proceed with the original OpenGL call. The parameters in this section are described by the OpenGL specification for the corresponding OpenGL function.

Camera and Aspect

The following two functions adjust the perspective from which the scene is rendered and the location and direction vector of the camera. raygluPerspective adjusts the aspect ratio and the field of view for the SDL scene.

RayGL uses an orthographic or perspective camera depending on which rayglFrameBegin function is called, regardless of which function was used, if gluPerspective is called, a perspective camera will be created for the scene. Using the model of an eye looking into the scene, aspect ratio adjusts the ratio of the horizontal viewable area to the vertical viewable area. If a finite imaginary plane is drawn in front of the eye, this value adjusts the ratio of the sides of the plane. Field of view adjusts the horizontal viewing angle of the plane, depending on the distance of the plane form the user, more or less of the plane will be within the viewable angle.

raygluPerspective

void raygluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)

This function captures calls to gluPerspective. It calls gluPerspective and then harvests the fovy and aspect values. These values are used to adjust the POV-Ray scene while the zNear and zFar values are not translated. This function does not write to the SDL, but rather adjusts the state of RayGL thereby affecting the camera parameters used for the next frame. In the SDL, this adjusts the cameras right vector and angle. A combination of both aspect and fovy is used to create the new right vector. Fovy is translated directly as the camera angle.

raygluLookAt

void raygluLookAt(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ, GLdouble centerX, GLdouble centerY, GLdouble centerZ, GLdouble upX, GLdouble upY, GLdouble upZ)

This function captures calls to gluLookAt. It calls gluLookAt and then harvests the eye, center, and up values. The eye values adjust the camera's location vector, this places the camera at those coordinates in the scene. The center values adjust the look_at vector; this represents a vector drawn from the camera location to the part of the scene that the camera is looking at. In effect it is the rotation of the camera about the location of the camera. The up values adjust the sky vector; this

has the effect of changing which direction is up relative to the camera, or the rotation of the camera about its direction vector.

Primitives

OpenGL primitive objects are defined by combinations of vertices delimited by glBegin/glEnd markers which determine the type of the primitive. While inside of a glBegin/glEnd sequence, OpenGL is limited to a subset of its normal API. OpenGL calls from outside of this subset will either return the expected result, uninitialized memory, or attempt to access memory outside of the current stack frame. Because of this, before calling a glBegin, several values of the OpenGL state machine need to be polled prior to executing the glBegin. The state just before entering a begin/end sequence is used to describe the parameters of the ensuing SDL object.

The rayglVertex functions gather each vertex as it is described, once the glEnd is reached, the vertices and state of the object are written out. OpenGL supports defining multiple objects of the same primitive type per glBegin/End sequence. The primitive type described in the glBegin determines how many vertices should be used for each object in a begin/end sequence. Because of some differences in how objects are described in OpenGL and POV-Ray, the first vertex is maintained separately for some types so that it can be used later to close the object. With exception glPoints which are translated as small spheres in POV-Ray, all other object types are redefined as polygons.

rayglBegin

void rayglBegin(GLenum mode)

This function is used to implement the standard glBegin. Prior to calling glBegin it polls OpenGL for a series of state variables which are then maintained in RayGL until the next call to glBegin. This is done because the OpenGL specification does not guarantee that calls to glGet will return with correct data and without errors while inside of a glBegin/glEnd sequence. Since the OpenGL state is used in complex and primitive objects, harvesting the current OpenGL state is abstracted out to the getAtribs function. rayglBegin also initializes multiple RayGL variables that are used during vertex processing.

rayglEnd

void rayglEnd()

This function implements the standard glEnd. Following that it signals RayGL that it is leaving the begin/end sequence. For all primitives with a known multiple of vertices, a representative polygon complete with matrix and attributes is written to the SDL each time the required number of vertices have been specified. However, in the case of a polygon type with an unknown number of vertices, the object cannot be written to the SDL until a glEnd has been reached, there is no other way to signal that the last vertex in the polygon has been reached. For primitive of type polygon, this function calls writeAtribs and writeMatrix.

rayglVertex3

void rayglVertex3i(GLint localX, GLint localY, GLint localZ)

void rayglVertex3f(GLfloat localX, GLfloat localY, GLfloat localZ)

void rayglVertex3d(GLdouble localX, GLdouble localY, GLdouble localZ)

This function is used to implement the standard glvertex3[i, f, d], where I, f, and d stand for int, float, and double. Different code paths are used depending on the current type of object being described. For all but the polygon type, writeAtribs and writeMatrix will be called upon finding the last vertex in each object. When creating polygons in POV-Ray, one additional vertex is needed for each polygon. This additional vertex is identical to the first vertex in the object; it is used to show that the object was closed along all edges.

glu and glut objects

While RayGL was originally intended to only support the OpenGL 1.2 specification, support for glu and glut objects is a worthwhile addition. The functions provide a considerable addition to RayGL's functionality without creating a lot of extra work. In setting up the attributes for these objects, they are treated similarly to n-sided polygons. However, each object does require some additional work to translate between its description in OpenGL to a description of a similar object in POV-Ray While these objects can all be modeled in POV-Ray, there are some issues with edge cases. If a cone or cylinder is drawn using gluCylinder or glutSolidCone, it is modeled in POV-Ray as a cone. In the case of the cylinder the cone is specified such that the walls of the cone do

not converge. A problem occurs in both situations if the cone or cylinder is specified in OpenGL with zero height. In OpenGL this will do nothing and continue with the program, gluCylinder do not have caps so nothing will be displayed, however POV-Ray cannot render a cone with a zero height. So while this object does not affect the OpenGL scene, trying to render a POV-Ray scene with an object like this will fail.

The glu objects are faceted approximations of smooth shapes; the number of facets is adjusted by modifying the slices and loops values. RayGL translates these objects into POV-Ray as the objects they are meant to approximate. The POV-Ray objects will be completely smooth, instead of having the faceted appearance of the glu objects. While this helps to improve the realism of the scene, it may not be what the original program was intended to render. In some situations, a low number of facets may have been used to create a specific effect. For example, there are several ways to create a hexagonal nut; one method is to use a gluCylinder with 6 slices.

raygluDisk

void raygluDisk(GLUquadric *quad, GLdouble inner, GLdouble outer, GLint slices, GLint loops)

This function implements gluDisk, the disk is translated to POV-Ray as a smooth circular disc. The outer value defines the radius of the disc, and the inner value defines the radius of a hole in the disc. The disk is flat and infinitely thin in both OpenGL and POV-Ray. The slices and loops values are not used in the translation, the POV-Ray object is not faceted.

raygluSphere

void raygluSphere(GLUquadric *quad, GLdouble radius, GLint slices, GLint stacks)

This function implements gluSphere, the sphere is translated to POV-Ray as a smooth sphere object. The radius value defines the radius of the sphere. The slices and loops values are not used in the translation, the POV-Ray object is not faceted.

raygluCylinder

void raygluCylinder(GLUquadric *quad, GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint stacks)

This function implements gluCylinder, the cylinder is modeled as a smooth cone in POV-Ray. In both OpenGL and POV-Ray, the base and top values define the radius of the cylinder at both ends. If the radius given for one end is zero, then a cone is created. GluCylinders do not have a cap covering the ends of the cylinder; this effect is achieved in POV-Ray by adding the open keyword to the cylinder description. The slices and loops values are not used in the translation, the POV-Ray object is not faceted.

rayglutSolidSphere

void rayglutSolidSphere(GLdouble radius, GLint slices, GLint stacks)

This function implements glutSolidSphere, the sphere is translated to POV-Ray as a smooth sphere object. The radius value defines the radius of the sphere. The

slices and loops values are not used in the translation, the POV-Ray object is not faceted.

rayglutSolidSphere

void rayglutSolidSphere(GLdouble radius, GLint slices, GLint stacks)

This function implements glutSolidSphere, the sphere is translated to POV-Ray as a smooth sphere object. The radius value defines the radius of the sphere. The slices and loops values are not used in the translation, the POV-Ray object is not faceted.

rayglutSolidCube

void rayglutSolidCube(GLdouble size)

This function implements glutSolidCube, the cube is translated to POV-Ray as a box.

rayglutSolidCone

void rayglutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks)

This function implements glutSolidCone, the cone is modeled as a smooth cone in POV-Ray. GlutSolidCones do not have a cap covering the ends of the cone; this effect is achieved in POV-Ray by adding the open keyword to the cone description. The slices and stacks values are not used in the translation, the POV-Ray object is not faceted.

rayglutSolidTorus

void rayglutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings)

This function implements glutSolidTorus, the torus is modeled as a smooth torus in POV-Ray. Both OpenGL and POV-Ray define tori by a combination of the inner radius of the torus and the outer radius. Given a cross section of a torus defined in the xz plane and centered at the origin, the torus can be described as two radius's relative to a circular centerline which runs through the center of the solid portion of the torus. The centerline can be represented as a point rotated 360 degrees about the y-axis. The inner or minor radius is then the distance from the centerline to the nearest edge of the solid torus. The outer or major radius is the distance from the origin to the centerline. The nsides and rings values are not used in the translation; the POV-Ray object is not faceted.

Textures

While much of the data needed to translate a scene from OpenGL to POV-Ray can be gathered by a series of calls to glGet, the creation and assignment of textures warrants special attention. The following two functions are used in OpenGL to define texture images, the pass in the type of the texture, format, width and size in pixels, the data type of the texture, and a pointer to the texture data. For RayGL to associate textures in OpenGL with textures in POV-Ray, one of these two functions must be called for each texture used.

rayglTexImage2D

void rayglTexImage2D(GLenum target, GLint level, GLint internalFormat, GLint width, GLint height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)

This function captures glTexImage2D. It maps the currently bound texture name to an element in the image vector by comparing the pointer to the texture data with the pointer to texture data for each element in the image vector. On finding a match it updates that elements texName value and calls the toString for the texture. If no matching texture is found, a warning is printed to the console. The new value for texName is found by calling glGet, so this function is not glBegin/glEnd safe.

raygluBuild2DMipmaps

void raygluBuild2DMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height, GLenum format, GLenum type, const GLvoid *pixels)

This function captures gluBuild2DMipmaps. It maps the currently bound texture name to an element in the image vector by comparing the pointer to the texture data with the pointer to texture data for each element in the image vector. On finding a match it updates that elements texName value and calls the toString for the texture. If no matching texture is found, a warning is printed to the console. The new value for texName is found by calling glGet, so this function is not glBegin/glEnd safe.

Internal calls

These functions are not meant to be called by the used actively, or passively. They are helper functions that are internal to RayGL.

Attributes and coordinates

The following functions are used to get, display, and write the color, material, location, and texture of an object.

For each object that is to be translated from OpenGL to POV-Ray, getAtribs is called to get the attributes of the object before writing it to the SDL file. If the object is a gl primitive that is created with a glBegin/glEnd sequence, then getAtribs is called prior to glBegin. This has to be done prior to calling glBegin, this is because getAtribs makes multiple calls to glGet in order to determine the current state of OpenGL The OpenGL specification does not allow calls to glGet between a glBegin/glEnd sequence, while this may work on some hardware, on others accessing this data within the sequence can cause the program to become unstable. For some of the complex objects that will create multiple polygons in POV-Ray, writing out these values to the SDL file is a concern. In this situation, RayGL breaks the begin/end sequence into multiple sequences and makes the required calls in between sequences.

getAtribs

void getAtribs()

This function queries OpenGL for the current modelview matrix, color, material properties, and the index of the currently bound texture. It also checks to see if

any special clipping planes have been added to the scene. These values are stored in global variables as the current state of RayGL. It should be called prior to describing an object, and must be called outside of a glBegin/glEnd sequence.

stringAtribs

void stringAtribs(string sret)

This function writes a string representing the current texture, pigment, color, and modelview matrix into the input string variable. This is used primarily for debugging RayGL.

writeAtribs

void writeAtribs()

This function combines several RayGL variables to create a string representing the color, material, texture, and texture location transformations of an object. Depending on the code path taken due to the current state of RayGL, this can be any of the following POV-Ray formats,

texture{,

texture{pigment{imagemap{}}translate, rotate, scale},

texture{pigment{color{finish}}interior{media{emission}}.

Since getAtribs is used to populate the variables representing the current state of RayGL, calling this method prior to getAtribs will result in an error. Unlike getAtribs, his method does not interact with OpenGL, as a result it can be safely called from within a glBegin/glEnd sequence.

writeMatrix

void writeMatrix()

This function writes the current modelview matrix to the SDL file. It relies on getAtribs to populate the matrix. This function does not interact with OpenGL, so it is safe to call it during a glBegin/glEnd sequence.