RAYGL: An OpenGL to POVRAY API

Kris Zarns
Department of Computer Science
University of North Dakota
Box 9015 Streibel Hall
Grand Forks, ND 58202-9015
kzarns@cs.und.edu

Ronald Marsh, Ph.D.
Department of Computer Science
University of North Dakota
Box 9015 Streibel Hall
Grand Forks, ND 58202-9015
rmarsh@cs.und.edu

Abstract

The OpenGL raster graphics API is well known amongst computer graphics programmers. However, while raster graphics dominates the interactive computer graphics industry, these systems are not able to produce scenes with the realism required by ventures such as the movie industry. For markets that require high levels of realism but not online generation of the scene, raytracing has become popular. Raytracing applications use application specific scene description languages (SDL) to describe the objects in a scene. However, there is no accepted standard SDL and an open source library to convert OpenGL code to SDL code would allow programmers who are already familiar with OpenGL to smoothly transition between raster graphics and raytracing without the need to learn an application specific SDL. We present an OpenGL like API that will interface with OpenGL (and the OpenGL Utility Toolkit - GLUT) and that will write Persistence of Vision Raytracer (POVRay) scene files.

1. Introduction

OpenGL is one of the most popular interactive computer graphics API's in use today. As such, many computer scientists are familiar with it and many computer games are written using it. However, it uses a raster graphics engine to render the scenes and while this engine does have a lighting model which can be used to create lighting effects in real time, the lighting model is somewhat simplistic and incapable of reproducing the effects that a scene in the physical world would exhibit. This is due in part to OpenGLs use of the Gouraud shading model and not the more realistic Phong shading model. In addition, each polygon in an OpenGL scene is considered independently of each other when rendering the scene. In order to approximate reflections additional passes of the renderer must be made. However, the Phong shading model and multiple passes of the renderer are both inefficient and can become prohibitively computationally expensive.

An alternative to raster graphics rendering is raytracing. Raytracing is a method by which the proper illumination of a 3 dimensional scene is calculated by tracing a light ray from the viewing plane into the scene until it contacts an object in the scene. Depending on the properties of the object, the light ray is absorbed by the object, refracted through the object, reflected off the object, or a combination thereof. The process is repeated until the light ray contacts a light source or until an imposed limitation on the path length or number of reflections is met. Unlike the direct lighting effects of raster graphics, raytracing is able to account for most of the lighting effects that are exhibited in the real world. The result is that images that are rendered with raytracing algorithms are able to approach photorealism while images generated from raster graphics are only able to provide a rough approximation.

However, there are some advantages to using raster graphics over raytracing. One issue is ease of use. The OpenGL API is widely used and provides a defacto standard for raster graphics. There is no established standard SDL for raytracing. Each SDL is specific to the raytracing application it is used for. The result of this confusion has been the development of utilities to translate between one raytracer and another. Another issue is the cost of prototyping. Depending on the amount of detail represented in a scene, rendering a raytraced image can be very time consuming. It may be very useful to have a visual representation of the scene prior to committing to the rendering in full detail. For example, it may be very convenient to render the scene from multiple camera angles before deciding on which one to use. While some raytracers do provide this functionality, it is usually done by converting the scene into OpenGL.

Since there appears to be a need to allow the developer to rapidly prototype a scene and since the OpenGL API is a defacto standard for raster graphics, a standard that many graphics developers would already know, we decided to create RayGL, an API that will allow an OpenGL programmer the ability to generate POVRAY SDL files from the OpenGL code itself. When using RayGL, all the developer has to do is replace all OpenGL command prefixes “gl” with the text “raygl”. For example, a call to glVertex3f (…) would be replaced by rayglVertex3f(…). This is possible because the syntax of

functions in RayGL are identical to the syntax used in OpenGL. Currently, RayGL extends OpenGL by creating Persistence of Vision Raytracer (POVRAY) SDL files. However, because there are no plans to provide explicit support for POVRAY specific functions in RayGL, RayGL could be extended to work with other raytracers as well.

2. Previous Work

There has not been a lot of development in this area. We only know of two other projects that have worked in the area of adapting OpenGL code to raytracing. These are the OpenRT API and SvenGL. While both of these systems feature an OpenGL like syntax, neither of them support a direct conversion between existing OpenGL code and raytracing.

2.1 OpenRT

OpenRT is an expansion upon the Real-Time Ray-Tracing (RTRT) project developed at the University of Saarbruecken in 2001. The OpenRT application was first introduced by Wald et al. [2002], followed by a more detailed description of the API by Dietrich et al. [2003]. The API was developed to provide the interactive features of OpenGL along with the rendering capabilities of raytracing. OpenRT was written with high commodity equipment in mind, the system is meant to be run a network of high performance processors interconnected with Gigabit Ethernet. For example, when [Schmittler et al. 2004] implemented a raytraced version of the game Quake 3, they used a virtual CPU that mimicked a 30 GHz machine.

The OpenRT API provides a real time raytracing solution with a syntax that is similar to OpenGL. However, there are some key differences between OpenGL and OpenRT. OpenRT does not offer support for immediate mode rendering and the way in which OpenRT handles material properties is different. These differences are the result of a fundamental change in semantics between OpenGL and OpenRT. OpenGL functions as a state machine, changes in how the scene should be rendered are stored as changes in the state of the rendering pipeline. Because the changes are bound to the state machine and not objects within the scene, the changes are global. OpenRT binds changes to the specific objects being rendered, not the state of the renderer. As such, changes can be considered to be local to a specific object.

In the interest of efficiency, OpenRT does not support immediate mode rendering. In OpenGL, immediate mode rendering refers to the rasterization of triangles as they are specified. Because of the need to calculate reflections based on all of the objects in the scene, immediate mode is not a possibility for real time raytracing. Instead, OpenRT uses a concept similar to that of display lists. Display lists have the advantage that they are reusable within the program, and only need to be declared once. Rather than each part of an object being drawn individually, display lists allow for an entire object to be specified

and then drawn as a whole. While this can be more efficient than immediate mode rendering, OpenRT takes things a step further and implements the display lists using an object oriented approach. Unlike OpenRT, RayGL is meant to be used for offline raytracing. Issues of renderer efficiency will be left up to POVRay. RayGL code may be written using either immediate mode or retained mode. Objects drawn in immediate mode will act as expected in the OpenGL render, however the distinction between immediate mode and retained mode will be transparent to the SDL file. This is approach is taken so that the fine raytraced scene can consider every object in the scene for determining illumination.

The object oriented approach used in OpenRT allows changes in the rendering mode to be applied to specific objects. Conversely, in OpenGL, changes to the state of the OpenGL renderer will affect the output of all display lists rendered while the state is in effect. As a result, the programmer must be wary about the global state of the OpenGL renderer when porting code from OpenGL to OpenRT. Because changes in the OpenGL state are global, care needs to be taken to preserve the state of the OpenGL machine when translating OpenGL code to a system that keeps track of this information locally. If a state persists over a large portion of OpenGL code, that state needs to be preserved for each of the objects when they are translated to OpenRT in order to faithfully reproduce the original application. This may become an issue when declaring complex scenes as there is no guarantee that a state change will occur near the object that it affects. However, in RayGL this issue is not a concern as the current state of OpenGL is written to the SDL file by RayGL for each object as it is declared.

OpenRT also handles materials differently than OpenGL. Instead of implementing material properties as a means of modifying lighting effects, OpenRT uses an independent shading language called OpenSRT. This shading language is similar in concept to the Stanford shader API and the shading language that was implemented for OpenGL 2.0. The independent shading language allows each program to customize the lighting effects available in a scene. Using this approach the application specifies the material features in the form of shader objects, these objects can then be bound to geometric objects. As a result of this binding, changes made to the shader appear local to all objects it is bound to. Because OpenSRT is independent of OpenRT, it is also possible to use other shading languages with OpenRT.

Finally, despite the word Open in its name, OpenRT is not actually open source. OpenRT has been developed as a commercial product. While there is a freely available academic version of OpenRT, it is a limited version of the software and does not support clusters and has a limitation imposed on the complexity of scenes drawn. Conversely, RayGL and POVRay are both open source and can be implemented on a cluster architecture.

2.2 SvenGL

SvenGL refers to the coupling of OpenGL with the TOpenGLApp class developed by

Maerivoet [2002]. TOpenGLApp is a combination of programs for window management, camera management, projection management, provides primitives for common shapes, texture management, and a raytracing engine. The system acts as an intermediary between the programmer and GLUT. To simplify the API, SvenGL does not support all of the functions used in GLUT.

SvenGL provides support for OpenGL rendering of code written in standard OpenGL as well as code developed with the features of SvenGL. It also contains a separate SDL. Code written for this SDL is rendered with both OpenGL and the included raytracing engine. Rendering a scene with the SDL creates two output files, one rendered in OpenGL and the other with the raytracing engine. However the OpenGL rendering does not support SDL textures. Because of its object oriented design, it is possible to make alterations directly to the SDL to provide support for additional features.

Similar to the way in which SvenGL produces both OpenGL rendered and raytraced images when using its SDL, RayGL provides outputs for OpenGL and POVRay simultaneously. The important difference here is that SvenGL takes code that is written for high quality display with the raytracing engine and then displays a low quality OpenGL render of it along with the high quality render. Regardless of the results of the OpenGL render, CPU time is still spent generating the raytraced render. Whereas RayGL first provides a low quality render based on OpenGL code and then allows the user to determine if a high quality render should be committed with the raytracer.

3. RayGL

RayGL is a utility for converting code written in OpenGL to code the can also generate raytracing SDL files. While the current version of RayGL only generates POVRay SDL files, it is possible to extend RayGL to generate other raytracing SDL files as well. This is possible because RayGL focuses on providing a direct translation from OpenGL to the raytracing SDL, not on offering raytracing specific features. POVRay was chosen as the raytracing backend to RayGL for three reasons. First, POVRay is open source. Because of this, it lends itself well to both modification of the source and the creation of third party utilities to interact with it. Second, the POVRay SDL is well documented. Extensive documentation on how to write for the SDL is provided on the POVRay website. Considerable support for POVRay is also offered in the form of mailing lists and forums. Finally, POVRay is available for Windows, Mac OS, Mac OS X, and i86 Linux.

Similar to the way in which SvenGL produces both OpenGL rendered and raytraced images when using its SDL, RayGL provides outputs for OpenGL and POVRay simultaneously. The OpenGL output displays to the screen in the same way a normal OpenGL application would. However, with each frame that is displayed in OpenGL, a POVRay SDL file is also written. The SDL file can then be rendered using POVRay at any time. If there is some issue with the scene displayed by the OpenGL rendering, then corrections

can be made to the code without dedicating the time to render the POVRay output. In terms of RayGL, a frame is defined as any time the front and back buffers are swapped using glutSwapBuffers(). Support for single buffered systems is still under consideration. RayGL can use either GLUT or FreeGLUT for handling window management when working with the OpenGL renderer. Because both these API's and POVRay provide geometric objects such as spheres and cones, there are plans to implement code generation for these objects as well as the basic OpenGL primitives. Modifying the RayGL code to support other window management API's should not be a problem as RayGL primarily relies upon them for determining when a new frame needs to be rendered by listening for a call to swapbuffers.

Because OpenGL acts as a state machine, RayGL must also keep careful track of the state of the OpenGL renderer when deciding how to write a scene using the SDL. As OpenGL primitives are created for rendering, they are translated to the appropriate primitives in the SDL. With each primitive declared in SDL, the current state of OpenGL is also written to the file. This is required because of the way in which the SDL is structured. Instead of a state machine approach like OpenGL which focuses on the state of the renderer, POVRay encapsulates all of the information needed to draw its objects within the object itself. The advantage to this approach is that the link between the properties that affect the way the object is drawn and the declaration of the shape of the object is much more obvious. An example of this is the way in which the two systems handle the coordinate matrix used to represent the objects in three dimensions. OpenGL keeps track of the current matrix by using a stack. There are two types of operations that can be performed on the current matrix, either stack management or transformations. By careful stack management the programmer can define different matrices for each of the objects to be drawn by placing them at different levels within the stack. Since the current matrix represents the state of OpenGL, when determining where to render objects, the current matrix is used to determine how to place the objects within the scene and any affine transformation (ie: translation, rotation, scale...) performed alters the current matrix. Therefore, transformations applied to the current matrix will affect the way in which all following objects are drawn unless the current matrix is modified or replaced. Because POVRay takes a more object oriented programming approach to defining geometric objects, the transformation matrix to be used for drawing is specified independently for each geometric object. As a result, when translating OpenGL primative objects to the POVRay SDL, RayGL retrieves the current transformation matrix from OpenGL for each object that is specified in the SDL file.

Another issue regarding the differences between OpenGL and POVRay is that they use different coordinate systems. OpenGL uses a righthanded coordinate system complete with a righthanded rotation scheme. That is to say, the positive X axis points to the right of the screen, the positive Y axis points to the top of the screen, and the positive Z axis points out of the screen, however POVRay uses a left handed coordinate system. With a left handed system the X and Y axis are defined in the same way as above, however the positive Z axis now points into the screen. As far as rotation is considered, right handed rotation causes the direction of positive rotation to be defined as counter clockwise about the positive direction in which the axis of rotation is defined. Lefthanded systems rotate

clockwise about the positive direction in which the axis is defined. The first attempt to correct for these issues involved performing operations on the transformation matrix retrieved from OpenGL for each object. The angle values specified opposite each other about the diagonal were swapped, however while this solved the rotation issues for x and y it did not account for the fact that POVray was using both a lefthanded rotation and a lefthanded coordinate system. Upon further review of the POVRay documentation another, simpler, method involving a change in the camera system was implemented. POVRay allows for the vectors specifying the coordinate system of its camera to be redefined and by modifying the up and right vectors used to describe the aspect ratio of the screen we are able to change POVRay to a righthanded coordinate system.

Parts of RayGL are still in the design phase. While the creation of geometric primitives is currently implemented, the method by which lighting and material properties will be handled is still being determined. Similarities between the way in which OpenGL and POVRay declare surfaces and light sources suggest that implementing these features should be fairly straightforward. However, there may be issues with the design that will only be apparent once these features have been implemented.

As of the current design, OpenGL material properties will be translated to the SDL file in terms of the Media, Interior, and Finish texture attributes given in POVRay. The glMaterial function will determine part of the lighting equation for the surface it is applied to. This matches well with the POVRay notion of texture attributes. POVRay uses textures to determine the material properties of a surface. These textures are able to adjust the normal vectors of a surface, its color, and reflective properties. Media, Interior, and Finish are subsections of texture that control the reflective properties. Because glMaterial can take FRONT, BACK, or FRONT_AND_BACK as an argument, Front will be assumed to be the outside of an object and will correspond to changes in Media and Finish. Material properties assigned to BACK will adjust the Interior properties of the surface.

Light sources in OpenGL have the following parameters; ambient, diffuse, specular, position, spot_direction, spot_exponent, spot_cutoff, constant_attenuation, linear_attenuation, or quadratic_attenuation. Light sources specified by glLight are modeled by creating corresponding light sources in POVRay. Ambient light is adjusted by modifying the ambient light parameter in POVRay's global settings. For diffuse light, point lights are used. Specular lights are handled by creating a spotlight with the appropriate position and direction properties. Spot _exponent and spot_cutoff are modeled with the tightness and radius settings for spotlights.

5. Example Code

The following is a snippet of OpenGL code rewritten to use the RayGL library and the resulting SDL code it generates. The code generates 3 sides of a box using polygons,

then rotates the polygons 45 degrees about X and Y, and translates it 45 units into the X, Y, and Z directions. Both systems use orthographic project in this example. Due to the simplicity of the scene, the output from both renderers is nearly identical.

```
int angle =45 ;
void subWindowDisplay(void){
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
//naming an output file, and setup for the camera location
//code written between rayglFrameBegin and rayglFrameEnd
//will be translated to one SDL file.
rayglFrameBegin
("testout.pov",320,240,1000,320,240,0,0,0,50);

glLoadIdentity();
glTranslatef(45,45,45);
glRotatef(angle,1.0,-1.0,0.0);
//front face
rayglColor3f(1.0,1.0,1.0);
rayglBegin(GL_POLYGON);
 rayglVertex3i(-20,-20,20);
 rayglVertex3i(20,-20,20);
 rayglVertex3i(20,20,20);
 rayglVertex3i(-20,20,20);
rayglEnd();
//right face
rayglColor3f(1.0,0.0,0.0);
rayglBegin(GL_POLYGON);
 rayglVertex3i(20,-20,20);
 rayglVertex3i(20,-20,-20);
 rayglVertex3i(20,20,-20);
 rayglVertex3i(20,20,20);
rayglEnd();
//top face
rayglColor3f(0.0,1.0,0.0);
rayglBegin(GL_POLYGON);
 rayglVertex3i(-20,20,20);
 rayglVertex3i(20,20,20);
 rayglVertex3i(20,20,-20);
 rayglVertex3i(-20,20,-20);
rayglEnd();
glutSwapBuffers();
rayglFrameEnd();
}
```

POVRay SDL code

global_settings {ambient_light rgb <1,1,1>}#declare cam_locx = 320;

```
#declare cam_locy = 240;
#declare cam_locz = 1000;
#declare cam_lookx = 320;
#declare cam_looky = 240;
#declare cam_lookz = 0;
#declare cam_rx = 0;
#declare cam_ry = 0;
#declare cam_rz = 0;
camera {
  location <cam_locx, cam_locy, cam_locz>
  up <0,1,0>
  right  <-1.33,0,0>
  rotate <cam_rx, cam_ry, cam_rz>
  look_at <cam_lookx, cam_looky, cam_lookz>
}
light_source {
  <0, 0, 50>
  color rgb <1, 1, 1>
  parallel
  point_at <0,0,0>
}

polygon{
      5,
<-20,-20,20>
<20,-20,20>
<20,20,20>
<-20,20,20>
<-20,-20,20>
  texture {
    pigment{color rgb <1.000000,1.000000,1.000000>}
  }
      matrix <0.853553,-0.146447,0.500000,
          -0.146447,0.853553,0.500000,
          -0.500000,-0.500000,0.707107,
          45.000000,45.000000,45.000000>
}
polygon{
      5,
<20,-20,20>
<20,-20,-20>
<20,20,-20>
<20,20,20>
<20,-20,20>
  texture {
    pigment{color rgb <1.000000,0.000000,0.000000>}
```

```
  }
      matrix <0.853553,-0.146447,0.500000,
          -0.146447,0.853553,0.500000,
          -0.500000,-0.500000,0.707107,
          45.000000,45.000000,45.000000>
}
polygon{
      5,
<-20,20,20>
<20,20,20>
<20,20,-20>
<-20,20,-20>
<-20,20,20>
  texture {
    pigment{color rgb <0.000000,1.000000,0.000000>}
  }
      matrix <0.853553,-0.146447,0.500000,
          -0.146447,0.853553,0.500000,
          -0.500000,-0.500000,0.707107,
          45.000000,45.000000,45.000000>
}
```

5. Conclusion

Compared to the limited lighting models of raster graphics approaches, raytracing provides complex lighting models and a means to render high quality photorealistic scenes. However, OpenGL is a standard graphics API that is commonly known among computer graphics programmers. To provide both ease of use and advanced lighting effects RayGL combines the familiar API used by OpenGL with the rendering features of the POVRAY raytracer. RayGL allows existing programs written for the OpenGL renderer to be ported to a raytracing engine with minimal alteration to the original code. Because RayGL translates OpenGL commands into a SDL, programmers who are familiar with the OpenGL API can create raytraced images without the need to learn an application specific SDL. Furthermore, since RayGL creates each frame as an independent SDL code file, it can be easily adapted for use with a computer cluster where each file is distributed to a different node of the cluster and rendered independently in parallel.

6. Future work

RayGL is implemented using a combination of POVRay, OpenGL, and GLUT/FreeGLUT, the next step is to expand support for RayGL to other raytracing and window management systems. Because the RayGL API is limited to functionality present

in OpenGL and GLUT/FreeGLUT, development of a raytracing engine exclusively for RayGL will also be investigated.

References

Dietrich, A., Wald, I. And Slusallek, P. 2003. The OpenRT Application Programming Interface - Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, Eurographics Association, Darmstadt, Germany, April 1-2, 2003, pages 23-31

Maerivoet, S. 2001-2002 Advanced Computer Graphics Using OpenGL. http://svengl.dyns.cx

Marion, J., Thornton S. 1995. *Classical Dynamics of Particles and Systems*. Fourth ed. Harcourt College Publishers

Schmittler, J., Dahmen, T., Pohl, D., Vogelgesan, C., and Slusallek, P. *Ray Tracing For Current And Future Games,* Published at 34. Jahrestagung det Geselleschaft für Informatik 2004

Shreiner, D. 2004. *OpenGL Reference Manual, The Official Reference Document to OpenGL, Version 1.4*. Fourth ed. Addison-Wesley

Shreiner, D., Woo, M., Neider, J., and Davis, T. 2006. *OpenGL Programming Guide, The Official Guide to Learning OpenGL, Version 2*. Fifth ed. Addison-Wesley

Wald, I., Benthin, C., and Slusallek, P. *OpenRT - A Scalable and Flexible Rendering Engine for Interactive 3D Graphics,* Technical report, TR-2002-01, Saarland University