Animation techniques: A Comparison

Ronald Marsh, Ph.D.
Computer Science Department
University of North Dakota
Grand Forks, ND
rmarsh@cs.und.edu

Narda Hamilton
Computer Science
University of North Dakota
Grand Forks, ND
Narda.Hamilton@und.edu

Abstract

The OpenGL raster graphics Application Programming Interface *(*API*)* is well known amongst computer graphics programmers. The OpenGL Shading Language (GLSL) is not as well known. Even lesser known is the OpenGL, POV-Ray and Raygl mix (developed at the University of Noth Dakota). 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 such markets, raytracing is 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. Thus, several years ago we developed an OpenGL like API that would interface with OpenGL and the OpenGL Utility Toolkit (GLUT) and that would write Persistence of Vision Raytracer (POVRay) scene files. However, raytracing is slow; too slow for real-time animations that require near photo realism. One of the first shading languages developed was RenderMan (1988); it used the CPU to generate cinematic effects. Since then, Microsoft's High-Level Shading Language (HLSL) and OpenGL's OpenGL Shading Language (GLSL) have been developed for real-time use on a GPU. The goal of this work was to compare the three animation techniques (OpenGL, OpenGL with RayGL and POV-Ray, and GLSL) with regards to quality of output, amount and difficulty of coding required. We are particularly interested in whether or not the output is 'Hollywood' quality or just good enough for gaming and how we might apply the lessons learned to our graphics courses. All three techniques display a scene with stationary platform with a bouncing ball on top.

1 Introduction

This project seeks to compare three animation techniques namely OpenGL [1], POV-Ray [2] with the use of RayGL [3, 4], and GLSL. OpenGL is a cross platform, cross-language, low-level procedural API used for creating 2D and 3D graphics where the programmer states the exact steps needed to render a scene. While the OpenGL API can be used to create 3D scenes, OpenGL alone is not able to produce scenes with the realism required by ventures such as the movie industry. Thus we created RayGL, an OpenGL like API that allows the OpenGL programmer to easily generate Persistence of Vision Raytracer (POV-Ray) SDL files from OpenGL and OpenGL Utility Toolkit (GLUT) programs. RAYGL allows programmers to easily extend their OpenGL code such that realistic animations are possible.

We compare the three mediums by creating a scene with a ball bouncing on top of a stationary platform. We seek to compare these techniques in terms of quality of output and difficulty of coding. The results of these comparisons may help determine what area or purpose each technique may or may not be useful for.

2 Background

2.1 OpenGL

OpenGL is a low-level, procedural API where the programmer is expected to state explicitly the exact steps needed to render the scene, as opposed to descriptive APIs where the programmer describes the scene and allow the library to manage the rendering. Programmers are required to be familiar of the graphics pipeline but also allow enough flexibility to implement their own rendering algorithm.

Created by Silicon Graphics (SGI) in the 1990s, the Integrated Raster Imaging System Graphics Library (IRIS GL) was considered the de facto industry standard for rendering 3D graphics as it was seen as ‘easier to use’. Standardized access to hardware was instantiated by OpenGL, which forced device drivers to become developed by hardware manufacturers and allotted windowing functions to the underlying operating system. This allowed software developers to have a higher level platform for 3D software development. 1992 led to the creation of the OpenGL Architecture Review Board (ARB) by SGI, the group of companies that would maintain and expand the OpenGL specification for years to come.

The Open Graphics Library (OpenGL) was first created as an alternative to IRIS GL. OpenGL is a cross platform, cross-language API used for creating 2D and 3D computer graphics (OpenGL was mainly created for video games). Although OpenGL was initially similar to IRIS GL, the main disadvantage of IRIS GL was that it only provided access to features supported by the underlying hardware; if the graphics hardware did not support a feature, then the application could not use it. To counter this problem, OpenGL provided support in software for unsupported hardware features which allowed for advanced graphics usage on less capable systems.

OpenGL provides only rendering functions; designed for graphic output only. There is no concept of windowing systems, printing to screen, audio, or other input devices by the core. Although this may seem restrictive at first, it allows the rendering code to run independent of the operating system it is running on, thus allowing for cross-platform development. To allow a smooth interaction with the host system, add-on APIs create some integration with the windowing system. These include:

- GLX-X11 – OpenGL Extension to the X Window System.
- WGL (Wiggle) – Windowing system interface to the Microsoft Windows implementation of OpenGL.
- CGL (Core OpenGL) – Apple Inc.'s Macintosh Quartz Windowing System interface to the MAC OS X.

Finally, some open source cross-platform toolkits GTK+ (Gimp Toolkit) and Qt (officially pronounced as cute) include widgets to embed OpenGL contents.

2.2 POV-Ray

The Persistence of Vision Ray-tracer (POV-Ray) is a freeware ray-tracing program for multiple platforms. Ray-tracing is a rendering technique which simulates how rays of light travel in the real world in order to produce a photo-realistic rendering of a scene.

First, the location of surfaces, textures, and properties of objects along with their interior (if transparent) and any atmospheric media such as fog, light sources, and the camera are specified by the user.

Secondly, ray-tracing starts with a simulated camera and traces light rays backwards out into the scene. Rays originating from the camera are checked to see if there exists any intersection with any of the objects (via every pixel in the image) in the scene. The color of the surface at the point where the object is hit by a ray is calculated, thus determining the amount of light coming from the source by sending the rays backwards. These are called "shadow rays" which are tested to evaluate whether the surface point lies in a shadow or not. To establish the amount of reflected or refracted light in the final surface color, if the surface is reflective or transparent, new rays are set up and traced. A lot of additional rays must be traced into the scene for each pixel due to special features like inter-diffuse reflection, atmospheric effects and area lights. By tracing backwards, we prevent the unnecessary cost of calculating rays that never fall on the camera. The most recent version of the software includes the following:

- Library of ready-made scenes, textures, and objects.
- Several kinds of light sources.
- Atmospheric effects such as fog and media (smoke, clouds).
- Image format support for rendered output and textures, including PNG , JPEG, TGA, amongst others,

Mathematical functions describe all POV-Ray primitive objects as opposed to many computer modeling packages which use triangle meshes to build objects. Thus, POV-Ray has certain advantages over other rendering modeling systems, including being more accurate and simpler. Also a sphere in POV-Ray is described with a center and a radius as opposed to many polygons. However, script-based modeling is not always practical for realistic characters and complex objects such cars. Finally, POV-Ray is solely a renderer; it does not include a modeling feature of its own. Users have developed a variety of modeling software for POV-Ray while others import and export its data structure.

2.3 RayGL

RayGL was formally created in 2006 by Kris Zarns, a UND computer science graduate student. RayGL creates POV-Ray Scene Description Language (SDL) files from Glut and OpenGL code using two approaches:

- Man in the Middle – intercepts OpenGL calls, builds SDL to represent them and run OpenGL calls.
- State Machine – queries the current state of OpenGL and inserts that information into the SDL code.

Included in RayGL are raygl.h and raygldefs.h. Users must then call rayglFrameBegin at the beginning of each frame, rayglFrameEnd at the end of each frame, and setRenderPov to toggle SDL generation on/off.

2.4 Shaders

As Graphics Processing Units (GPUs) evolved it became possible to program these GPUs by defining special shading functions in their API. A shader is a program that describes the characteristics of a pixel or vertex.

There three most common types of shaders are vertex shaders, geometry shaders and fragment shaders.Vertex shaders run once for each vertex given to the graphics processor. Their main purpose is to transform each vertex's 3D position in virtual space to the 2D coordinates presented on the screen. Vertex shaders can manipulate properties such as position, color, and texture coordinate, but they cannot create new vertices. Geometry shaders execute after the vertex shader and are used to add and remove vertices from a mesh. Fragment shaders, also called pixel shaders, are used to compute the color attributes of a pixel. This shader is used to apply a lighting value, translucency, bump mapping, and altering the Z-buffer.

2.5 OpenGL Shading Language (GLSL)

GLSL, also known as *GLslang*, is a high level shading language that uses the C Programming language. GLSL is intended to provide a more intuitive means of programming the GPU. Introduced as an extension of OpenGL 1.4 and introduced in

OpenGL 2.0 by the OpenGL ARB, GLSL was developed to allow developers to program points in an OpenGL pipeline. Benefits of GLSL include:

- Cross-platform compatibility across multiple Operating Systems.
- The GLSL driver is part of each vendor's hardware, thus allowing optimized code for each graphics card architecture.
- Shaders that are written can be used on any hardware vendor's graphics card that supports GLSL.
- Operators with the exception of pointers are supported by GLSL. Functions and control structures such as loops, if/else, while, do-while, break are used in GLSL. This allows for optimization at the hardware level of these built-in functions if necessary.

3 Animation techniques Used

3.1 OpenGL

In order to guarantee portability across the platforms, we kept the animation very simple; a bouncing ball on a flat surface. Our initial technique, OpenGL, had the following specifications:

- Double buffering.
- Implemented lighting modes such as gl_diffuse, specular, normalize, and blend.
- Used GL_PROJECTION, GL_MODELVIEW for viewing.

This resulted in OpenGL showing a white ball bouncing on a black surface with a red background, shown in figure 1.

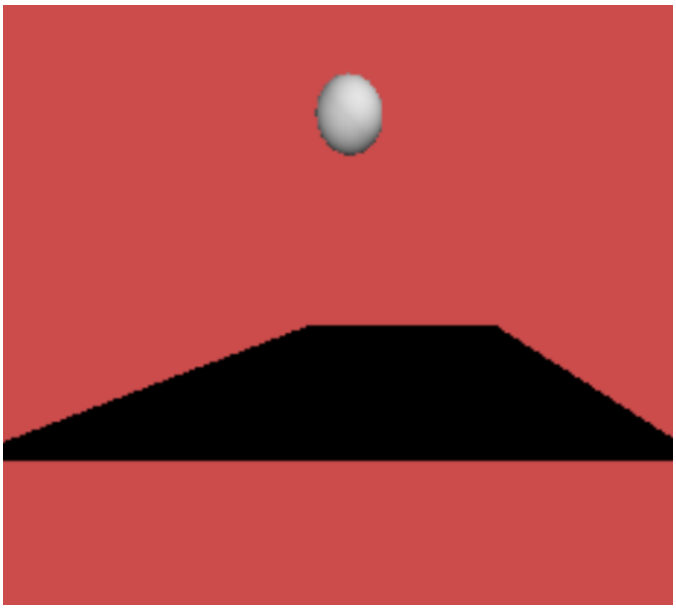

Figure 1. Initial OpenGL scene.

3.2 OpenGL, RayGL, and POV-Ray

We then modified the code to use RayGL and POV-Ray; this included the following code modifications:

```
//included in header
#include "raygl.h"
#include "raygldefs.h"

//before calling the drawball() function
setRenderPov( true );
char filenameForRayGL[] = "./RAYGLIMAGES2/image";
rayglFrameBegin(filenameForRayGL);

//after calling drawball() function
rayglFrameEnd();
```

In order to view rendered images, we then created POV-Ray SDF files using:

```
/usr/local/bin/povray +L/usr/local/share/povray-3.6/include +A0.3 +W640 +H480
+I$1 +FN -D +O$2
```

Where $1 represents the name of the input file (*.pov) and $2 represents the output filename (*.png).

The results of this approach also showed a white ball bouncing on a black surface with a red background. But this technique also showed the ball casting a shadow on the surface it was bouncing on.

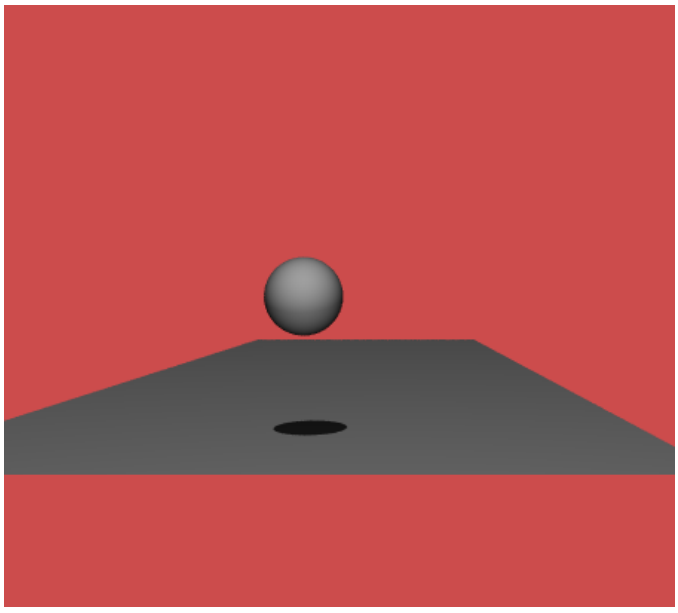

Figure 2. OpenGL/RayGL/POV-Ray scene.

Difficulties encountered with using RayGL/POV-RAY revolved around having to install RayGL and POV-Ray and the use of RayGL itself. The Makefile was recompiled where the two objects within the Makefile were compiled separately; thus discovering that the Raygl files needed to be compiled on their own before compiling the entire project together. This resolved the problems.

3.3 C# and HLSL

We then modified the code to use a shader language; this included the use of two shaders, a vertex shader and a pixelshader:

- The vertex shader returns the transformed position, light vector, normal vector and an eye vector for each vertex.
- The pixel shader normalized some parameters to ease the cost of computations and returns the color of each pixel.

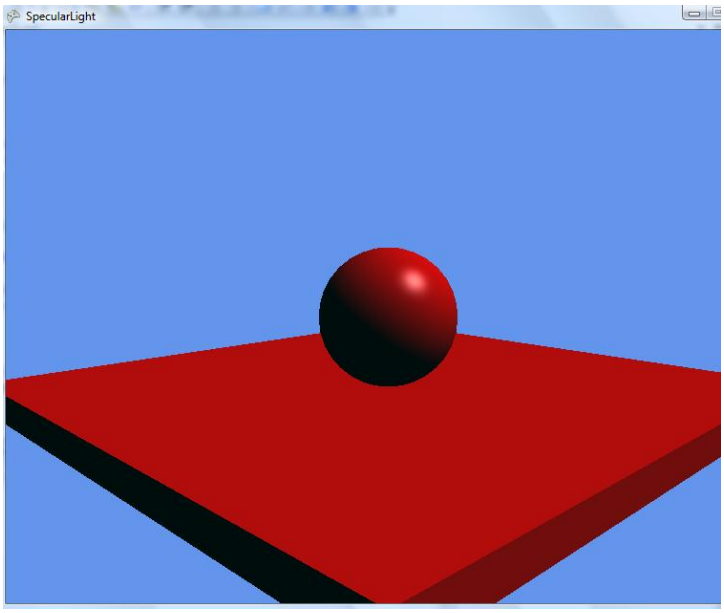

Figure 3. Shader scene.

Due to the many difficulties we encountered in using GLSL (see below), the scene shown in figure is actually Microsoft C#/XNA code. The shader code was preexisting code downloaded from digitseven [5] which was modified to ensure that all three techniques used the same objects in the animation. The code was modified by replacing the existing models, changing lighting directions and using a box class to create the vertices for the surface; this was done to closely recreate the scene to be like the previous scenes of the OpenGL and RayGL implementation with a ball and a surface.

The shaders accommodated for multiple passes within the graphics pipeline which allowed for the specular lighting effects to be shown on the ball but no reflection on the surface.

Difficulties encountered with using GLSL revolved around trying to understand exactly how the code works from its core in order to implement it. This was not possible due to the time constraint of the project. Another option was to find GLSL code that already existed, and then tailor it to meet the needs of the current project. This proved to be difficult as the plethora of code found was overwhelming and aided in confusion; and over 90% of code found had multiple errors which could not be deciphered. However, C#/XNA examples and well documented tutorials are commonplace.

4 Conclusion

This work was implemented using the C++ programming language. The OpenGL component on its own took no more than an hour to implement with around 154 lines of code. The extra lines required to implement RayGL took no more than around ten minutes which included the adjustment to the Makefile to accommodate for Raygl. Additional files in support of Raygl were added to the folder for execution; the C++, header and object files.

In terms of the quality of the output from the three techniques, the OpenGL produced the poorest quality image, and this was no surprise as OpenGL uses Gouraud shading (called "smooth" shading by OpenGL), which calculates lighting values only at the vertexes of polygons and interpolates them over the rest of the polygon resulting in unrealistic lighting. POV-Ray and C#/XNA shader produced images of similar quality for our very simple scene and we suspect that differences would have been more apparent should a very complex scene be used. However, when reviewing the animations we found that OpenGL produced animations that had no flickering or distortion. Obviously, the POV-Ray also had no flickering or distortion and shadows were created as part of the rendering process. Of course POV-Ray is not a real-time animation approach.

Shadowmapping requires the scene to be rendered from the light's perspective and then saving the resulting Z-buffer. Then, when rendering the scene from the camera's point of view the distance between the pixel being drawn and the light is compared to the depth value retrieved from the saved Z-buffer to determine whether that pixel is in the shadow or not. We simply ran out of time while trying to master the many intricate details required to implement a shader supporting shadows.

References

[1] OpenGL, http://www.opengl.org/. Last accessed March 1, 2011.

[2] The Persistence of Vision Raytracer (PovRAY), http://www.povray.org/. Last accessed March 1, 2011.

[3] K. Zarns and R. Marsh, "RAYGL: An OpenGL to POVRAY API," in *MICS 2006: Proceedings of the Midwest Instruction and Computing Symposium*, Mt. Pleasant, IA, 2006.

[4] R. Marsh and K. Zarns, “RAYGL: An OpenGL to POV-Ray API,” in *CATE 08 - Computers and Advanced Technology in Education,* Crete Greece, 2008.

[5] Digit 7, http://digitseven.com/shadertutorials.aspx. Last accessed March 15, 2011.