Creating an OpenGL Deferred Rendering Implementation

Andrew Dunetz Eugene Umlor

Abstract

We implement and demonstrate an industry standard method of rendering. The method is capable of rendering large numbers of lights in real time. Starting with a discussion of previous comparable research and methods, we describe the rendering pipeline we implemented. In addition, we provide some commentary on the kinds of tools and support utilities that are helpful or required to develop such a pipeline in practice. We demonstrate the validity of the method and discuss the necessities of such an implementation. We conclude that deferred rendering is an effective and practical procedure for the rendering of large scenes with complex lighting and geometry.

Introduction

There is immense motivation to render complex scenes in real time. Applications like video games and other interactive mediums must be able to perform renders at a sufficient rate that they are responsive on a human moment to moment scale. Graphical Processing Units (GPUs) have long provided the ability to make this task parallel and thus more efficient.

As opposed to deferred rendering, forward rendering simply draws the triangles the geometry of the scene in unsorted sequence, using the depth buffer to make sure the items closest to the camera are visible in the scene. To light scenes in a forward rendering pipeline, every light is rendered against every piece of geometry in the scene. While distance based culling can prevent this, it is still inefficient.

Deferred Rendering decouples the lighting sources from the geometry of the scene and thereby dramatically reduces the performance cost. The technique is popular not only thanks to the tremendous improvement in computing time, but also the variety of post processing effects which are trivial to implement as part of the pipeline.

This style of rendering is not new, and has become the standard method of high performance graphics rendering in the game industry. In part, this is because of the impressive flexibility of such pipelines.

Background/Related Work

Deferred shading was first developed by Michael Deering et al. in his much-cited SIGGRAPH 88' paper. His team separated the process of rendering lighting from rendering geometry, by first rasterizing all geometry, and then performing a second lighting pass using a phong model shader over the scene. This scheme resulted in large efficiency gains over prior forward rendering pipelines. It was later popularized by Saito and Takahashi in 1990, with the introduction of geometry buffers, or G-buffers.

G-buffers are data structures that store a single attribute about the rasterized geometry. Fragments are rendered to several G-buffers simultaneously during the first pass of the deferred pipeline, which allows the shaders in the lighting pass to make accurate contribution calculations by combining each of the buffers generated. G-buffers are not limited to physical properties of the geometries they represent, such as albedo or reflectance, but can store any attribute for use in further shaders, including useful variables for artistic post-processing.

Our lighting scheme is based heavily on Thaler's implementation of light volumes (2010). He describes a system whereby point lights are represented as a bounding box around the sphere that the light will reach. After a certain distance, the light contribution from each point light source will be so insubstantial, that significant speedup can be achieved by simply not calculating it. Light volumes are stored as the geometry of the bounding box, and a second set of G-buffers are generated based on the lighting information. The polygons that make up each light volume are then rendered as geometry, but shaded so that they look like the existing scene, with the contribution from the represented light. Thaler also describes a method of represented other types of lights as differently shaped light volumes, with a spotlight represented as a cone, and an area light represented as a screen-spanning quad.

Algorithm

Deferred rendering, or deferred shading, can be split into 2 stages. The decoupling of these 2 stages is the primary difference of the technique and is the source of most of its advantages.

The first step of the algorithm is to perform an initial render pass on the scene. Unlike forward rendering, this pass is output to multiple textures, which store the information about each pixel that ends up on the screen. This set of output textures contains the information that will be necessary for computing the final color of the pixel, once all the lights are factored in and drawn.

The textures typically contain a range of information. To create a complete scene, textures containing the material color, the depth from the camera, and the normal are created and passed out of the shader for later use. Additional information might be the specular component, and other information about the material and the way that it reflects light.

Because this part of the process is merely collecting data, the shader is extremely simple. It needs to read in the relevant values for a given fragment, and write them out to the framebuffer with depth testing enabled.

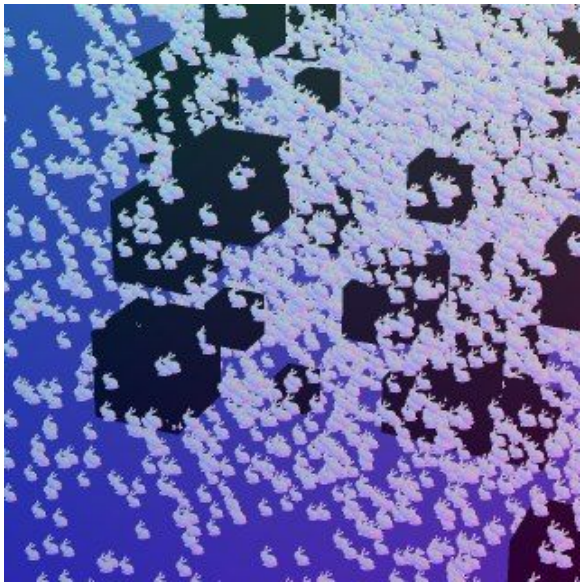

Figure 1: This demonstration scene includes a large number of geometry objects as well as the volumes that represent the lights. The black boxes are the light volumes, but drawn normally, instead of as lights.

At this point, the second part of the pipeline begins. The textures that were used as output from the last pass are bound to the shaders of the next part of the pipeline. The specific details of this part of the pipeline are heavily dependent of the implementation.

In general, the textures are drawn to a quad that is placed in front of the camera. While this occurs, the lights are passed to the shader and applied to the fragments being drawn. Since the positional data has already been filtered by a depth buffer in the previous step of the pipeline, every single fragment will be part of the final image.

During this part of the process, post-processing effects can be applied easily using the abundance of information available.

Implementation in OpenGL and C++

Our implementation closely follow the algorithm description while making several more unusual choices for the sake of development time. Because the demonstration of a deferred renderer's utility requires relatively complex scenes, we wrote tools for the automatic generation of those scenes in python. These python scripts produce input files which our program parses to generate the scenes. These test cases were used to calculate the performance speedup of our algorithm.

These input files have the ability to specify color, matching geometry files, position, and scale. They allow each object to have any number of time steps, so that a scene can be dynamic and animating. While parsing these files our program produces a list of required geometry that contains no duplicates, in addition to storing all of the per object information. These files also provide the same types of utilities for adding lights to a scene.

After parsing these files, initialization takes place. Uniforms, bufferIDs, and other standard OpenGL objects are setup. It also prepares these objects for rendering. In addition, glGenFramebuffer, and glBindFramebuffer are used to stop the GPU from rendering to the screen, and instead render to a frame buffer that can be output to a texture. In addition, the program uses OpenGL to create and bind textures that can be written to. Finally, the FBOs and VAOs relevant to both the geometry and the lights are generated, bound, and provided with data.

At render time, all of the scene geometry, not the lights, were passed into the first pass shader. The raw geometry was drawn as

materials, normals, world position and so on out to textures. In our implementation, we chose to write out world positions instead of depth, because the math is trivial. Most implementations use just depth, and then use that to find the world position of the fragment within the fragment shader.

At this point, the intermediary framebuffer is unbound, which tells the GPU to draw to the screen again. The second pass begins. During this section every single light is drawn into the scene. It should be mentioned that our implementation provides geometry for lights. Each light has a bounding box associated with it, and adding the light to the scene is the process of drawing the bounding box. When a given light bounding box is drawn the light center, color, and intensity are also sent in via uniforms. Every fragment being drawn to takes in the world position (p), the normal (n), and the material color(m). These pieces of information are used to generate an output color using a very simple lighting model. Where l is the location of the light, i is the light's radius, and d is the distance from the fragment to the light:

$$color = (1 - (d/i) * dot(normalize(l - p), n) * m$$

As presented, the maximum number of lights affecting any given pixel is 1. This is because the lights will overwrite one another. But this can be fixed by utilizing a common feature of modern GPUs: blend functions.

The GPU allows the programmer to specify what assignment to the framebuffer actually means. For instance, using OpenGL functions like glBlendFunc, glBlendEquation, and related utilities, the user can instruct the GPU to perform raw addition on the frame buffer instead of assignment. For instance, if the blend function is set to add and a color RGB(0.2, 0, 0) is assigned to a framebuffer fragment of color RGB(0.1, 0, 0), the resulting value in the framebuffer would be RGB(0.3, 0, 0). This modification allows many lights to be drawn over any given number of pixels.

In practice, this approach requires some arbitrary constant that lowers the contribution of individual lights, or a tone-mapping post processing effect. We simply introduced a constant that reduced the contribution of individual lights to help prevent overdrawing.

Results

Using a computer of the following specifications, here are our results.

CPU - Intel i7 3520m 2.9GHz, 2 cores, 4 logical processors
Graphics Card - nVidia nvs 5200m, 1gig ram
Ram - 8 Gigabytes

City Scene
757 Lights - 9,084 Triangles
2490 cubes - 29,880 Triangles
3,247 Draw Calls
10.65 fps on 800 by 800 window size

Rabbit Scene
100 Lights - 1,200 Triangles
125 Rabbits - 5,000,000 triangles
225 Draw Calls
12.30 fps on 800 by 800 window size

Sphere Scene
1 Sphere - 20,000 faces
100 Lights - 1,200 Triangles
101 Draw Calls
10.42 fps on 800 by 800 window size

Our implementation was able to render these taxing scenes in times sufficient to be considered real time. We expect that running them on better

hardware would improve the times by some margin.

One peculiar stand out is the Sphere Scene - given that its statistics don't seem to match the performance, we believe this to be an outlier due to some mistake or bug.

Conclusions

We have successfully implemented a deferred pipeline for rendering point lights in geometry much more efficiently than traditional forward rendering pipelines. Our code successfully renders far more point lights than a forward rendered scheme could quickly, and the lights are reasonably represented in the final image.

Our implementation is bottlenecked currently in a few places. For one, we have created a separate geometry buffer for each point light source to be applied to the final render. This requires the same number of glDraw calls as there are lights in the scene, which could be reduced to one, simply by rendering all point light geometry to a single geometry buffer.

Our implementation also writes a world position buffer alongside a depth buffer for the sake of ease of calculation, however, the world position can be calculated from the depth buffer and camera position, meaning that a world position buffer is not required. Fixing this would reduce the storage cost of our implementation non-trivially.

With the core functionality of the deferred rendering pipeline established, there are several simple improvements that can be made to refine the output of the renderer.

Future Work

Ferko has compiled a number of improvements to the deferred pipeline we have implemented. He proposes a means of representing images with a high dynamic range by performing an extra pass after the lighting calculation, which generates a geometry buffer storing the total light intensity at a particular fragment. This buffer can be adjusted with various HDR schemes, such as the one described by Reinhard et al.

It would also be possible to support transparency for objects, by rendering all fully opaque models to the G-buffers first, and then modifying the buffer, scaling the existing data by the transparency, and adding the non-transparent component.

Also possible would be implementing the differently shaped light volumes described by Thaler, to represent different types of lights, including spotlights and area lights.

Figure 2: This scene contains a sphere, surrounded by a large number of colored lights. At this moment in the scene, a large number of lights happened to overlap in the right side. This caused the entire region to be 'overdrawn' and appear white.

Roles

Eugene was responsible for implementing most of the utility code for this project, including overhauling the parser, writing scripts to generate test scenes, implementing mesh classes separate from geometry to be able to render multiple objects with independent transforms, as well as light volumes.

Andrew was responsible for the setup of the deferred pipeline and for much of its actual execution. Most drawing code was at some point written by or modified by him. Setting up and binding framebuffers, vertex buffers, vao's, and shaders were his job. He also set uniforms, bound textures, chose texture formats, and wrote the shaders, as well as making the actual glDraw calls.

Ultimately, however, 90% of all programming took place in the same room together during pair programming. Substantial bugs belonging to both people were substantially helped by with the assistance of the other person.

References

Deering, M., Winner, S., Schediwy, B., Duffy, C., & Hunt, N. (1988). The triangle processor and normal vector shader. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88*. doi:10.1145/54852.378468

Ferko, M. (2012). Real-Time Lighting Effects Using Deferred Shading. *Comenius University*. Retrieved from http://old.cescg.org/CESCG-2012/papers/Ferko-Real-time_Lighting_Effects_using_Deferred_Shading.pdf

Reinhard, E., Stark, M., Shirley, P., & Ferwerda, J. (2002). Photographic tone reproduction for digital images. *ACM Transactions on Graphics, 21*(3). doi:10.1145/566654.566575

Saito, T., & Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. *Proceedings of the 17th annual conference on Computer graphics and interactive techniques - SIGGRAPH '90*. doi:10.1145/97879.97901

Thaler, J. (2010). Deferred Shading. *TU Wien*. Retrieved from https://www.cg.tuwien.ac.at/courses/Seminar/WS2010/deferred_shading.pdf

Appendix: Color Plates

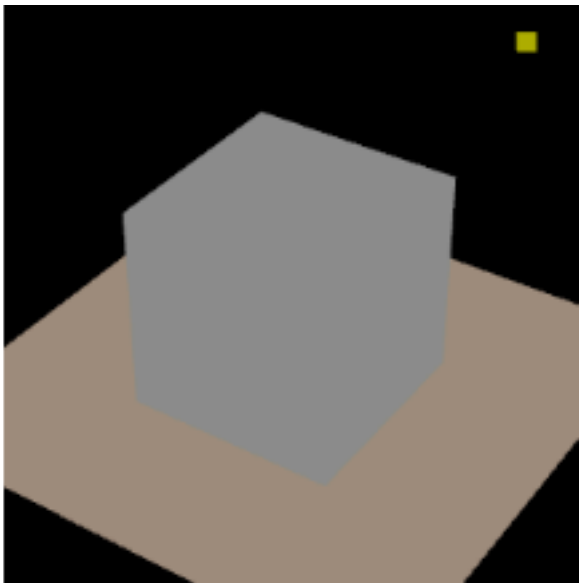

Figure 3: This is the output texture of the first pass, this particular texture contains the material colors of the objects in the scene.

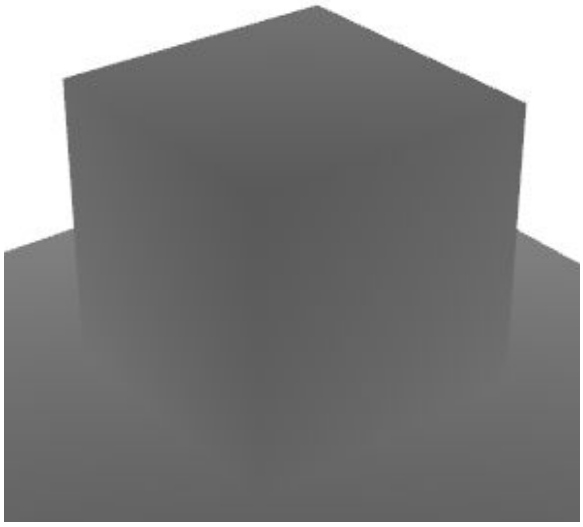

Figure 4: The same scene as figure 3, but rendered using the depth value to decide the brightness. This is a visualization of another first pass output buffer.

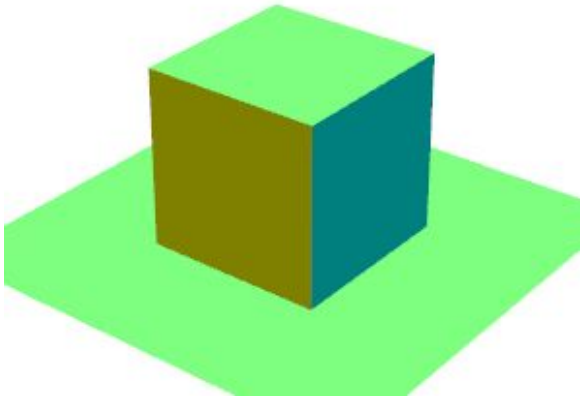

Figure 5: Yet another output texture from the first pass, this one visualizes the normals on an object.

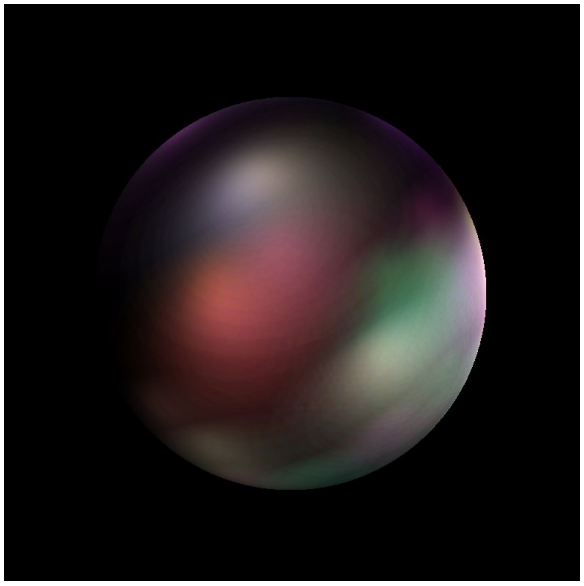

Figure 6: The sphere scene without the overdraw discussed earlier. The shader for this image simply dampened the brightness of all lights.

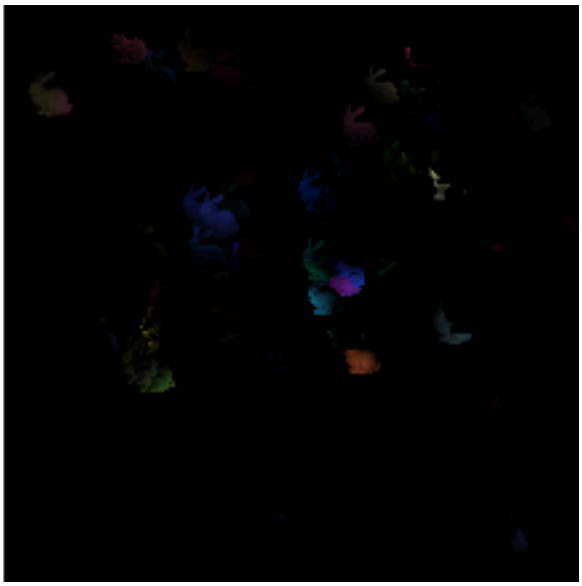

Figure 7: Final implementation rendering a scene with 125 bunnies with 40k tris and 100 lights.

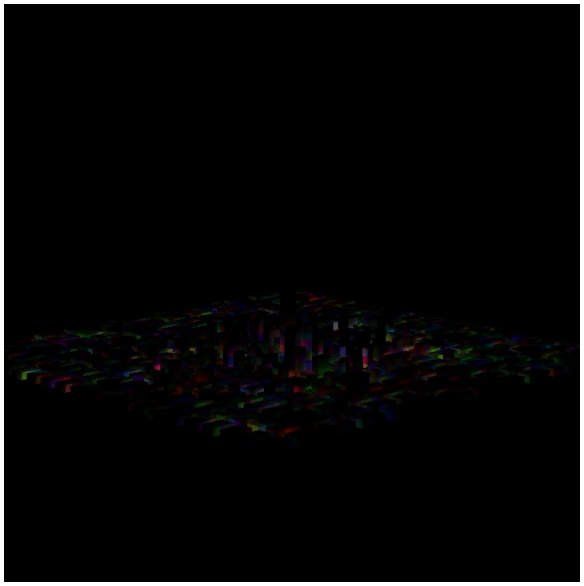

Figure 8: Final implementation render of a cityscape using 2490 cubes and 757 lights.