CS 448 Project Documentation: Confined

Alex Klen
ayklen
20372654

July 30, 2014

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this project was to create a rendering pipeline utilizing modern OpenGL shader techniques and exhibit it with an interactive demo game. The focus is on implementing several graphical features in an OpenGL shader pipeline. These features must not only be compatible, but they should together build a comprehensive rendering engine. The demo should extensively use lights and dynamic shadows and run at interactive frame rates.

This project was interesting and challenging because it involves using several advanced shader techniques at the same time. I learned how to effectively use GLSL on modern hardware to achieve quality light and shadow rendering at interactive frame rates, as well as how several other popular effects are achieved in shaders.

1.2 Overview of Features

Most of the rendering engine's features were objectives of the project, with a few extra features implemented as well. The features are covered in more depth and their technical aspects are discussed in Section 3.2.

The interactive demo is navigated using first-person controls, similar to first-person shooter games. Texture mapping and normal mapping allow high levels of detail on all the surfaces rendered with the pipeline. Textures are used as a polygon's diffuse colour if it is set. Normals for each fragment are perturbed by a tangent-space normal map to create realistic specular highlights. Dynamic shadow maps make a huge impact to the quality of the rendered output - it gives the player depth cues, but also makes the environment much more interesting because movements cause drastic changes in how the environment appears. Ambient occlusion is approximated with a technique called Screen Space Ambient Occlusion (SSAO) to improve the overall look of models. Instead of adding a constant ambient component to light partially shadowed areas and areas not in direct light, an occlusion factor is computed that approximates how much surrounding geometry covers each pixel. This causes depressions and crevices in models to be darker.

Keyframe animation was added to allow models to dynamically change between frames. It's tedious to simply apply model transformation matrices manually - keyframes can be exported from animation software. Planar mirror reflection was added. It's a feature that's interesting to interact with - often to see if it works like a real-life mirror. It also allows the player to see what their character looks like in the demo and to see his animation. Motion blur was added as a post-processing step that smears pixels proportional to camera rotations and translations in screen space. It improves the overall look of the game when the player is moving around and smooths out frame transitions at lower frame rates. Sound was added using OpenAL - sound effects are synchronized with game interactions and when using a stereo headset they will sound like they are coming from the direction that they would be for the character in the game.

Figure 1.1: Screenshot of Game - Looking at self in mirror.

Picking using shaders is an extra feature. Since you know which fragment is being drawn for each mesh in one of the shaders, you can use this to cheaply pick meshes.

The rendering pipeline is implemented as a deferred rendering pipeline. This is to avoid directly shading each vertex with each light (see 3.1.1).

These features can all be enabled at once to provide an immersive experience. Most of them can be disabled by pressing a number key. Toggling the features on and off lets you clearly see their effect on rendering. These settings are explained in Section 2.5.

Chapter 2

Manual

2.1 Running the Program

2.1.1 Hardware

This program requires a graphics card that supports OpenGL 3.3+. This version is required for its efficient buffers, customizable rendering pipeline, and multiple render targets (required for deferred shading). This project was developed with an NVIDIA GTS 360M.

2.1.2 Operating System

This program was written to run on linux. It uses a cross-platform library for windows and I/O, so can easily be adapted to work on other platforms, but this was never attempted during development.

2.1.3 Dependencies

In addition to libraries typically found on linux distributions, the following libraries are dependencies of this program:

- GLEW - OpenGL extension wrangling.
- GLU - OpenGL utilities.
- glfw3 - for cross-platform windows and keyboard and mouse I/O.
- assimp - for importing external model files.
- freeimage - for image loading, saving, and conversion.
- openal - for sound.
- alut - utilities for OpenAL.

These should be installed on the system prior to building the program.

2.1.4 Building

To build the program, simply use the 'make' command. Note that this program requires several libraries that are dynamically linked in the Makefile. They are outlined in the previous section.

2.1.5 Running

To run the program after building, simply run './confined' from the command line. There are no command-line parameters - see Section 2.5 for setting configurations, which are toggled during runtime. To quit the program press escape.

2.2 Controls

2.2.1 Looking Around - Mouse

The base controls for this program are similar to other first-person games. Simply move the mouse in two dimensions to change the angle you are looking in the 3D game world. Note that when the game is running it will capture the mouse cursor. You can either quit the program by pressing escape or minimize its window to regain control of the mouse cursor.

2.2.2 Movement - Keyboard

To navigate the 3D environment, you can use either the 'WASD' keys or the arrow keys. Using the 'WASD' keys is recommended if you are right-handed since it's easier to use them with your left hand while you use the mouse with your right. 'W' is up, 'A' is left, 'S' is down, and 'D' is right. Moving forwards and backwards will move you towards or away from the direction you are facing. Moving left and right will make your character strafe sideways.

Mainly for debugging purposes, the keys 'Q' and 'E' make your character float up and down.

2.2.3 Special Keys

- The escape quits the program.
- The 'F' key will toggle your flashlight, if you have one.
- The space bar will trigger a short hand wave animation by your character. You'll have to either be looking in the mirror or be looking at your shadow to see it.
- The number keys toggle settings. See Section 2.5.

2.2.4 Taking Items and Shooting - Mouse Buttons

Press the right mouse button while your cross hair is over an item to pick it up. The two items you can pick up are the flashlight and the gun.

Press the left mouse button after you have a gun to shoot it.

2.3 Playing the Demo

This project features a short demo to showcase the rendering engine. Following is a walkthrough of the demo.

When the game starts (after loading for a few seconds), you will see a wall ahead of you, but everything will be quite dark. Looking to the left, you see a door with light coming through it. You can step through the door and approach a table with several items on it. The light source is a glowing candle on the right side. There is a large, flat mirror that lets you see yourself in it. Also on the table is a red sphere and a flashlight. You can pick up the flashlight by right-clicking on it. After getting the flashlight, you can press 'F' to turn it on. Now that you can see farther because of the flashlight, you'll notice there is a cube with a brick texture and another door in the room. Progressing through the door leads to a room with a large window. Lighting flashes through the window periodically, with thunder sounds following. At the end of

the room is a monkey figure. If you check behind the monkey figure, there is a gun on the ground that can also be picked up. After picking up the gun, you can left-click to shoot (causing a sound effect and a flash of light). Shooting toggles a light in the previous room. This light orbits around the brick-textured cube. If you're feeling adventurous, you can step through a wall or float out the window to get outside. The sky is pitch black, but the grass has a nice texture and normal map you can look at.

2.4 Testing Features

You can test each feature of the rendering engine in this demo. Textures were mapped on the floor, walls, mirror, cube, gun, and grass. Normal maps are evident in several places: when you look at the candle from a distance, you see yellowish specular highlights on the ground; you can also point the flashlight straight down to see the floor's normal map working. After enabling the orbiting light, when it comes between you and the cube you can see the specular highlights on the brick texture line up with the texture. Shadow maps should be evident throughout the walkthrough. To test animation you can look in the mirror and press the space bar.

SSAO is off by default and can be toggled on using the '5' key. You should also turn on blur by hitting '6' (see Section 3.2.3). Motion blur is on by default, but you need to toggle it to see the difference it makes. You can toggle motion blur with the '7' key.

2.5 Settings

The number keys toggle the following features of the rendering pipeline.

0. Diffuse Lighting
1. Specular Lighting
2. Shadow Maps
3. Normal Maps
4. Texture Maps
5. SSAO
6. Blur
7. Motion Blur
8. Mirrors
9. Highlight Pick

Chapter 3

Technical Details

3.1 Rendering Pipeline

3.1.1 Deferred Rendering

Deferred shading is a method of shading that speeds up subsequent lighting and shadow depth passes by letting them perform per-pixel lighting calculations only, instead of computing them for all geometry and all fragments [2].

When forward-rendering, the vertex shader will compute lighting parameters (incident, normal, colours) which will be interpolated for each fragment. Deferred shading, however, uses two passes. In the first it computes geometric parameters for each pixel and writes them to a textures for later use. Subsequent shading passes can then sample these textures and operate per-fragment only. The benefit of this is clear when you consider shadow mapping with many lights, since when forward rendering you would need to run the shadow mapping algorithm over all geometry for each light and each fragment, whereas with deferred shading you only need to do it for each fragment. Although there are many shader passes the memory bandwidth of each is quite small.

3.1.2 Confined's Rendering Pipeline

Figure 3.1 shows the overall structure and some details of the rendering pipeline.

The Geometry Pass Shader is the first stage in deferred shading, and is responsible for taking scene data and creating 6 output render target textures. It takes a Vertex Buffer Object (VBO) that contains vertices, UV coordinates, normals, tangents, and bitangents of all scene geometry. This shader is run once on each mesh per frame. The shader also takes model, view, and perspective matrices so that it can convert vertices in model space to screen space. The Id of the mesh being drawn is also sent to the shader for picking. This shader outputs the 6 textures shown: diffuse, specular, normal, emissive, depth, and picking. The purpose of each texture should be self-explanatory.

Next, a loop is entered to perform the second stage of deferred shading. For each light in the scene, shadow maps are first generated and then shading for that light is accumulated to another texture. For the Shadow map Shader, only the vertices of meshes need to be sent to it, and it only outputs a single channel depth texture, which is a relatively quick operation. Next, this shadow map, along with the 6 textures generated in the first shader stage are passed to the Deferred Rendering Shader as samplers. This is the largest and most complicated shader - it performs all lighting, shading, and shadow calculations for each fragment. SSAO is also implemented in this shader. Note that the actual vertices sent to this shader are just a single quad; everything is done per-fragment with the teaxture-sampled values of that fragment. The results of this texture are accumulated for each light into the rendered texture.

The last step is post-processing. Now that all light shading has been accumulated to a texture, we pass this texture along with our depth and picking textures to the Post-Processing Shader. This shader also only

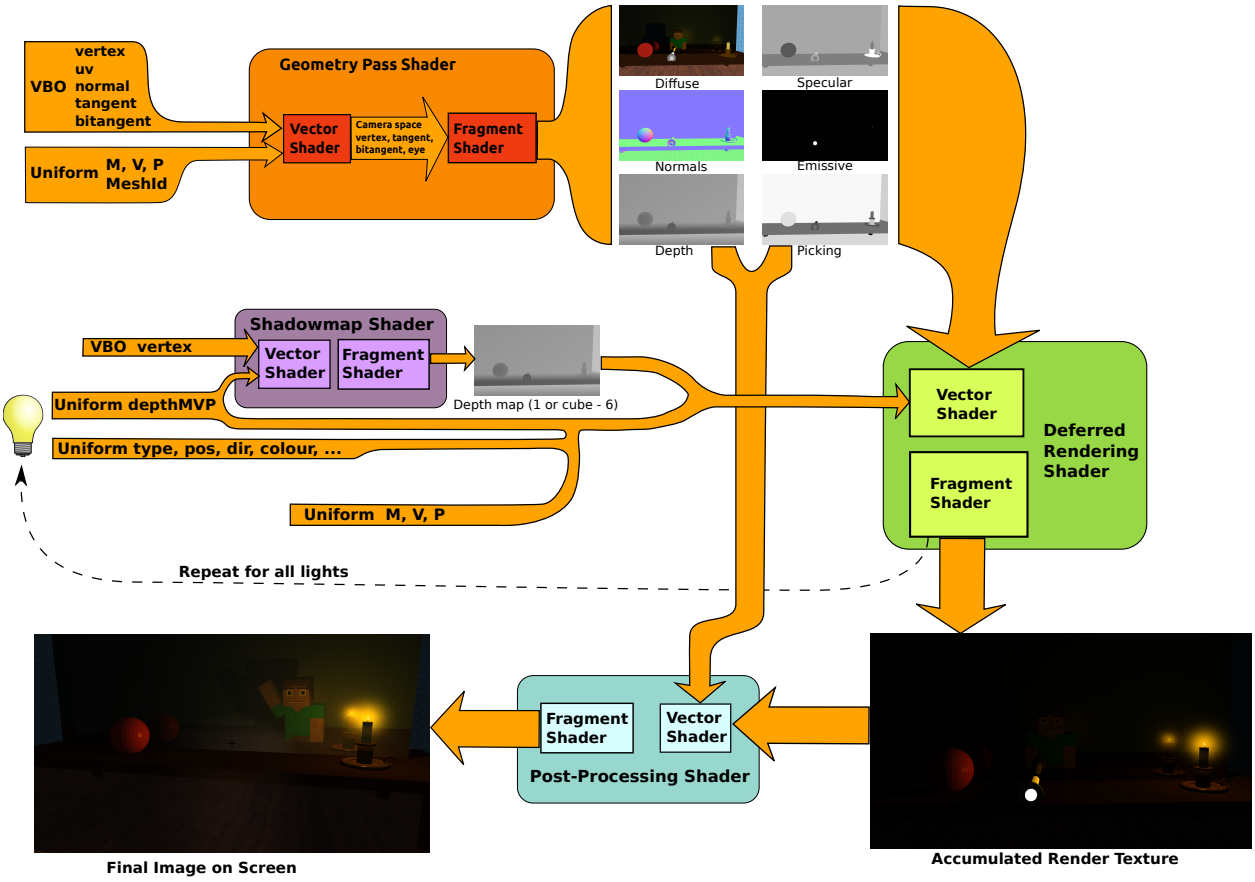

Figure 3.1: Rendering Pipeline

takes a quad for vertices and operates only per-fragment. Motion blur is performed here, along with regular blurring for SSAO, and debug highlighting for mesh picking (which is why the picking texture is needed). This shader outputs the final rendered image to the screen (or back buffer).

Not shown is how mirrors are implemented; see Section 3.2.5 for information about mirrors.

3.2 Technical Details

3.2.1 Normal Mapping

Normal mapping is when, usually in addition to a diffuse colour texture, you have a texture representing how normals should be perturbed at each pixel of the texture. The x, y, and z values of a normal are stored as the RGB values of the pixel. This technique lets you have very detailed bumps on a surface that would take millions of extra polygons to construct using only geometry. These normal vectors are stored in tangent space, and have to be converted to model space in order to be used for lighting calculations. I do this by pre-computing tangent and bitangent vectors on the CPU when loading models (see src/mesh.cpp:83-100). Then in the Geometry Pass Shader I compute a tangent-bitangent-normal matrix (TBN) to convert these to model space (see shaders/geomTextures.frag:64-71). These normals are then used to perform the regular shading calculations in the Deferred Rendering Shader (see shaders/deferredShading.frag:146).

I followed a helpful tutorial at http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/.

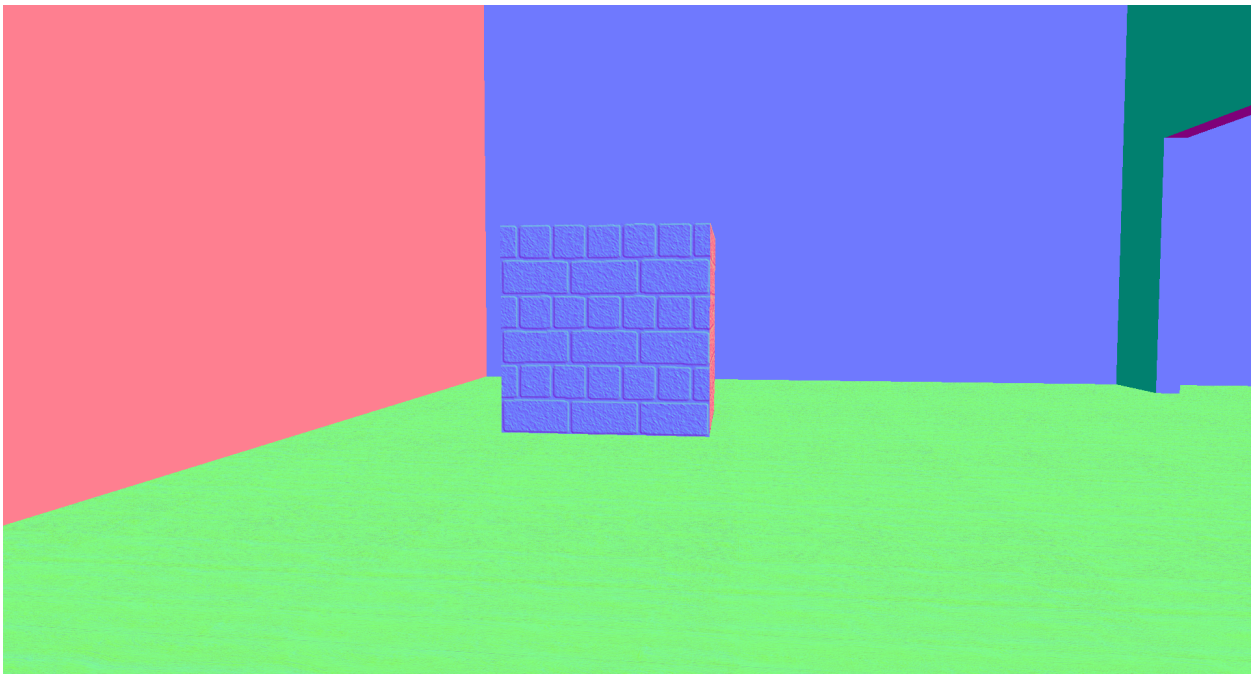

Figure 3.2: Normal Texture with Brick Normal Mapping

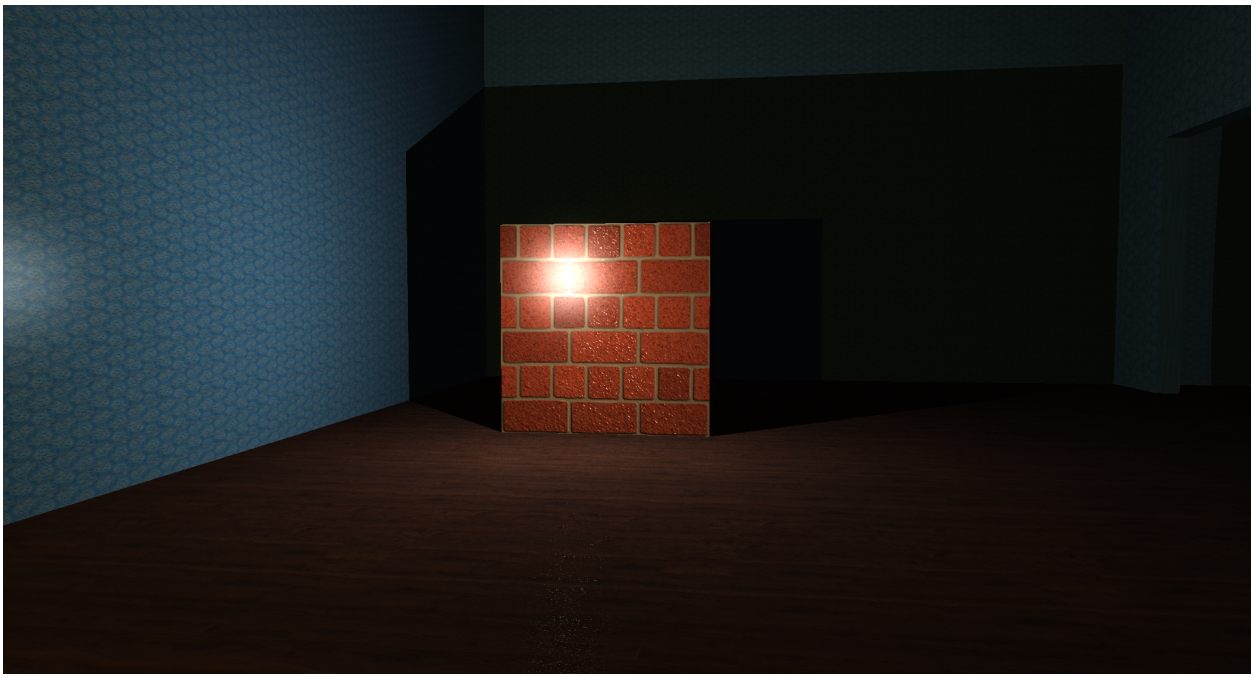

Figure 3.3: Render with Brick Normal Mapping

3.2.2 Shadow Mapping

The regular shadow mapping algorithm is well-known and consists of a first pass for each light where geometry is rendered from the perspective of a dynamic light source, and only a depth buffer is written [6]. Then

a second pass can perform lighting calculations while checking for visibility by comparing the depth the camera's perspective sees points to the depth that each light sees them. If the depth the light sees is less, then something is obstructing the point and it should not be lit by the light.

This algorithm is compatible with deferred-rendering and in fact allows you to use more dynamic light sources because they only have to perform shading for each fragment instead of for each vertex (see shaders/deferredShading.frag:235-291). See Figure 3.3 for a screenshot of point light shadow mapping.

Three light source types were be implemented (see src/light.hpp).

1. Directional Lights - no position or falloff and parallel light rays.
2. Spot Lights - position, direction, angle, and falloff
3. Point Lights - position and falloff, casts shadows in all directions.

Shadow mapping was the most challenging to implement for point lights because a texture cube map was required, which is created by performing 6 shadow map passes - one for each direction of a cube face at the light source. However, GLSL has a cube map texture sampler to make it easy to sample the correct face of the cube in the second pass.

There are multiple extensions to shadow mapping that eliminate visual artifacts present in the basic implementation and allow more realistic penumbrae, but they were outside the scope of this project. Two of these are Cascading Shadow Maps and Percentage-Closer Soft Shadows [1] [3].

I followed a helpful tutorial at http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

3.2.3 SSAO

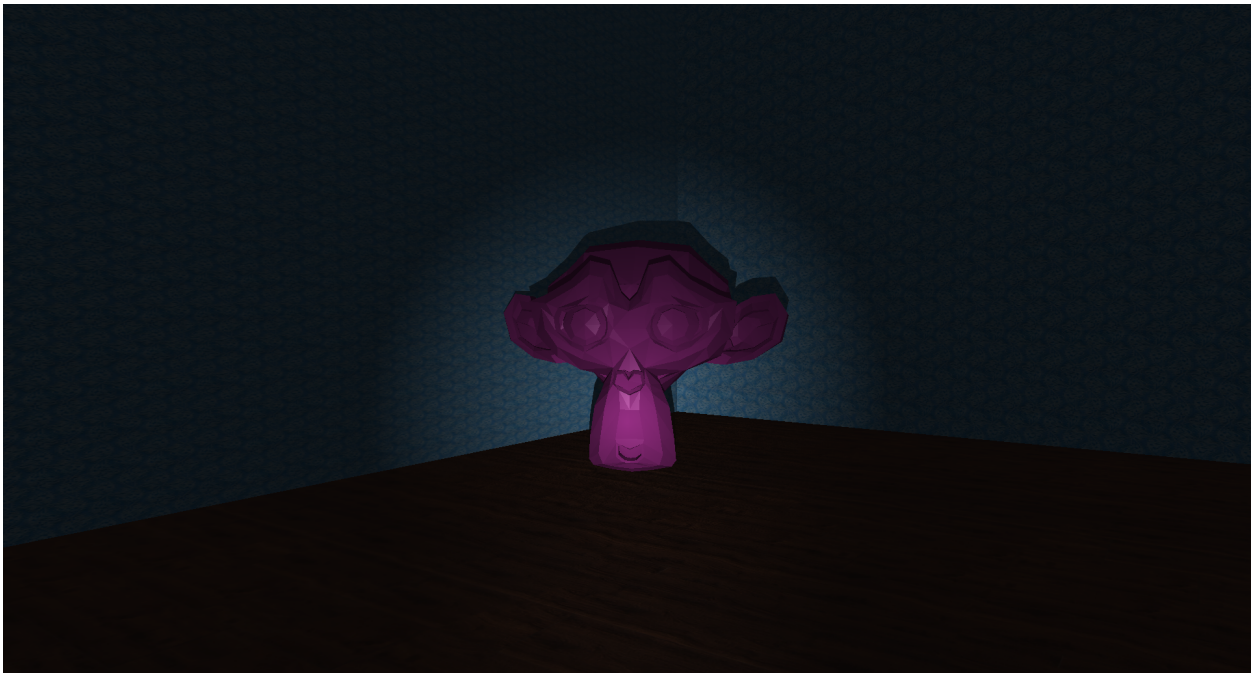

Figure 3.4: SSAO turned off

Ambient Occlusion means calculating an occlusion factor that can be applied to the ambient light term to improve lighting correctness. There is more than one algorithm for this, but this project uses the quick algorithm named Screen Space Ambient Occlusion [4][5].

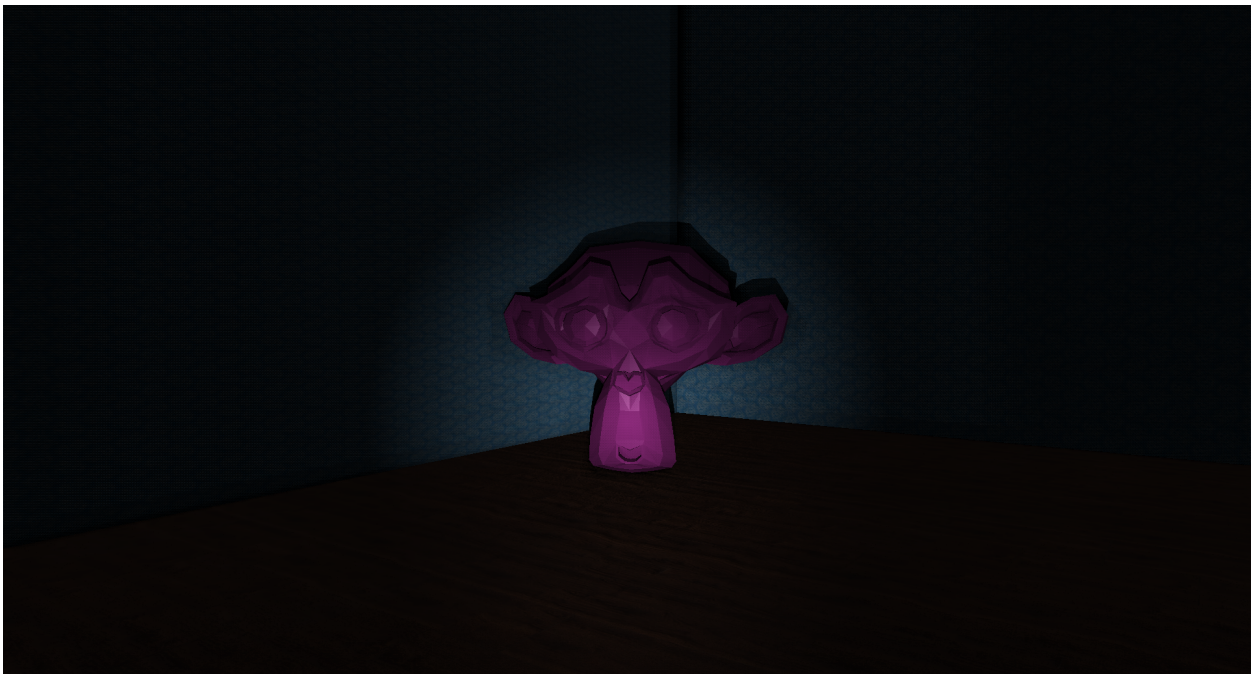

Figure 3.5: SSAO turned on

The algorithm works by sampling the depth of neighbouring pixels to determine whether a fragment should receive more or less ambient light. From the depth and normal information of each fragment, a view-space position and normal can be computed. Sample positions within a hemisphere oriented along the fragment's normal are selected based off of a pre-computed kernel and rotated randomly based off of a noise texture. These sample points are projected onto screenspace and the depth texture is sampled to see whether these points are in front of the fragment being shaded. The fraction of sampled points that are in front of the fragment being shaded is used to decrease the ambient shading factor for this fragment. See shaders/deferredShading.frag:304-315,68-89.

I followed a helpful tutorial at http://john-chapman-graphics.blogspot.ca/2013/01/ssao-tutorial.html?m=1.

3.2.4 Motion Blur

Motion blur is a an effect that blurs objects when move your camera. The amount of blur is proportinal to the speed of movement of the points on the screen. I implemented a screen-space technique that runs in the post-processing shader. The technique is based off of this tutorial: http://john-chapman-graphics.blogspot.ca/2013/01/what-is-motion-blur-motion-pictures-are.html.

The algorithm is to approximate where each pixel would have been in the last frame by storing the previous frame's view matrix, and blurring by sampling along the delta vector for each fragment. A matrix is computed and passed to the shader that maps the current frame's screenspace coordinates to the previous frame's screenspace coordinates. With this matrix a blur vector is computed by subtracting the two screenspace positions. Fragments are sampled along this blur vector and blended together to get the blurring effect. See shaders/postProcess.frag:69-86.

This technique is affected by frame rate, since if the frame rate is very high, then the previous view matrix will be only differ marginally and almost no blurring will occur. This can be fixed by calculating the frame rate and passing the shader a correction factor.

3.2.5 Planar Mirrors

Planar mirror reflections were implemented by rendering the complete scene an extra time to texture, and then applying the texture to a quad when rendering from the camera's perspective. Each frame, before starting the main render, the rendering pipeline is invoked for each mirror and rendered to a texture (see src/viewer.cpp:1068-1093 and src/mirror.cpp:58). For this invocation, instead of the camera's view, a reflection of the camera's viewing frustum about the mirror is used. The target texture is owned by the mirror, and it uses it as a diffuse colour texture for subsequent renders. An important step is to set the UV coordinates of the quad that represents the mirror to the geometry of the quad projected in the reflected viewing frustum's perspective. I came up with this technique on my own (with some inspiration from posts on the internet). See Figure 1.1 for a screenshot.

3.2.6 Shader Picking

I use a simple picking technique that leverages the fact that a fragment shader can be told the ID of the mesh it is rendering, and that ID can be stored in a texture. Then to pick I simply read the center pixel of the picking texture and recover the ID of the mesh the player is looking at. See src/viewer.cpp:695-700.

3.3 Implementation

3.3.1 Algorithms

See Section 3.2 for descriptions and code pointers of the main algorithms I used.

SSAO runs in time (kernel size) * (num fragments). I used 4 for kernel size in my implementation, so SSAO has only fixed overhead. Motion Blur runs in (num blur samples) * (num fragments). I used 10 blur samples in my implementation.

Normally, forward render pipelines require (num lights) * (num verts) * (shade time) + (num frags) to render multiple lights. Deferred rendering takes (num lights) * (num verts) * (shadowmap time) + (num frags) * (shade time). While this only differs by constants, outputting only the depth makes a large difference because depth buffers are highly optimized and smaller than colour buffers.

My mirror algorithm is not scalable - it scales the number of pipeline invocations linearly with the number of mirrors.

Shader Picking adds no new overhead - it's a single texture lookup.

All of the other algorithms run as a constant number of steps for each fragment (for example, normal mapping), with possible pre-processing costs.

3.3.2 Data Abstraction

I treat meshes as logical units that are rendered through parts of the pipeline one at a time. Each mesh is composed of a few VBOs and textures and represents some amount of geometry. A useful next step would be to be add a better hierarchical model with logical nodes instead of only meshes so that modelling transformations can be set to these nodes instead of every mesh.

My viewer class is monolithic and handles all of interactions with shaders. Shaders should be created as objects with interfaces to bind data and invoke them in order to make the code more modular and less fragile.

3.3.3 Code and Development Considerations

- The system is configurable with defines in the code and runtime settings that can be toggled by the user.
- There is a debugging mode that displays the textures relevant to the deferred rendering pipeline. I frequently used a helper method to check and print OpenGL errors if they occurred. Since the OpenGL

API is very stateful and doesn't explicitly report many errors, finding bugs in a larger program is quite involved. Having a debug mode that wraps all calls with error checking would be a useful addition for further development.

- The code is all my own, with ideas and algorithms based off of the referenced sources and linked blog posts in the relevant sections.
- I used git for version control, backups, transferring between my own computer and the lab computers, and for staying organized.
- Testing and verification was all manual since testing graphical output automatically is difficult.
- There is currently a bug that was introduced with hasty last-minute additions. When lightning goes off, some faces of spheres become darker.

Chapter 4

Conclusion

4.1 Future Possibilities

I am planning to continue developing this rendering engine in the future. I have learned a lot about GLSL and modern OpenGL techniques. I've been interested in graphics and game development for many years, but have never put in the time to properly learn how to write shaders - I'm glad that this project accelerated my learning and that the course taught me the fundamentals.

I am planning to create a full game eventually - it will likely use this project as a starting point. Next steps for this project would be skeletal animation and a physics engine.

There is lots of room for improvement for the shadow mapping technique used in this project, as mentioned in Section 3.2.2. Cascading Shadow Maps is a popular technique that would allow shadow mapping larger areas with high-resolution shadows.

Shader performance for this project should be improved. This is important because it would allow more dynamic lights to be added - currently adding too many drops the frame rate too low.

4.2 Acknowledgements

I would like to acknowledge http://www.opengl-tutorial.org for teaching me a lot of what I know about OpenGL 3.3+ techniques. It was a great starting point for learning the basics.

I would like to acknowledge john-chapman-graphics.blockspot.ca for providing me tutorials for some of the more advanced algorithms I implemented.

Bibliography

[1] Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.

[2] Michal Ferko. Real-time lighting effects using deferred shading.

[3] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, page 35. ACM, 2005.

[4] V KAJALIN. Shaderx7. charles river media, march 2009, ch. *Screen Space Ambient Occlusion*, pages 413–424.

[5] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007.

[6] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.