TSBK07 - Project

Hans-Filip Elo (hanel742), Lage Ragnarsson (lagra033) and Isak Wiberg (isawi527)

May 2016

1 Introduction

We have created a 3D graphics engine that we call "Emerald Engine" from a specification created before the start of the project[1].

The project specification were divided into two parts, one "will do" part and one "might do" part.

The reader of this document is expected to have some knowledge of the OpenGL rendering pipeline[2].

1.1 Will do features

- Deferred Shading
- Screen Space Ambient Occlusion (SSAO)
- User Controlled Movement of the 3D Camera
- Multi-Mesh Model Loading

User controlled movement of the 3D-camera is quite self-explanatory, but for the others a short background.

1.2 Might do features

The original "might do" features were:

- Functionality to compare deferred and forward rendering
- HDR
- Screen Space Reflections
- Camera animation paths
- Normal mapping
- MSAA
- FXAA
- Frustum culling
- Gamma correction
- Bloom
- Shadow mapping
- Shadow volumes

Of these features, comparing deferred/forward rendering, camera animation paths, normal mapping, and frustum culling have been implemented in the engine.

2 Background

The techniques used are described in this section.

[1]Project Specification, `https://github.com/lragnarsson/Emerald-Engine/blob/master/doc/project_specification.md`

[2]OpenGL rendering pipeline, `https://www.opengl.org/wiki/Rendering_Pipeline_Overview`

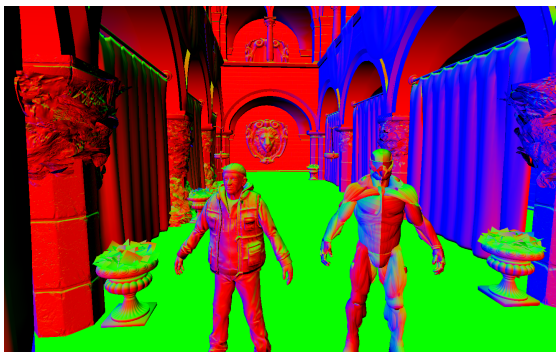

Figure 1: Normal component of the g-buffer.

Deferred Shading Deferred shading is a technique where you split the rendering into two separates stages; the geometry pass and the lighting pass. The geometry pass writes to what's called the geometry buffer which consists of four textures with the size of the screen. Instead of drawing colors to the screen the g-buffer is filled with information needed for light calculations. No lighting is calculated in the geometry pass. The technique is called deferred shading because you *defer* lighting calculations to a later stage. The g-buffer contains the following information for each pixel:

- Position - *World space coordinates*
- Depth - *View space depth, (GL_DEPTH_COMPONENT)*
- Normal - *World space normal vectors*
- Albedo - *Albedo texture colours*
- Specular intensity - *Specular light multiplier, as alpha component of the albedo texture*

In the lighting pass the vertex shader doesn't perform any calculations but the fragment shader reads all necessary values from the g-buffer and applies regular Phong shading. Since the g-buffer only contains values for geometry that's visible on the screen the renderer only calculates shading for fragments actually drawn.[3]

SSAO Screen Space Ambient Occlusion is a method for approximating shadows in areas with much geometry. The main idea is that areas that have much geometry in the vicinity should be darker, since light is less probable to reach in there. By using a deferred shading pipeline, the information about the scene geometry in screen space is available in the geometry buffers.[4]

With the scene geometry available in the screen space geometry buffers we get a data amount that is manageable. Using the entire scene to calculate occlusion would be more inefficient. SSAO can be accomplished in a number of different ways. The main idea can be broken down to a few simple steps that needs to be done for every pixel:

[3]LearnOpenGL, Deferred Shading, fetched Mars 2016, `http://learnopengl.com/#!Advanced-Lighting/Deferred-Shading`

[4]LearnOpenGL, SSAO, fetched May 2016, `http://learnopengl.com/#!Advanced-Lighting/SSAO`

1. Place a number of samples in the vicinity of the pixel
2. Check how many of the samples that are "behind" the geometry using the position-buffer.
3. Write the result to another buffer that is used in the lighting pass.

We have settled for a technique which places the samples in a normal-oriented hemisphere in relation to every pixel. The sample placement can be illustrated by figure 2.

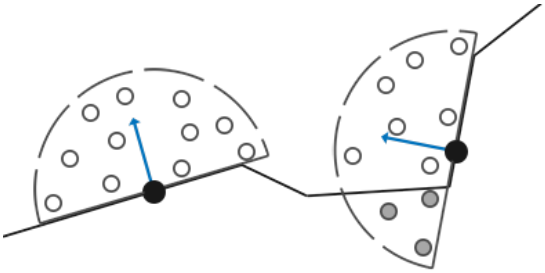

Figure 2: Samples are placed in a normal-oriented hemisphere. The samples in grey contribute to the occlusion factor since they are "behind" the geometry.[6]

The benefits of placing the samples in a hemisphere instead of a sphere is that flat surfaces will have no samples contributing to the occlusion, which means that edges of objects will have the same occlusion factor as a flat wall. The drawback however is that it will require more computations to find the sample locations. For every pixel an orthonormal basis must be created that transform the tangent-space sample kernel to world-space. If a sphere was used, this transformation would be a simple addition. Figure 3 shows the the result of the algorithm.

The reason for the repeating pattern in figure 3 is that a small random rotation of the sample kernel was introduced to allow the number of samples to be lower. This random rotation was not really random, but pre-generated in a 4x4 texture that was tiled across the screen. Since the exact cause of the pattern was known, it was easy to implement a smoothing shader that removed this pattern. The result from the smoothing shader can be seen in figure 4 and this is what was sent to the deferred lighting pass to be used as occlusion value.

Multi-Mesh Model Loading This means that the Engine is able to load complex models consisting multiple meshes with different textures, specular maps, normal maps and shininess.

2.1 Might Do Features

Features we wanted but weren't sure we'd have the time to implement.

[6]LearnOpenGL, SSAO, fetched May 2016,`http://learnopengl.com/#!Advanced-Lighting/SSAO`

Figure 3: Results from SSAO before smoothing. Notice the repeating pattern.

Figure 4: Results from SSAO after smoothing shader.

Figure 5: Normal mapping on a flat surface to the right and the albedo color to the left as a comparison. No shading is applied on the left which explains the difference in color and shade.

Comparing Deferred and Forward Rendering Comparing deferred/forward rendering is pretty straight forward, but means that the engine can switch between these rendering modes on the fly. The engine can also render each of the deferred rendering intermediate buffers separately in order to illustrate the composition of a frame.

Camera Animation Paths All animation paths, for the camera and for models, are Catmull-Rom splines. The position and view target of the camera can be animated by two separate animation paths.

Normal Mapping The engine is capable of normal mapping, which means that normals for calculating light can be different from the normal of a mesh plane for a specific fragment. This enables flat surfaces to look like they include bumps, which enables simpler models to look more life-like without using complex meshes. This saves on memory as well as calculations in the geometry stage. This can be seen in figure 5 where a bumpy wall is compared to the albedo color of the texture.

Frustum Culling Frustum culling in Emerald is done using bounding spheres. A bounding sphere is generated for each model which consists of a center point in model coordinates and a radius. All vertecies of the model is within this sphere. Using a method where the point of the bounding sphere closest to a frustum plane is checked, the models with spheres outside the frustum are excluded from the rendering pipeline as described in the course literature[7]. If a bounding sphere is not within the viewing frustum, a flag in the model is set which tells the renderer to ignore it.

3 Implementation

This section describes the implementation of the engine, for command mapping and usage of Emerald Engine, see the

[7]Ragnemalm I, Polygons feel no pain v2016, `http://www.computer-graphics.se/TSBK07-files/PFNP-2016-1.pdf`

Figure 6: Visualization of the bounding spheres generated.

project Github page[8].

The engine is written in C++ and GLSL. The engine uses the libraries

- SDL2 - *Simple Directmedia Layer version 2* [9]
- GLEW - *The OpenGL Extension Wrangler Library* [10]
- Assimp - *Open Asset Import Library* [11]
- glm - *OpenGL Mathematics* [12]
- Anttweakbar [13]

in order to get some abstractions and be platform independent. A lot of the shader work is inspired by the excellent guides on the website LearnOpenGL[14].

The engine currently consists of six main modules in the form of C++ classes, with some glue in between, and 11 shader pairs. The components of the C++ program are:

- Animation Path
- Camera
- Light
- Loader
- Model
- Renderer

Loader The Loader reads a scene file written in the Emerald Engine scene file format[15]. Having a separate file format

[8]Emerald-Engine, Github, `https://github.com/lragnarsson/Emerald-Engine/pull/64/files`

[9]`https://www.libsdl.org`

[10]`http://glew.sourceforge.net/`

[11]`http://www.assimp.org/`

[12]`http://glm.g-truc.net/`

[13]`http://anttweakbar.sourceforge.net/`

[14]`http://learnopengl.com/`

[15]`https://github.com/lragnarsson/Emerald-Engine/blob/master/doc/scene_file_format.txt`

for specifying scenes greatly reduces code clutter and enables us to modify the scene without recompiling.

Camera The camera holds information about the camera position, orientation and viewing frustum as well as witch animation path to follow (if any).

Animation Path An animation path holds a series of control points in 3D-space (at least four) and a value that specifies the period time for a full cycle when moving along the spline. When an Animation Path is asked for a position for a given time-parameter, it decides which four control points that the time-parameter corresponds to. The control points are then used by the Animation Path to calculate world-space positions along Catmull-Rom splines[16] which is used for smooth movement of objects.

Renderer This module renders all objects in the world using references to GPU-data found in models, lights and the Camera. All logic in the rendering pipeline as well as initialization is controlled from this module.

Light A Light is a container for light sources. Emerald Engine currently only supports point light sources. A Light can upload itself to the GPU when changed. There is a corresponding Light struct in shaders which apply lighting as well as an array of Light struct instances.

Model A Model is a container for everything that is needed in order to render a 3D model to the screen. A model can consist of multiple meshes. Mesh is another class which contains a VAO, EBO, the VBO's and texture references. It also contains all vertex data though there is no reason to keep this data after it has been uploaded to the GPU. The Model class includes functions to load a model from file.

A model can have lights as well as animation paths attached to it, in order for it to shine as well as move. When a model moves or rotates, any light sources attached to it follows, keeping their relative positions inside the model. When a model is created, it is specified as a normal object or a *flat* object. Flat objects are rendered using a flat color shader. If a flat object has any lights attached to them, it gets the color of the lights. Our intent with flat objects is to visualize the positions of light sources and it looks better if they are not shaded at all by other light sources.

4 Problems

During this project, fewer than usual problems where encountered. Some of the few things that bothered us however were:

The Mac OS X depth buffer (at least when using the Intel driver). The internal format of the depth buffer was not the same as GL_DEPTH_COMPONENT. This meant that blitting the depth buffer from the g-buffer did not work as intended. This was however solved by explicitly using GL_DEPTH_COMPONENT32 when generating the depth buffer for the geometry buffer.

We have a different related problem with Z-fighting on OS X when in forward rendering mode. The cause of this is unknown to us.

Another thing that bothered us early in the project was the **GLSL preprocessors lack of an "#include-statement"** since we couldn't define common constants for shaders in a single file. There is an ARB[17] extension that achieves this by adding a C interface to define these, however they were not compatible with standard APIs for loading and compiling shaders. This meant that we either needed to build our own include mechanism or manually update all the shaders when we wanted to change a variable. We decided on the former. We define our shader constants in the makefile and preprocess the shaders before handing them off to OpenGLs shader loading.

We had a problem with a **Nvidia GPU running on an Ubuntu GNU/Linux system which didn't want to upload parts of a struct as a OpenGL uniform**. The structs in question were light sources, which resulted in the lights not moving, even though the models for them did. We did a work-around by instead uploading the entire struct to the GPU, which only consists of seven floats.

When the **Catmull-Rom splines** were implemented, we encountered some bugs where the screen turned all black. After some investigation it was found that this happened when the camera position and the look position were very close to each other. This was solved by checking if the camera and it's look-point were following the same spline, and the difference between their time-parameters was too small. In that case the look-point's time-parameter was increased a small amount to ensure that the camera position and the look at-point never ended up too close.

5 Conclusions and Future Work

All in all we are happy with how the project turned out. There's always room for improvement but the end result looks pretty decent and has smaller amount of code clutter than all project members could have imagined. The group really worked as a team during the entire project, while still being able to work separately thanks to the work flow.

One thing that we were really pleased with is the choice to use the GitHub work flow[18]. This workflow has lead to very few bugs appearing in to the mainline code, while also

[16] Ragnemalm I, Polygons feel no pain, fetched May 2016, `http://www.computer-graphics.se/TSBK07-files/pdf16/11b.pdf`

[17] OpenGL, All about OpenGL extensions, `https://www.opengl.org/archives/resources/features/OGLextensions/`

[18] GitHub work flow, fetched may 2016, `https://guides.github.com/introduction/flow/`

helping create a larger understanding of the codebase for all project members.

Our current plan is to expand the feature set of the engine during the advanced course. If we decide to go this route possible additional features may include physical based rendering, shadows, reflections, blending, collisions/physics, scripting support, particle effects, some kind of global illumination approximation, in engine scene editing, water, post processing effects such as HDR, bloom, gamma correction or depth of field etc etc.