Advanced Graphics and Realtime Rendering

First Semester Report

A picture containing sign, outdoor, light, lit

Description automatically generated

Anthony Sturdy (18015368)

BSc Computer Games Programming

Staffordshire University

07/01/2022

Contents

[User Guide 3](#_Toc91193668)

[Start-up 3](#_Toc91193669)

[Controls 3](#_Toc91193670)

[Features 4](#_Toc91193671)

[Normal Mapping 4](#_Toc91193672)

[Self-Shadowed Parallax Occlusion Mapping 4](#_Toc91193673)

[Render Pipeline / Compute Shaders 5](#_Toc91193674)

[Screen Space Effects 5](#_Toc91193675)

[Bloom / HDR 6](#_Toc91193676)

[Depth of Field 6](#_Toc91193677)

[Shadow Mapping 7](#_Toc91193678)

[GUI 8](#_Toc91193679)

# User Guide

## Start-up

To begin the application, open the *‘Advanced Graphics and Realtime Rendering Engine.sln’* file with Visual Studio 2022. Click *‘Local Windows Debugger’* at the top of the window. The application should compile and start. If Visual Studio 2022 is not installed, instead, open with Visual Studio 2019. The project’s *Platform Toolset* version may need changing if the application will not compile. To do this, right click each project in the solution, then click *‘Properties’* Under *‘Configuration Properties -> General’* there should be an entry labelled *‘Platform Toolset’*, change this to version 142. This will need to be done for both projects within the solution.

## Controls

The scene can be controlled via the GUI. If there are no GUI windows, or the windows appear very small, click-and-drag each window to dock it to the main window. Subsequent start-ups should not have this problem. There should be six windows in total: Viewport, Scene Controls, Performance, Image Filters, Depth of Field, and Bloom.



The control gizmo seen in the above image will appear when an object is expanded within the *‘Scene Controls’* window. To control a point light, the position can be modified by moving the Cube as it is attached to the Cube, and the emissive values on the Cube object control the light colour and intensity.

# Features

## Normal Mapping

Starting in the vertex buffer initialisation stage, normal mapping requires the binormal and tangent to be calculated for each vertex to properly apply the normal map sample. This data is then passed into the Vertex Shader with the other vertex data (Position, Normal, Texture Coordinate), where the values converted to World space, and passed into the Pixel Shader.

The normal map texture is passed into the Pixel Shader, along with the world-space vertex data. The normal map texture is then sampled at the current texture coordinates, decompressed to a value which ranges between -1 and 1 (instead of 0 and 1 when it is first sampled). Lastly, the decompressed normal map sample is multiplied by the TBN matrix, which consists of the Tangent, Binormal and Normal, which results in the final normal value and can be used in lighting calculations as the vertex normal would usually be.

|  |  |
| --- | --- |
| only diffuse | Diffuse + normal mapping |

## Self-Shadowed Parallax Occlusion Mapping

Parallax Occlusion Mapping is performed before the normal mapping in the Pixel Shader, as it modifies the texture coordinates which are used in the normal mapping process.

Firstly, a view direction is calculated by multiplying the vector from the eye position to the world position, by the TBN matrix, converting it to tangent space. The number of layers are then calculated using the dot product between the normal (tangent space) and the view direction, this optimises the algorithm as less layers are required to achieve the occlusion mapping effect when the view direction is perpendicular to the surface normal. More layers would mean more calculations, which are required when the view direction is more parallel to the surface for visual fidelity.

A loop is then entered which ray marches through the depth map until an ‘intersection’ is found. The ray marching algorithm moves the current texture coordinates by an amount determined by the view direction, samples the depth map at that texture coordinate, and increases the current depth by the amount of one layer. If the depth of the current layer is less than the sampled depth map value, the loop can exit as an intersection is found. The offset texture coordinates are then used for the rest of the shader.

The self-shadowing uses a similar algorithm, except the algorithm travels upwards from the current depth towards the light source. If an intersection is found, in the same way it is checked for parallax occlusion mapping, the current pixel is in shadow. If the loop reaches the surface without intersecting with anything, the pixel is not in shadow. The algorithm has a soft-shadow effect by using the depth of the intersection for the intensity of the shadow, giving the shadow an effect that light has bounced around the edges and lightly illuminated inside the shadow.

|  |  |
| --- | --- |
| diffuse + NORMAL | Diffuse + normal + PARALLAX |

## Render Pipeline / Compute Shaders

Each render pass is split into it’s own class which inherits from the RenderPipelineStage class. This allows a vector of RenderPipelineStage objects to be stored, and common methods can be called on each, such as Initialise and Render. The base class also stores the device and device context, as well as a static Unordered Access View (UAV) which allows both reading and writing and is shared across all RenderPipelineStage objects. This allows each render pass, specifically for post-processing passes, to iterate upon the image. The UAV is what is eventually rendered to the Viewport GUI window.

The post-processing passes, following the geometry pass, all use compute shaders for image processing. These shaders are what read and write the static UAV. The compute shaders are dispatched in blocks of 8x8 threads.

## Screen Space Effects

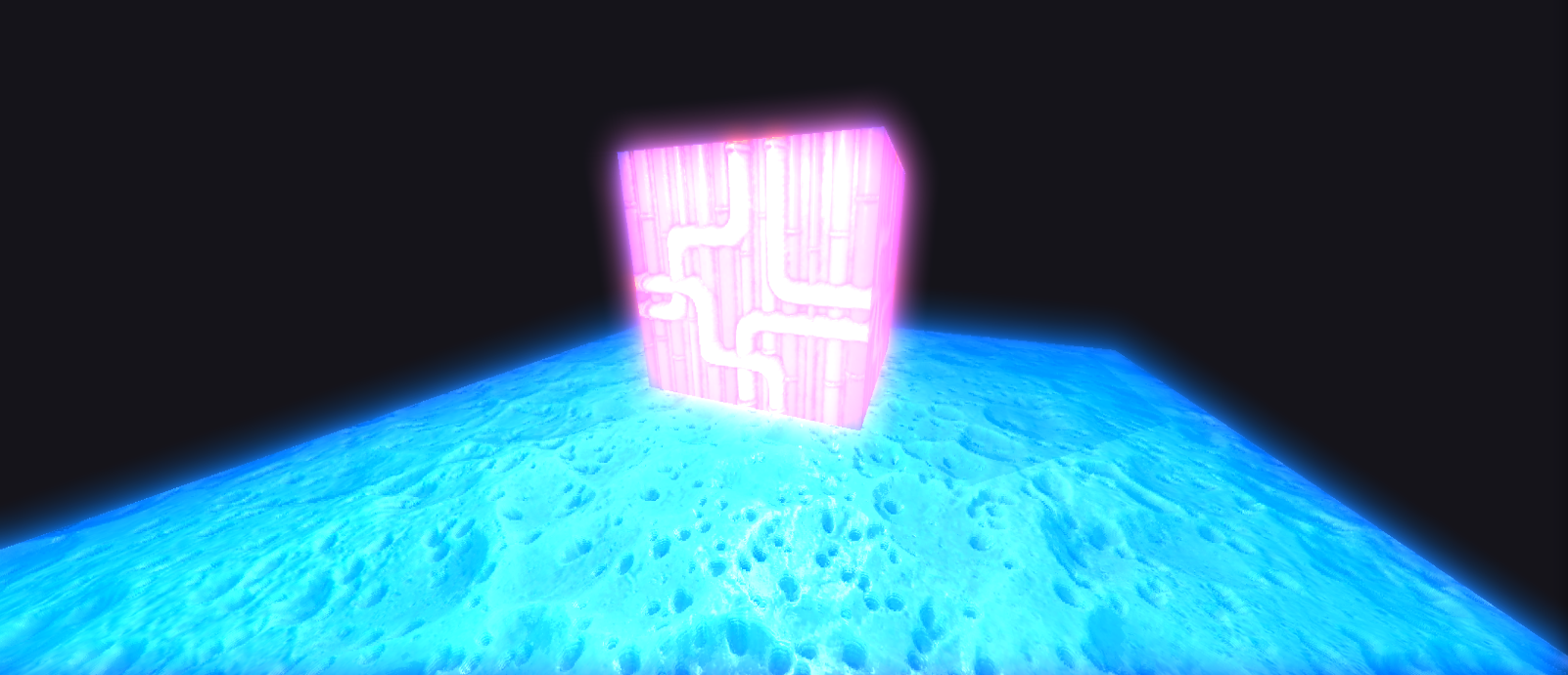
The screen space effects are the final pass in the render pipeline, meaning the effects applied over all other effects. The application offers four types of image effect: Invert, Greyscale, Film Grain, and Vignette. There is also an intensity slider for adjusting how strong the visual effect is.

|  |  |  |
| --- | --- | --- |
| Graphical user interface  Description automatically generated |  |  |

## Bloom / HDR

Bloom begins in the geometry pass, where two render targets are bound and output to by the Pixel Shader. One render target is the main image output, the other is the HDR output. The pixel shader can output the highlights on the image to the HDR, such as if the pixel’s colour value is higher than 1, but for the best visual result, the emissive material value is output to the HDR render target as it causes the boxes look much more like they are realistically glowing.

The next stage of the Bloom effect is performed within a Compute Shader. A single pass gaussian blur algorithm, which samples pixels in a circle around the current pixel and averages the colour, is applied to the HDR layer, then is combined with the main image colour and output to the shared UAV for use in the next render pipeline stage.



## Depth of Field

Depth of Field is an effect which also begins in the geometry pass. The alpha channel output in the Pixel Shader is set to the depth value for that pixel, which is used for the Depth of Field effect and eventually discarded for final output for a value of 1.

At the Depth of Field pass, which also uses a Compute Shader for image processing, the same blur algorithm as Bloom is used, and the blur intensity for each pixel is calculated at runtime depending on where the user is looking.

As the compute shader can sample the entire image, the depth is sampled at the centre of the image, as well as for the current pixel. The blur intensity is then calculated based on the difference between these two values by using the following equation:

In the equation, is the current pixel depth, is the middle/centre pixel depth, is a value to determine how quickly steep the blur falloff is, the smaller is, the more intense the depth of Field effect is. The result is that the camera ‘focusses’ on the point in the centre of the screen, and that the blur intensity is based on the difference between the two points relative to the depth in the centre of the screen. This creates the effect seen with cameras where two objects which are far apart in depth can both be in focus if they are both far from the camera. Whereas if one object is relatively close to the camera, the blur falloff is much steeper. This can be seen in the following example images which compares real life photographs to engine renders in similar situations. The out of focus effect also applies when the foreground is not the focal point, which follows similar properties except the closer object would be blurred.

|  |  |
| --- | --- |
|  |  |
|  |  |

## Shadow Mapping

Shadow mapping is the first stage of the render pipeline, it finishes in the geometry render pass within the Pixel Shader.

Firstly, the scene depth is rendered from the light’s point of view using an orthographic projection. An orthographic projection simulates shadows similar to the sun’s. Once the depth has been rendered from the light’s perspective, the depth texture can be passed into and used in the geometry pass Pixel Shader to calculate whether the current pixel is shadowed.

In the Vertex Shader, it is required to calculate the vertex position in light space, this is done by multiplying the vertex position by the world matrix, light view matrix, and finally the light projection matrix.

In the Pixel Shader, the shadow map texture coordinates are calculated firstly, followed by the depth from the current pixel to the light. Sampling the light depth texture using the calculated shadow map texture coordinates and dividing that value by the true depth value results in the difference between the actual depth and the depth which the light can ‘see’. If the resulting value is less than 1 (theoretically, the shader subtracts a small bias value, so actually checks for less than 0.999), then the current pixel is in shadow.

## GUI

The GUI is handled and rendered using the ImGui library. There are two ImGui extensions used, ImGuiFileDialog for selecting textures at runtime, and ImGuizmo for the transformation gizmo controls which appear when a GameObject is expanded in the ‘Scene Controls’ window.

Use of the ImGui ‘docking’ branch allows windows to be removed from the main window and docked to the main window which allows the window to be rearranged and organised however the user would like.

The viewport window renders the final post-processed image to an image which fills the size of the window. This separation allows the viewport to be moved to another screen if desired, or to be resized independently.

A screenshot of a computer

Description automatically generated with medium confidence