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

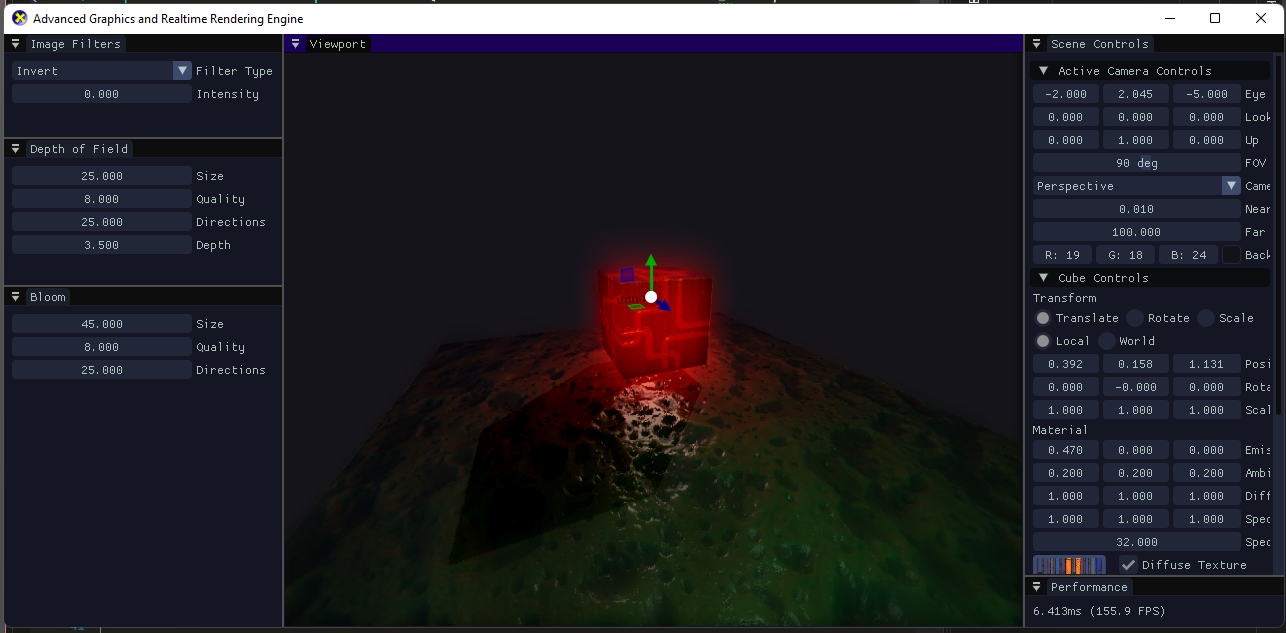
# User Guide

## 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.

# 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 illuminated inside the shadow slightly.

|  |  |
| --- | --- |
| 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.

## Bloom / HDR

## Depth of Field

## Shadow Mapping

## GUI