# Advanced Shader Programming – Coursework Report IMAT3907



## Introduction

This report provides a deep analysis of the design and implementation methodologies I have used to develop a terrain generation system, as per the specification of the marking scheme.

This project was driven by three core objectives, to develop a modular shader pipeline capable of rendering advanced lighting and visual effects, the use of appropriate Level of Detail (LOD) handled through shader code and the generation of an efficient and visually attractive procedurally generated terrain.

Thus, my system utilizes a robust shader pipeline, capable of handling distinct terrain features such as: The Planet, Water, Atmosphere, Moon & Clouds. To set my work apart from the norm I decided to use dedicated compute shaders to generate a cubic, subdivided mesh for the planet's terrain and another to develop a Perlin noise-based cube map for height mapping.

The shader sequence, composed of Vertex, Tessellation Control, Tessellation Evaluation, Geometry, and Fragment shaders, dynamically transforms the vertices. Notably, the terrain is sculpted through two pivotal processes: heightmap application and a mathematical model that morphs a cubic structure into a spherical one. This dual modification results in a planetary terrain that is both computationally efficient and visually striking.

[Further details explain the utility and justification of each shader stage & application are listed in their relevant sections of this report document.]

## Lighting

*Per Vertex Normals:* In preparation for my lighting calculations using TBN Matrixes in my geometry shader, I chose to use per-vertex normals which are calculated using the Central Difference Method. This technique approximates surface gradients by evaluating the rate of change across three dimensions, ensuring that the normals are of unit length for accurate light calculations. The Central Difference Method is more accurate than the first order forward or backward differences as it utilizes points on both sides of each vertex to compute gradients. Historically, the significance of this method was recognized by Lagrange fifty years after its introduction by Brook Taylor in 1715, who lauded finite difference methods as the cornerstone of differential calculus [1].

*Skybox:* As the goal of creating an entire universe and rendering it with lighting in real time is a very difficult computational task, I opted to use a Skybox to encapsulate my scene with a seamless universe like backdrop. Skyboxing is a method of encapsulating a scene with a textured cube that stores its data in a cube map, consisting of 6 seamless texture that represent the different faces of a cube. By converting the viewport’s projection matrix to a mat 3 and back to a mat 4 then projecting the position coordinates to pos.xyww the cube will be projected surrounding the camera and thus does not need a model matrix as it will always be centred around the viewpoint. To avoid future fragments being ignored due to the skybox’s proximal depth coordinates, depth mapping must be disabled for the skybox’s draw call. In 2004 Marino [2] released a book in which he explored “3D Game-based Filmmaking”, the process described in this book seems to be one of the earliest uses of a skybox / skydome. He describes his methodology in replicating expansive backdrops for films using a green screen with game engine technology rendering seamless textures into the film, instead of the costly process of hiring artists to paint and detail physical backdrops for filming.

*Billboarding:* The cost of rendering objects like clouds and tree’s is roughly the same as the cost of rendering my singular planet within the scene, to keep costs down rather than spend computation time on rendering a high quality 3d model of a cloud, I opted for an advanced billboarding technique that uses multiple cloud textures at different elevation levels and azimuth angles, and computes a local coordinate system based on the direction of the cloud from the centre of the planet. This then compares that with the elevation and azimuth angles of the cloud to the view position to select the appropriate texture from a texture array.

The method of billboarding itself is relatively simple in contrast to this calculation, the billboards are simply a quad generated at a singular position given to the shader, scaled by a scale uniform that is passed from the C++ code base and the texture is oriented by taking the cross product of the Up direction and the view direction, then recomputing the Up vector based on the cross product of the Right and view direction.

*Physically Based Rendering & Blinn Phong:*  PBR is a mathematical model proposed by Greenberg, D. P. [3] in a publication from Cornell in which he detailed the use of a mathematical model to simulate the interactions of light with surfaces that mimic real world properties using complex formulae for light reflection and absorption. This maths is based around material properties of Albedo, Roughness, Metalness and AO. In contrast, the Blinn-Phong model, a simpler and less computationally demanding method, approximates specular highlights using exponentiation derivatives of dot products, which can lead to less accurate rendering under diverse lighting environments. However, after profiling the computation time, the computation of PBR with a high-fidelity model at high tessellation levels took roughly 780% of the computation time of a standard diffuse lighting method.

A screenshot of a computer

Description automatically generatedThese results are very easy to interpret, the increase in computation time and Ram usage dependent on lighting model shows a massive increase in both when using PBR. Yet, PBR's main advantage is its accuracy and photorealism which are particularly beneficial in applications like film and realistic video game rendering.

However, my personal opinion as the coder is that I prefer the simplistic diffuse only version of this rendering over both Blinn Phong and PBR. While the implementation of each of these is usable in my scene, I have elected to make standard diffuse lighting the default state of the render. [This can be changed with the ImGui Interface on runtime]

*Image Based Lighting:* IBL is a commonly used method of approximating global illumination by using environmental mapping to simulate dynamic reflections and refractions of light on a surface. The use of this in my system allows my lighting equations to simulate light cast by my universe skybox and reflect this lighting effect onto my water simulation. This is an effective method of making an object in the scene seem more natural through accurately depicting its reflective and translucent properties. In the case of my water shader, I have combined this with a Fresnel equation that is used to make these reflections seem more prevalent with a shallower angle of view.

## Level Of Detail (LOD)

*Subdivided Mesh in Compute Shader:* To optimize the mesh generation of my terrain, I chose to compute and store vertices directly on the GPU via compute shaders. This approach leverages the GPU's parallel processing capabilities, significantly speeding up the recompilation of base planet vertices, as opposed to handling these calculations on the CPU.

*Dynamic Tessellation Level:* For dynamic tessellation levels, my initial step was to precompute a base static subdivision level through a compute shader thus enhancing vertex fidelity in regions of low tessellation and allowing for higher runtime tessellation detail. The downside of this is that this introduces challenges in processing time for areas of higher tessellation level and dramatically increases computation for these regions due to the use of a linear polynomial factor of subdivisions.

A close up of a text

Description automatically generated

This estimates the base processing time for any runtime calculation on a single core thus illustrating an exponential increase in complexity as either the tessellation level or the base subdivision level rises, therefore impacting performance on devices with lower GPU performance.

*Distance Dependent Heuristic & Smooth LOD Transitions:* This principle explained in the prior paragraph, as stated results in exponential increases in processing time, so in a situation utilizing only a dynamic tessellation level, rendering a multi planetary sim, this exponential factor would likely cause even the most capable chipsets to fail under load. Thus, a heuristic factor for tessellation and LOD dependent on linear distance results in large reductions in computation time for fragments by reducing the exponent term of the prior computation algorithm.

For this I tested multiple heuristic LOD algorithms; Linear Distance Falloff (LDF), Exponential Distance Falloff (EDF), Screen Space LOD Optimization (SSO) & a Z-Prepass Depth Falloff (ZPD).

A screenshot of a computer screen

Description automatically generated*Conclusions:*

LDF: The use of Linear Falloff on a 3d Planetary Surface works very effectively, due to the curved shape of the planet, at areas closer to the view the curvature of the vertices is less notable thus resulting in a clean falloff with Minimal Edge Visibility.

EDF: Exponential Falloff surprisingly performed worse that Linear falloff in terms of edge visibility, I believe that this is due to the use of a cubic planetary body converted into a Spherical Mesh, I preceded to repeat this test with the spherical nature turned off and this method seemed to improve with massively improved edge visibility, but for my use case this method is less efficient in computation time and less effective in edge visibility.

SSO & ZPD: The algorithms used in these directly sample distance of visible edges only, both cases were more efficient computationally, however held the same fault on cases where edge vertices were not visible, the tessellation level was massively reduced and thus resulted in edges falling towards the sides of the viewport having clear visual tearing. A solution for this could be alter the projection matrix have a greater FOV than the visible viewport to avoid this edge tearing, however due to the effectiveness of LDF I have chosen to use that in my tessellation shaders for my shader programs.

[Reference to the testing of these functions is in my Testing-branch of Git]

# Terrain

*Terrain Generator:* The first step of my terrain generation process is featured in my Terrain Generator Class; This class handles the creation of a subdivided mesh at a level based on an input integer subdivision level and a compute shader with input work group sizes of x and y equal to the subdivision level. The VAO, VBO & IBO for the planetary surface is generated in my initialize mesh function. The IBO is populated by a function generate Indices that returns a vector of GLuint indices and appends the indices to the VAO as buffer data. The generation of the terrain mesh as described in the LOD subdivided mesh section of the report is then created through binding the VBO to a SSBO buffer base and is modified by a compute shader on initialization. This can be called again through the ImGui Interface to recompute and change the subdivision level.

*Biome Distributor:* The Biome Distributor is a class designed to generate the height map for the VAO and the DuDv map for the water, the height map is generated through a generic Perlin noise algorithm acquired from a SIGGRAPH Course Notes [4] which details the production of this. To make the generation of procedural terrain easier I faced an issue that Perlins permutation tables for this method are too complex to be used directly as an input to the shader, so to enable randomness per initialization I chose to use an alternative version of this noise known as simplex noise with a float seed. I then improvised a method of production for layered Perlin noise that combines 3 separate scales of noise layered on the same texture to provide continental heights, scaled terrain protrusions and fine noise for small bumps in the terrain. (For this I was inspired by a method discussed by Notch in the production of Minecraft terrain). For The DuDv map, the coordinates of the noise are multiplied by a uniform elapsed time so that the motion of the waves changes with time passed instead of frames elapsed. The Biome Renderer also generates 2 other textures known as precipitation and temperature. The temperature map is a simple texture that uses Longitudinal and Latitudinal coordinates to map the equator of the planet and return higher values closer to the equator. This is then modified in the Biome Render to also consider the height of the terrain to determine biome colours / textures used in rendering. The Precipitation Map is an inverse of the original height map noise and is used to describe the proximity of water to the terrain locations determining whether to use forest, grass or sand textures / colors. As this Class handles the generation of texture cube maps rather than just singular textures, this class also has a getDirection function that is used to determine what face of the texture is being generated and through the use of the workgroup.z each face is generated in parallel for all of these functions.

*Biome Renderer:* The biome renderer has been discussed in prior segments so I will not go into too much further detail as to its rendering functions. However, the main undiscussed utility of this class is that it is used to project the model matrices of the scene for elements of terrain such as the water, atmosphere, moon, and planetary body. This is also housing the methods of loading uniforms and textures using the struct ‘Celestial Body’ housed in the Biomes.h class and is used as an instantiable class for planets.

*Weather System:* Again, the weather system has been discussed in prior topics such as the billboarding and skybox systems. Its main function is to handle and provide uniforms and draw calls for each of these systems.

# Critical Reflection

Reflecting on my project, I am pleased with both the visual outcomes and data flow of the system. However, the system is not without its flaws. Challenges I have faced that I have not been able to resolve are that of mapping coordinates from a cube to a sphere resulting in stretched textures at high gradients in cases near to the edges of the cube faces when using Tri planar projection techniques. For this I found many cases online with similar problems and no viable solutions to this problem. As such a future enhancement would be to attempt other mapping techniques possibly using Euler inference instead to map my coordinate systems to UV coordinates.

Further enhancements could be; The implementation of IBL across all rendering shaders. Refining my Rayleigh scattering algorithm for the atmosphere renderer to better interpret realistic atmosphere effects. Transitioning my texture arrays for clouds to a texture atlas so that I can render a larger data set of textures within blender to provide a seamless texture switching system. And the implementation of a particle-based weather system that could derive weather data from temperature and precipitation textures to simulate realistic weather behaviour.

Moving forward I intend to continue to expand and refine my system to improve my shader and coding skills with the goal being to create a system capable of rendering a realistic and performance aware method of real time universe simulation.

# Bibliography

[1] Brook Taylor, 1715, Methodus Incrementorum Directa & Inversa: <https://books.google.co.uk/books?id=r-Gq9YyZYXYC&printsec=frontcover&redir_esc=y#v=onepage&q&f=false>

[2] 2004, Paul Marino, 3D Game-Based Filmmaking: The Art of Machinima

[3] Greenberg, D. P. (n.d.). A framework for realistic image synthesis. *Communications of the ACM, 42*(8), 44–53. Retrieved 4 22, 2024, from http://www.graphics.cornell.edu/pubs/1997/GTS+97.pdf

[4] Ken Perlin, Noise hardware. In Real-Time Shading SIGGRAPH Course Notes (2001), Olano M., (Ed.).