Advanced Graphics and Real Time Rendering (Semester 2)

Anthony Sturdy

Staffordshire University

Terrain Rendering, Tessellation, and Skeletal Animation

6th May 2022

# Introduction

This project aims to explore real-time rendering features such as multiple types of terrain mesh generation (with procedural heightmap generation), hardware tessellation, and skeletal humanoid animation.

Various stages of the Graphics Pipeline are utilised for the implementation of these features, such as the Hull and Domain stages which are used for tessellation and modifying applying the heightmap transformations for rendering terrain. Terrain rendering uses multiple algorithms, one for generating a plane mesh where vertex positions are altered in the Domain shader, and another uses the marching cubes algorithm to allow for more versatile and interesting terrain featuring caves and overhangs.

Skeletal animation is implemented using the MD5 file format. Models and animations are split into separate files which contain information regarding the bone structure and positioning data. Vertex positions are then calculated at runtime using the loaded bone/joint data.

# Implementation

## API

The graphics application programming interface (API) chosen for the development of this application is DirectX 11, paired with DirectXTK (DirectX Tool Kit), which provides a collection of helper classes for writing C++ DirectX code (Microsoft, 2022). DirectX 11 allows for applications to utilise graphics hardware via shaders, which are small programs written in a C-style language that are executed on the graphics processing unit (GPU) in parallel.

A common alternative to DirectX 11 would be OpenGL, which is very similar in features and performance, but offers multi-platform support whereas DirectX 11 is limited to the Windows operating system.

Other modern graphics APIs such as DirectX 12, Vulkan, and Metal offer the developer lower-level control of the rendering process. The low-level control can lead to increased performance due to the mitigation of driver overhead, where the driver software can make less assumptions about the desired functionality, thus reducing CPU load (Furmaniak, 2018). However, the amount of control offered by these APIs often leads to more verbose code.

## Graphics Pipeline

This application utilises multiple stages of the DirectX 11 graphics pipeline (appendix 1) for the implementation of various features. The programmable stages DirectX 11 offers are the Vertex Shader, Hull Shader, Domain Shader, Geometry Shader, and Pixel Shader. Non-programmable stages include the Input-Assembler stage, Tessellator stage (between Hull and Domain), Stream Output stage (after Geometry), Rasterizer stage (before Pixel), and finally the Output-Merger stage. This application takes advantage of all stages except the Geometry Shader stage. The programmable graphics pipeline in this application would be as follows:

|  |
| --- |
| Domain Shader  Pixel Shader  Hull Shader  Vertex Shader |

## Features

### Heightmap Generation

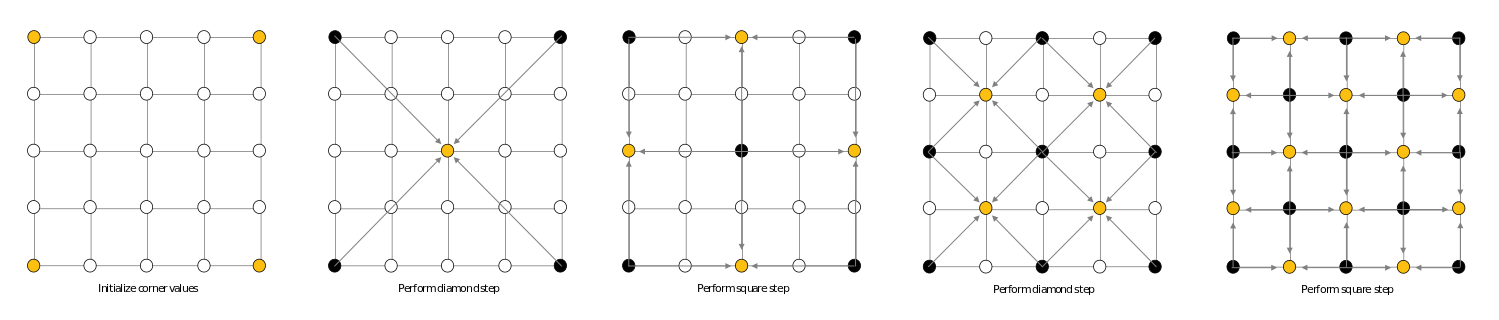
Heightmap textures can be generated or loaded at runtime, which are later sent to the domain shader to be applied to the plane mesh in order to render the terrain. The heightmap values are also used in the player controller to set the player’s height, which allows for the player to walk across the terrain. There are four options to create a heightmap, loading from a .RAW file, generation using the Diamond Square algorithm, generation using the Fault Line algorithm, and generation using multiple players of Perlin noise.

#### Loading From .RAW File

Heightmap data is read from a .RAW file by reading a stream each byte of the file into an unsigned char vector, the heightmap size is then determined by calculating the square root of the number of unsigned chars loaded into the heightmap vector, as .RAW files do not provide a header containing this information.

#### Diamond Square

The diamond square algorithm which generates a terrain effect and can be used to procedurally generate a heightmap within this application at runtime. The algorithm iterates over a 2D grid in an alternating diamond / square shape, generating and filling the grid with terrain data until each grid cell is filled. See the diagram (appendix 2) below.



The algorithm requires a 2D array of size (for example, 129x129, 257x257, 1025x1025) and begins with pre-seeded values in the four corners of the array, then iterates over the grid progressively taking smaller step sizes until the grid is filled (Mecom, 2015).

For the diamond stage, the algorithm finds the midpoint and sets the value to the average of the four corner values, plus a random value, the strength of which lessens as the algorithm progresses. The square stage then takes a diamond, finds the midpoints, and averages the values of the four points forming the corners of the diamond, plus another random value of diminishing strength. These two stages are repeated, decreasing the step size upon each iteration, until the grid is entirely populated (Mecom, 2015).

#### Fault Line

Fault line is a terrain generation algorithm which, upon each iteration, divides the terrain into two randomly sized sections, divided by a straight line, and displaces the height upwards of one half while the other half is displaced downwards (Fernandes, n.d.). See the diagram (appendix 3) below, which demonstrates the algorithm over different iterations.

Shape

Description automatically generated

The implementation of this algorithm loops for a pre-determined number of iterations, and for each iteration, selects two random positions on the heightmap grid, of which represent two points of an infinite line intersecting the grid. Then, each pixel is either displaced upwards, if the following equation returns true, or downwards, should the equation return false. In the equation represents the first point in the line, and is the second point in the line (the order is not relevant, though, the positions of each point do determine the orientation of the line, and whether the point is being tested for being, visually, to the left or to the right of the line). is the position of the current pixel being evaluated.

#### Perlin Noise

Perlin noise is a smooth noise where similar inputs result in similar outputs, as opposed to random noise, where similar inputs can result in completely different results. See appendix 4 for an example of 2D Perlin noise.

The Perlin noise algorithm consists of two main stages. The first stage generates a repeatable pseudorandom value for every integer position in 3D space using a hash function. The second stage utilises the pseudorandom integer to access a table of 3D gradient vectors, which is used to calculate a scalar value by calculating the dot product between the gradient value and fractional position within the noise space. The final value is obtained by interpolating between noise values for each surrounding eight points in space (Lefebvre et al., 2005).

Perlin noise can be sampled for terrain generation, from it’s appearance in appendix 4, it is clear that it would produce a somewhat terrain-like result being used simply with 1 sample, but to achieve a more realistic result, multiple Perlin noise samples of increasing frequency and decreasing magnitude are applied. This leads to the initial iterations greatly influencing the terrain, creating the overall shape of hills, with later iterations adding finer detail to the terrain. The following diagram (appendix 5) demonstrates this effect with three iterations of noise.

A picture containing furniture

Description automatically generated

### Terrain Rendering

#### Plane Mesh

#### Marching Cubes

### Skeletal Animation

#### Mesh Loading

#### Mesh Animating

# Critical Evaluation

# Bibliography

Fernandes, A. (n.d.). *The Fault Algorithm*. Retrieved May 1, 2022, from http://www.lighthouse3d.com/opengl/terrain/index.php?fault

Furmaniak, L. (2018). *Stuck on OpenGL ES? Time to move on! Why Vulkan is the future of graphics*. https://www.imaginationtech.com/blog/stuck-on-opengl-es-time-to-move-on-why-vulkan-is-the-future-of-graphics/

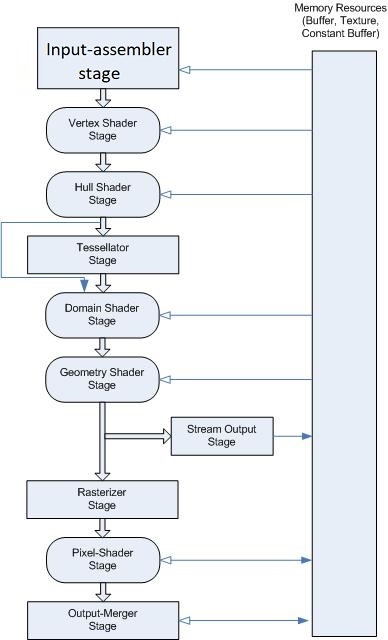
Lefebvre, S., Hornus, S., & Neyret, F. (2005). *GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation*.

Mecom, J. (2015). *Procedural Terrain Generation: Diamond-Square*. http://jmecom.github.io/blog/2015/diamond-square/

Microsoft. (2022). DirectXTK. *GitHub Repository*. https://github.com/microsoft/DirectXTK

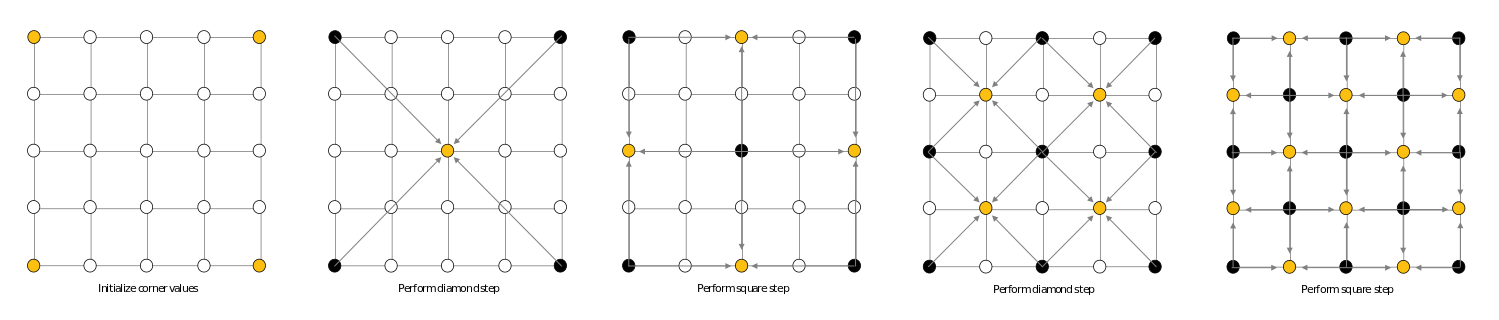
# Appendices

## Appendix 1:



Microsoft (2022). Available at: <https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline>

## Appendix 2:



Mecom (2015). Available at: <http://jmecom.github.io/blog/2015/diamond-square/>

## Appendix 3:

Shape

Description automatically generated

(Fernandes, n.d.). Available at: <http://www.lighthouse3d.com/opengl/terrain/index.php?fault>

## Appendix 4:

A close up of a person's face

Description automatically generated with medium confidence

(Biagioli, 2014). Available at: <http://adrianb.io/2014/08/09/perlinnoise.html>

## Appendix 5:

A picture containing furniture

Description automatically generated

(Red Blob Games, 2015). Available at: <https://www.redblobgames.com/maps/terrain-from-noise/>