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 applying the heightmap transformations for rendering terrain. Terrain rendering has the option of two different 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 and supports the MD5 file format. Meshes and animations are split into separate files which contain data 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 stage. The programmable sections of the graphics pipeline in this application are as follows:

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

Tessellation can be applied to objects in this application and is used for LODs when rendering terrain. Tessellation is configured in the hull shader, which is executed for each control point. Following the tessellator stage, the domain shader is then executed for each patch and modifications can be applied.

Tessellation can be applied by increasing the tessellation factor in the Hull shader, then the tessellator stage, following the Hull stage, will subdivide each triangle based on the tessellation factor. The tessellator stage is fixed-function and generates sample points for a given patch based on the tessellation factor (Nießner et al., 2015).

## Features

### Component System

This application features a component system which allows for objects to take ownership of generic functionality which can be combined for complex functionality. The component system helps to keep the codebase modular and promotes reusable code. The component system works using template functions and an unordered map, which uses the type\_id of the passed type for a key. The templated functions use modern C++ functionality to ensure the base class of any class passed to the system is of type Component, this prevents any object type from being passed to the system which could lead to errors when compiling. C++ requires templated functions to be entirely within the class header file, otherwise linker errors will occur unless each use-case is explicitly instantiated within the source file.

### Heightmap Generation

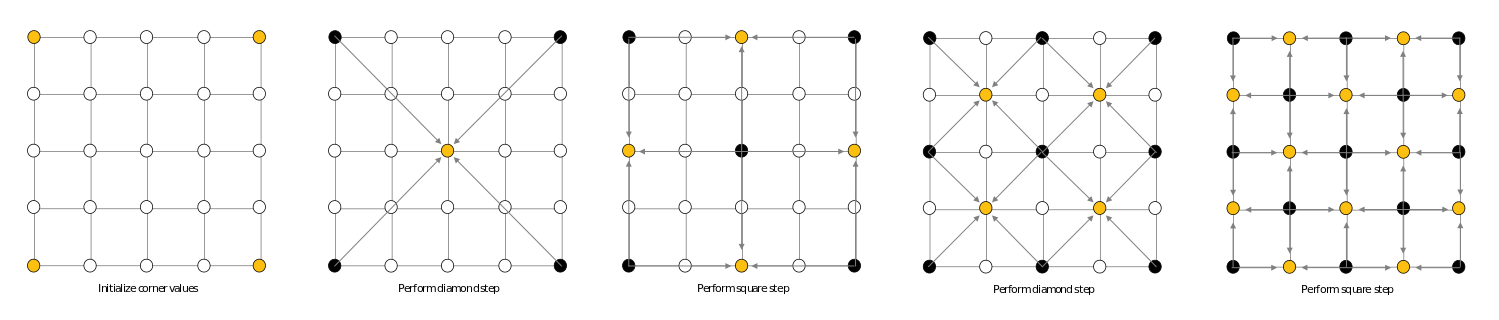
Heightmap textures can be generated or loaded at runtime, which are later sent to the domain shader to be applied to the plane mesh in order to render the terrain. The heightmap values are also used in the player controller to set the player’s height, which allows for the player to walk across the terrain. 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 bytes from 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 produces a terrain effect and can be used to procedurally generate a heightmap 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 grid with dimensions of (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, 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 its 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

Once a heightmap has been generated or loaded, a terrain can be rendered using two different techniques which are implemented into this application. The first technique is using the domain shader and a plane mesh. The second does not use the domain shader for rendering terrain, but instead generates a marching cubes mesh which allows for the terrain rendering with caves and overhangs.

Level of detail (LOD) can be applied to the terrain via tessellation (hull and domain shaders), the amount of LOD applied is determined by the distance from the current patch being tessellated in the hull shader, to the camera. Values passed from the GUI can determine the LOD spread and maximum intensity.

#### Plane Mesh

For this terrain rendering method, a flat plane is generated at a size determined via the GUI. When the graphics pipeline reaches the Domain shader stage, the heightmap is sampled as a texture (which is bound to the GPU as a shader resource view), and the height of the current vertex is displaced upwards by the terrain height value (set in the GUI) multiplied by the current heightmap sample. This renders the terrain with the heightmap applied, which can then be textured and shaded in the pixel shader. The benefit of applying the heightmap in the domain shader, rather than baking the height values into the mesh upon the plane being generated, is that when tessellation is applied, the terrain becomes more detailed. Should tessellation be applied to a mesh which has pre-determined vertex positions, the triangle count of the mesh will increase, but the mesh will not gain any more visual detail.

The plane mesh is generated by performing a nested for loop through the plane size and creating a vertex for each iteration position. For every iteration, except for the final iteration on each axis, indices are generated for each quad which makes up the larger plane.

A close-up of a leaf

Description automatically generated with medium confidence

#### Marching Cubes

Rendering terrain using a plane mesh allows for fast mesh generation and rendering but lacks the capability to render overhangs or caves within the terrain. Marching cubes can be slower to generate a mesh but allows for overhangs and caves to be implemented into the terrain.

Marching cubes renders the terrain using a terrain sampling function, which returns whether a given 3D position is solid, or not. The function utilises the heightmap to determine whether the given position’s height is above or below the terrain, or whether it is inside of a cave/overhang. Perlin noise is used for creating caves. The advantage to using a single function to determine whether a point is inside the terrain or not, is that it allows for as much, or as little, detail as the user would like. Generating a marching cubes mesh from voxel data would limit the resolution of the mesh to what is stored within the data and would use much more memory.

The marching cubes algorithm has five key stages (Preim & Botha, 2007):

1. Determine the case index of each cell
2. Determine the intersected edges
3. Compute intersections using linear interpolation
4. Triangulate intersections
5. Calculate surface normals

The algorithm iterates over a uniform grid of cubes, if all 8 points/vertices of the cube are either solid or empty, no triangles are created for that cell. Otherwise, the cube uses a look-up table (as there are possible combinations) to determine which triangles should be constructed based on which vertices of the cell are solid. Another table of indices can then be referenced to triangulate the intersections (Fisher, 2014).

Once the mesh has been triangulated, surface normals are calculated using the cross product of two edges of a triangle. The following equation represents the calculations needed for calculating the surface normal. and are the directional vectors of two edges of a triangle, , , and are each vertex position of a triangle, and N is the final normal value.

An example of marching cubes terrain meshes. Caves can be seen below the terrain.

|  |  |
| --- | --- |
| Low resolution Marching Cubes mesh | High resolution Marching Cubes mesh |

### Skeletal Animation

Animation is implemented using the MD5 file format, which has the mesh and animation in separate files. The MD5 format does not store vertex positions, but instead calculates the vertex positions at runtime based on the skeletal data.

#### Mesh Loading

MD5 mesh files are stored in plain text, which simplifies the process to load the data into relevant structures created for storing bone hierarchy data, and the application can use a standard std::wifstream to open the file and the right pipe operator (>>) to copy the next string from the file into a usable variable.

Firstly, the mesh header information is loaded which contains information such as the number of joints, number of sub-meshes, and MD5 version. Then the joints data are each loaded into a Joint object which contains the joint name, parent index, position, and orientation. Once all joints are loaded, each sub-mesh is loaded to ModelSubset object which contains a Mesh object and a std::vector of Weight objects, which holds data such as the joint index, bias, position, and normal. After the mesh data has been loaded into relevant structures, Vertex positions are calculated. Vertex positions are calculated by firstly calculating the Weight’s position based on the Joint’s position and orientation, which is calculated by iterating through each Weight that the current vertex is bound to, calculating the Weight’s position first in Joint space, then translating to object space. That position is multiplied by the Weight’s bias and added to the vertices final position.

After all sub-meshes have been loaded and vertex positions calculated, normals can be calculated using the same method used for generating the Marching Cubes surface normals. The final rendered object can be seen in the following image.

A picture containing tripod

Description automatically generated

#### Mesh Animating

Loading a MD5 animation file is a similar process to loading a mesh file, the file is in plain text which makes it simple to load into custom data structures. The structures introduced for the implementation of animation are: BoundingBox, which stores two min and max positions which can be used for simple collision detection, though collision is not implemented into this project so this is not utilised; FrameData, which stores the frame ID and a vector of floats to store the frame data; AnimJointInfo, which stores the name of the joint, parent index, flags and starting index; and finally ModelAnimation, which stores all data for the loaded animation, including the number of frames, number of joints, frame rate, number of animated components, vectors of AnimJointInfo objects, BoundingBox objects, Joint objects (base frame joints), FrameData objects, and a two-dimensional vector of Joint objects (frame skeleton).

To animate the mesh, in the component’s Update method, the current frame is retrieved by flooring the current frame time (e.g., 3.5457 will become 3) and subtracting the floored current frame time by the actual frame time returns only the fractional part of the number (e.g., 0.5457), which is used as the interpolation value between frames.

The current and next frame can then be retrieved, and the orientation of each joint interpolated between and stored in a separate vector for the updated skeleton. Finally, the new vertex positions, based on the interpolated skeleton, can be calculated similarly to how vertex positions are calculated upon loading the model.



# Critical Evaluation

This project has enabled research and implementation of a multitude of advanced real-time rendering features such as terrain rendering with both a plane and marching cubes mesh, utilised the hull and domain stages of the graphics pipeline for tessellation and terrain LODs, procedural heightmap generation, and skeletal animation.

The terrain rendering has been successful, though the marching cubes algorithm can become extremely slow when generating a moderately high-resolution mesh, which may benefit from multithreading or splitting the mesh being generated into smaller chunks.

The heightmap generation algorithms also produce interesting results, but could be improved, and would benefit from tweaking the parameters which could remove some artefacts leading to strange and unrealistic terrain, which, for example, can contain noticeable patterns.

Animation functions as desired, but the downside is the file format used (MD5) is not a common format, meaning the number of assets available for testing are extremely limited compared to other, more common, file formats. An improvement on this project for future development would be to add support for loading more file types which would greatly increase the number of assets compatible with this engine.

The development of this project as helped me to learn about advanced rendering features, having no prior knowledge or experience in using hull or domain shaders, or implementing skeletal animation to an engine, there has been a large learning curve to overcome. The project also allowed me to explore features of C++ I had not yet had experience with and helped me to become more proficient with C++ templates.

# Bibliography

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

Fisher, M. (2014). *Marching Cubes*. https://graphics.stanford.edu/~mdfisher/MarchingCubes.html#:~:text=Marching%20cubes%20is%20a%20simple,a%20region%20of%20the%20function.

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/

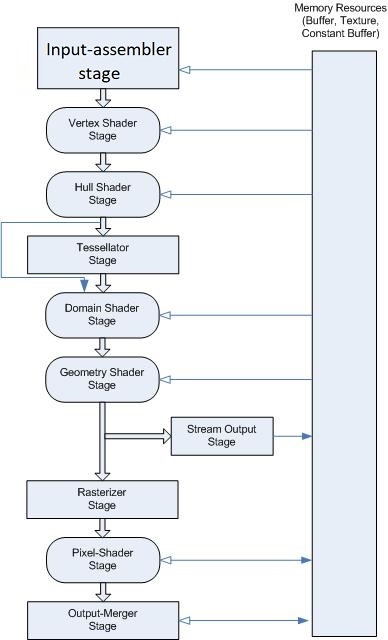
Microsoft. (2022). DirectXTK. *GitHub Repository*. https://github.com/microsoft/DirectXTK

Nießner, M., Keinert, B., Fisher, M., Stamminger, M., Loop, C., & Schäfer, H. (2015). Real-time Rendering Techniques with Hardware Tessellation. *Computer Graphics Forum*. https://graphics.stanford.edu/~niessner/niessner2015survey.html

Preim, B., & Botha, C. P. (2007). *Visual Computing for Medicine: Theory, Algorithms, and Applications*.

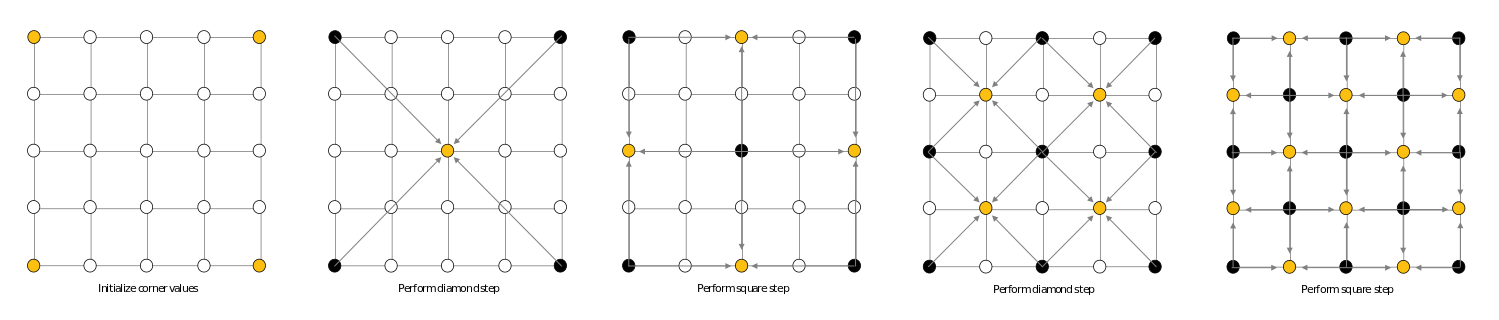
# Appendices

## 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/>