FC6P01 Final Project

Pretty Terrain

3D Terrain Development Software

Interim Report

Student: Kim Kane

Student ID: 15021826

Course: BSc Computer Games Programming

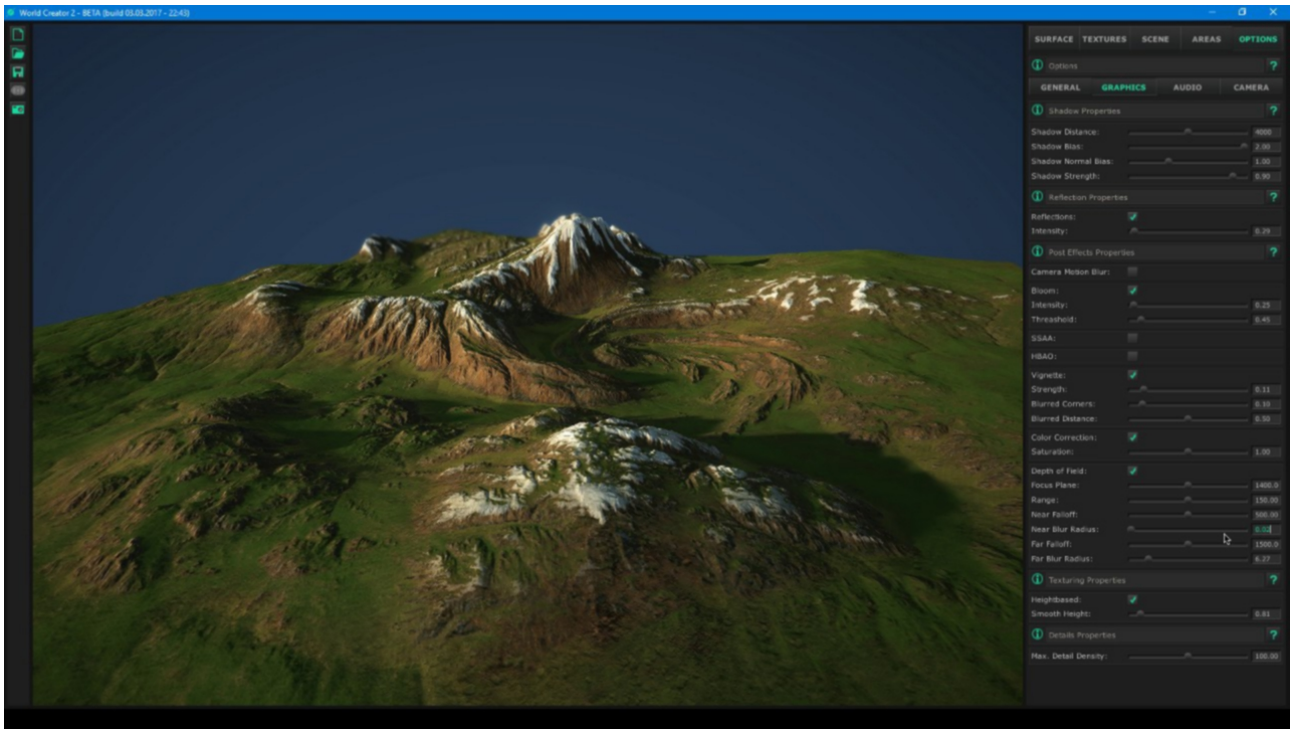
Module Leader: Fiona French

Contents

Introduction

Project Concept

For my dissertation I have decided to develop a 3D terrain creation tool, to be used in conjunction with my own game engine. This tool will allow the user to generate 3D terrain procedurally[1] or by using heightmaps[2].



Example of World Creator[3], a popular terrain generation tool created by 3D software development company BiteTheBytes.

The software will work in conjunction with my own game engine, The Pretty Engine, built using the SDL2[4] and OpenGL[5] API's (application programming interface).

Pretty Terrain will be a user-friendly tool to create, modify and save open-world 3D terrain, which can then be easily loaded into a game. The software will support heightmaps[2] (designed) and procedural[1] (randomized) terrain generation primarily.

The user will create their desired 3D terrain using the software. Once they are happy with the terrain they create, they can save the terrain data to a file, which can then be easily loaded into their game.

Similar to the 3D model import library, Assimp[6], the user will have access to the internal data of the terrain. This will enable them to modify or manipulate the terrain data further, through programming.

Project Goal

I chose to focus on terrain generation for my dissertation, as I thrive to be a Level Designer. I discovered I had a passion for this area specifically, in my second year at London Metropolitan University.

Although terrain generation is not strictly Level Design, I believe it is important that I discover, research and learn as much as possible in this area. This will increase my knowledge on how levels are developed as a whole and will aid my success in future career prospects.

I wish to showcase my software on my portfolio. I will provide complete source code and documentation on the software, as well as tutorials and articles on the development process. This will appeal to future potential employers, but it will also serve as a great resource for other Game Developers and Level Designers.

Aims and Objectives

Project Aims

- To show potential employers my versatility. Rather than being pinned as a programmer, I wish them to see I have a good eye for detail, a knack for design and that I specialise in something, in retrospect this will broaden my horizons.

- Prior to developing the terrain software, I will first finish my own game engine. Once I have finished my game engine and have all necessary aspects covered and in good working order, I will then develop the terrain software using my engine. This should make the development process of my terrain software extremely quick, easy and non-repetitive.

- I am a perfectionist when it comes to designing and creating levels – I wish to showcase this and generate beautiful open-world realistic environments, where I am able to show my level design skills, but also my understanding of how levels in games are generated.

- I am passionate about level design, but also helping people and wish to learn as much as I can about the field, so that I can help other games developers in future.

- I enjoy graphics programming and enjoy creating new, interesting effects that can really show off my environments, such as: skyboxes, ground fog, rain, sunrise/sunset effects, lens flare, shadows, etc. I wish to showcase this also – I thrive to make environment's look beautiful to the user and improve their overall experience of the game.

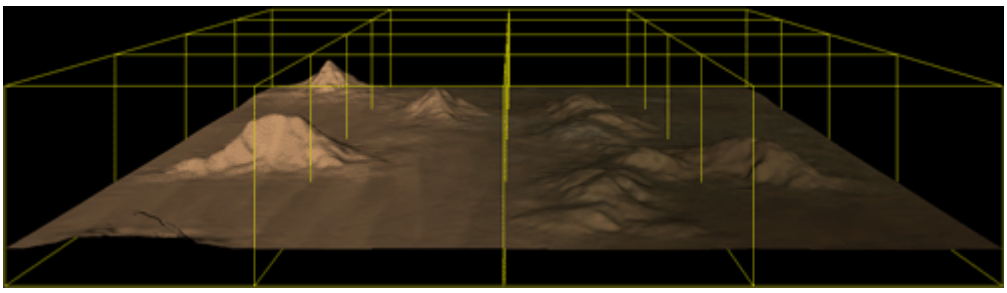
A screenshot of the open-world terrain from the game ARK: Survival Evolved. Environmental effects can vastly compliment terrain.

Project Objectives

LO1: To obtain a full understanding on how terrain is generated in games.

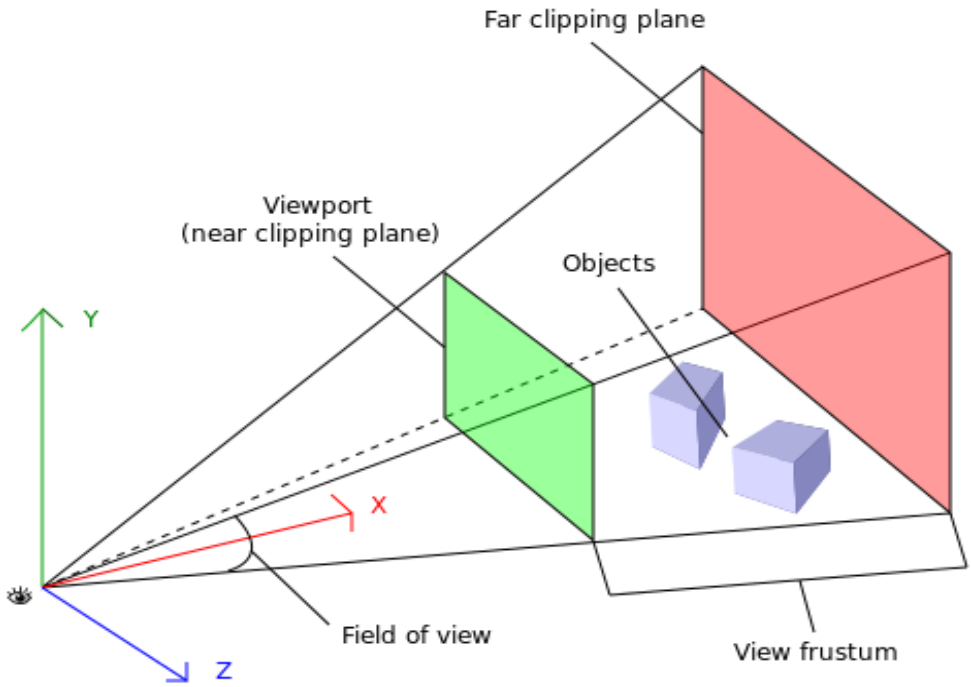
Optimization is important in generating open-world terrain. Small terrain can be rendered as-is, however larger terrain rendering can cause the game to slow down. This is due to the vast amount of vertices[7] being rendered at once. In order to accomplish large terrain being rendered, without the game slowing down (lagging), I need to fully understand how to break the terrain up into small sections and only render small blocks of terrain at a time (only what the user can see). There are many ways to achieve this, I have discovered, but my aim is to find the best, most practical method for this project, so there will be a vast amount of experimenting and unit tests. This is the most important task, as without optimization the terrain will not render.

Quad-tree's[8] can be used to break the terrain up into small chunks. They are relatively fast and simple to code. I used a quad-tree the first time I generated a terrain using a heightmap[2].



An example of a terrain quad-tree being generated.
Image from RasterTek Tutorial 5: Quad-tree's[9].

Quad-trees work in conjunction with frustum culling[10]. Frustum culling is the method of only rendering that which can be seen within the viewport.



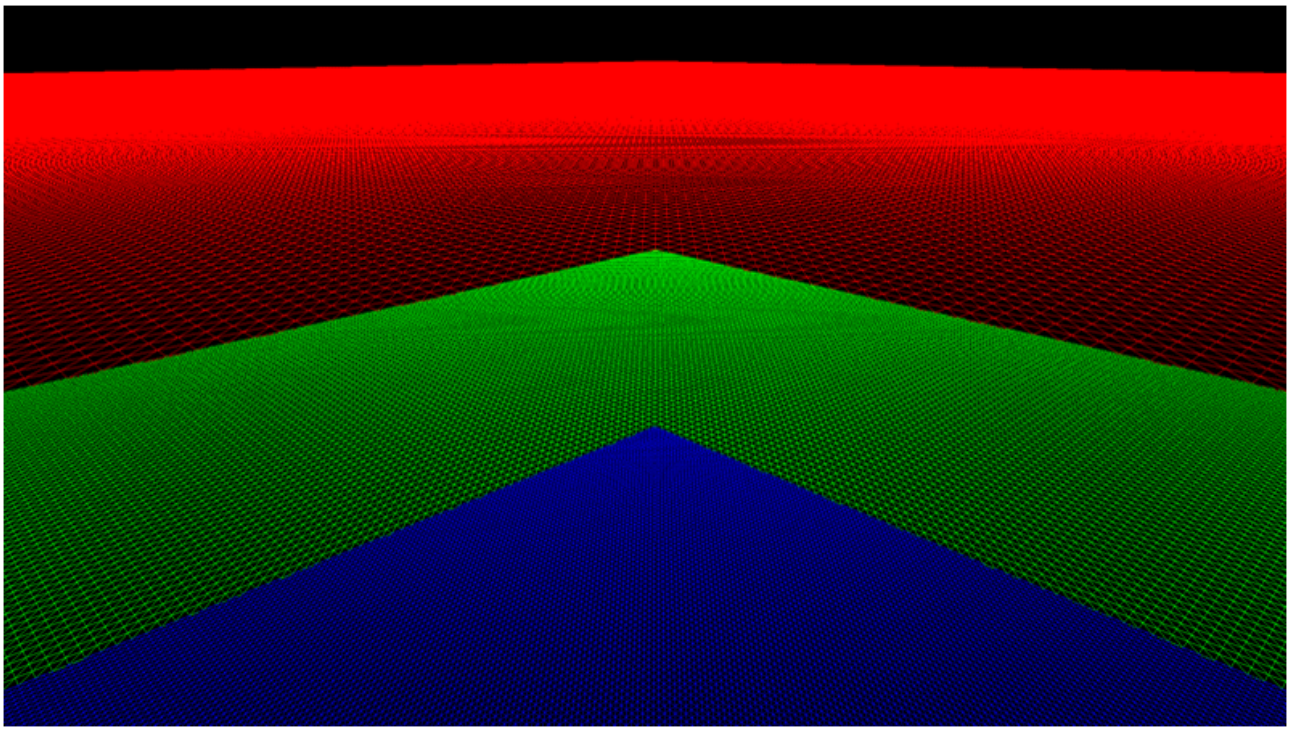
The image above shows how frustum culling is achieved.
Everything within the view frustum is rendered and anything outside it is not.
Image credits: https://rhiannongriffiths.wordpress.com/2012/10/11/game-platforms

Occlusion culling can be used in conjunction with frustum culling to further improve performance and speed of rendering. Where frustum culling culls whole objects, occlusion culling culls specific vertices[7] (think of an object hidden behind another object). I will use OpenGL's built-in occlusion queries[11] to implement occlusion culling for this project.

Using just a quad-tree to render terrain would not be enough. As the terrain grows in size, we have to think about the level of detail of the terrain from the camera's location to the far (clipping) plane.

This is vital, as terrain being rendered with extensive detail through-out will counteract the optimization we achieved when using a quad-tree.

To resolve this, we can use Node-Based LOD (Level of Detail) in conjunction with quad-trees.



An example of how node-based LOD may look once generated. Image from RasterTek Tutorial 18: Large Terrain Rendering[12].

The basis of node-based LOD is that the terrain nodes closest to the camera will be rendered in high detail. The level of detail of the terrain

will decrease depending on the distance of the terrain nodes from the camera.

A complete planetary scale LOD terrain generation video example can be seen at LeifNode.com[13].

Background (Daemon)[14] threads are used in conjunction with node-based LOD for fast loading and unloading of nodes.

LO2: To learn about procedural[1] terrain and write an algorithm that generates terrain randomly.

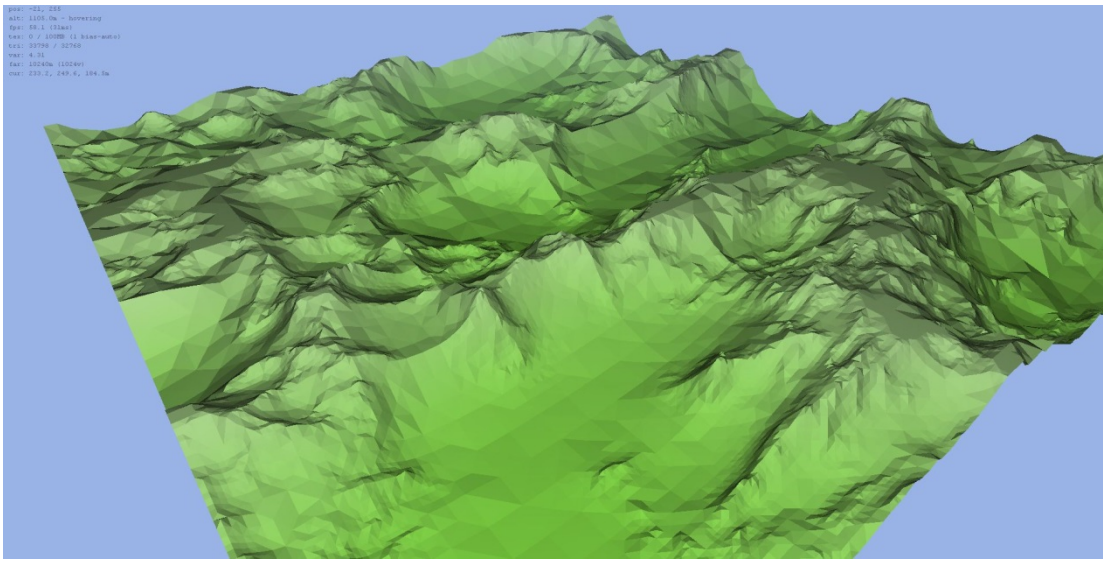


Image above shows a terrain that has been procedurally generated.
Image credits: https://sainarayan.me/2015/05/13/procedural-terrain-generation-using-perlin-noise-and-machine-learning

Unlike heightmaps[2], procedural terrain is created via programming through the use of an algorithm. In terms of terrain, it is the method of interpolating the vertices of the terrain mesh, allowing the user to be in full control over the overall shape of terrain (flat, mountainous, rugged, smooth).

Procedurally generated terrain allows for terrain randomization, tiled terrain and in more advanced algorithms it can be used to distinguish the correct placement of roads, streams, oceans, etc. I will keep my algorithm simple, as I am only interested in generating random terrain.

LO3: Understand how to create my own images through programming.
I hope to give the user the option to save the terrain data they generate to a heightmap[2] gray-scale image file. For this I will need to learn about saving vertex data to a file, converting it into pixel data and so on. It sounds simple, but with image headers, compression, etc. taken into account, it can be quite a complex procedure.

LO4: Understand and create a robust UI with user controls.
I hope to give the user the option to edit the vertices of the terrain in real-time (similar to the Computer Animation and Modelling Software, Maya[15]). For this, I will need to understand ray-casting[16] so I can project a ray into the screen using the mouse and know when the mouse is hovered over a particular vertex. I also need to think about undo/redo options.

LO5: Create my own parser for the terrain files generated.
I will use the serialization library, Cereal[17], for this specifically, but I need to create a wrapper class that parses the data correctly and efficiently. This will give me full control over what internal terrain data is saved and will be structured in such a way that the user can edit the file outside of the program. I wish to give the user ample control over their terrain.

LO6: Obtain a full understanding of graphics effects used most commonly in 3D games to enhance the environment.
Including ground fog, water, weather, post-processing (for minimaps), environmental factors i.e. wind, clouds, atmosphere, etc. This is so I can showcase the project in a polished way. I want my project to be fast and stable but I also want it to look appealing to the user.

LO7: Completely understand design patterns[18] and write re-usable code.
This will make it easier for me to update or modify code, if I decide to add or change anything inside my project.

Deliverables

Prioritised deliverables

- The sole purpose of my terrain generation software is to **provide an easy-to-use tool that allows for 3D terrain generation**, for games developers, specifically level designers.
- **The user interface (UI) will be robust, user-friendly and extremely simple**. The programming knowledge required of the user will be minimal.
- The software will **support procedural[1] and heightmap[2] terrain generation**, through the use of algorithms and serialization.
- **The terrain will be a maximum size of 4096 x 4096 pixels and be fully optimized**. I will achieve this by breaking the terrain up into smaller chunks and make use of frustum culling[10], occlusion culling[11] and quad-trees[8].
- **The terrain file will include collision data**. This will allow the user to check for collision against the terrain easily. I will provide a function that checks for collision, using a popular formula known as Barycentric[19].
- The retrieval of terrain data **will work in the same way as the Assimp[6] library**. Assimp retrieves model data from a file and stores this data into containers (array's). The user can then access all the data relevant to the model, e.g. normals[20], texture coordinates[21], etc. I really like this design and believe it to be the easiest option for the user.
- **The terrain software will be extremely fast**. I am currently running many unit tests and keeping track of my frame rate, as I do not want the terrain software to lag.

Additional features

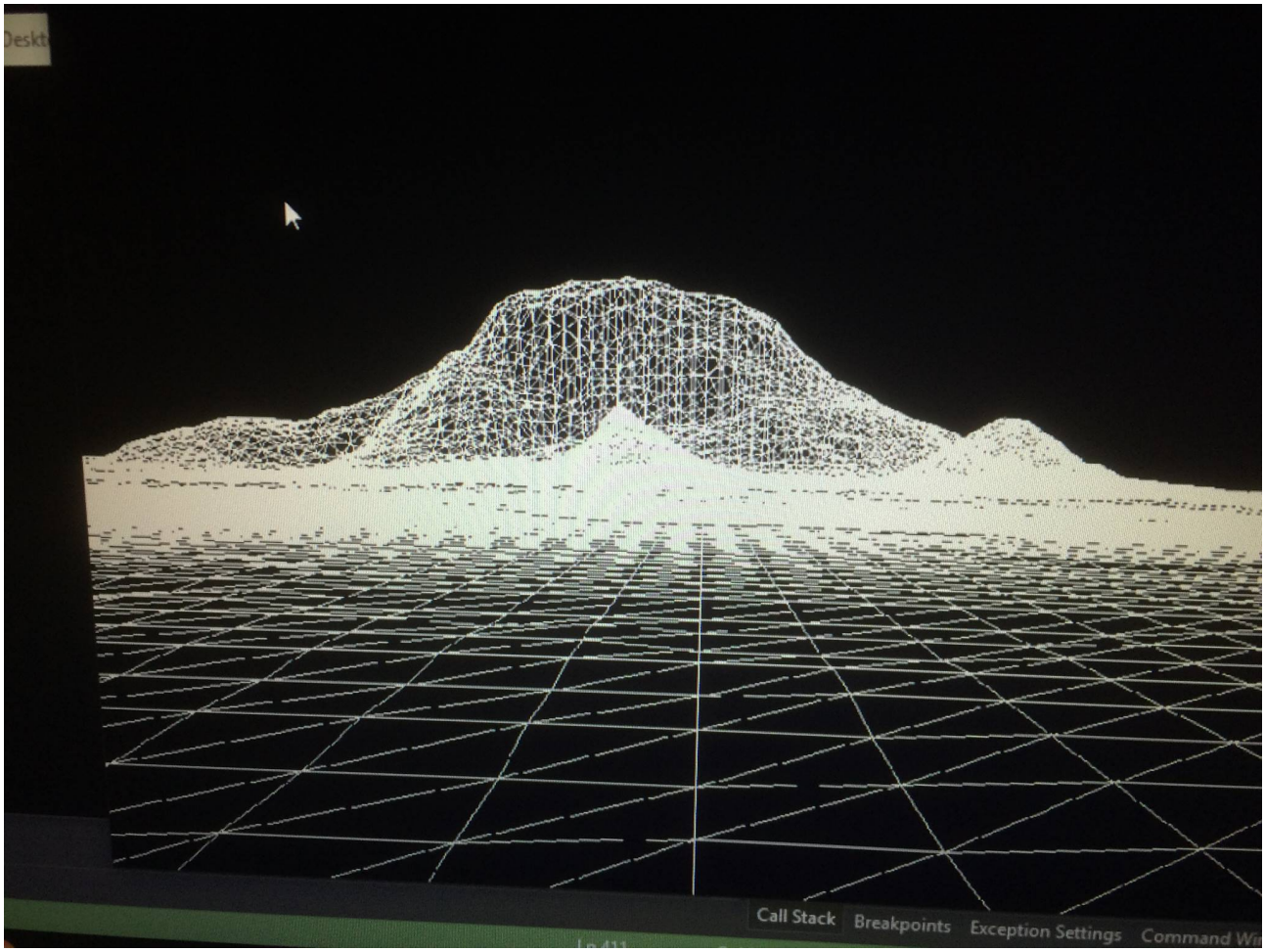
- I would like to include texture data within the terrain file. The user can load in a blendmap[22], as well as terrain textures and this information is also stored within the output file – allowing them to retrieve it easily.

- I would like to include image generation within the software. I wish to learn how to create a heightmap image through procedural terrain, as I believe this will be a nice additional feature.
- I would like to provide the user with singular or multi vertex[7] selection – allowing them to drag vertices around and adjust the mesh of the terrain in real-time.
- I would like to have a more advanced user interface (UI) – redo/undo buttons, paint tools, mesh editing tools, etc.
- I would like to incorporate shader effects so the user can see how the terrain would look under specific light.

Background

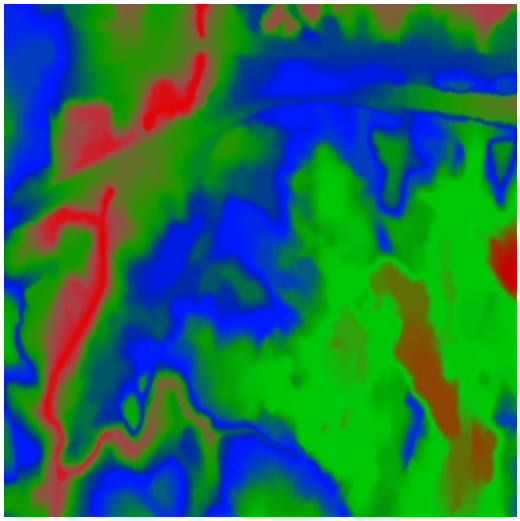
I began building the terrain software in June 2018 and have been working on it since. The software works in conjunction with my own game engine, which I began developing in September 2017.

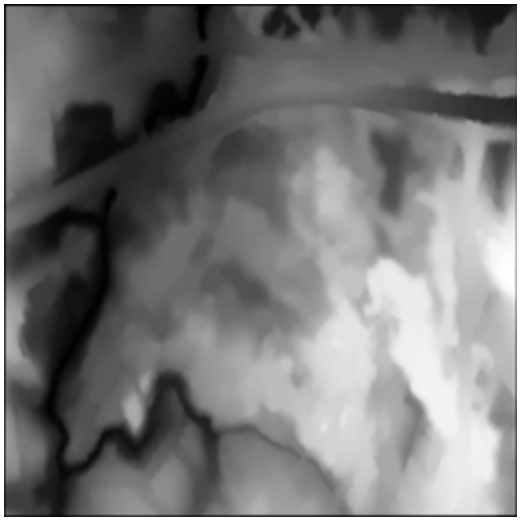
When I first began developing my game engine, I managed to generate an open-world terrain using a heightmap[2], complete with a blendmap[22], diffuse textures and normal textures for lighting.



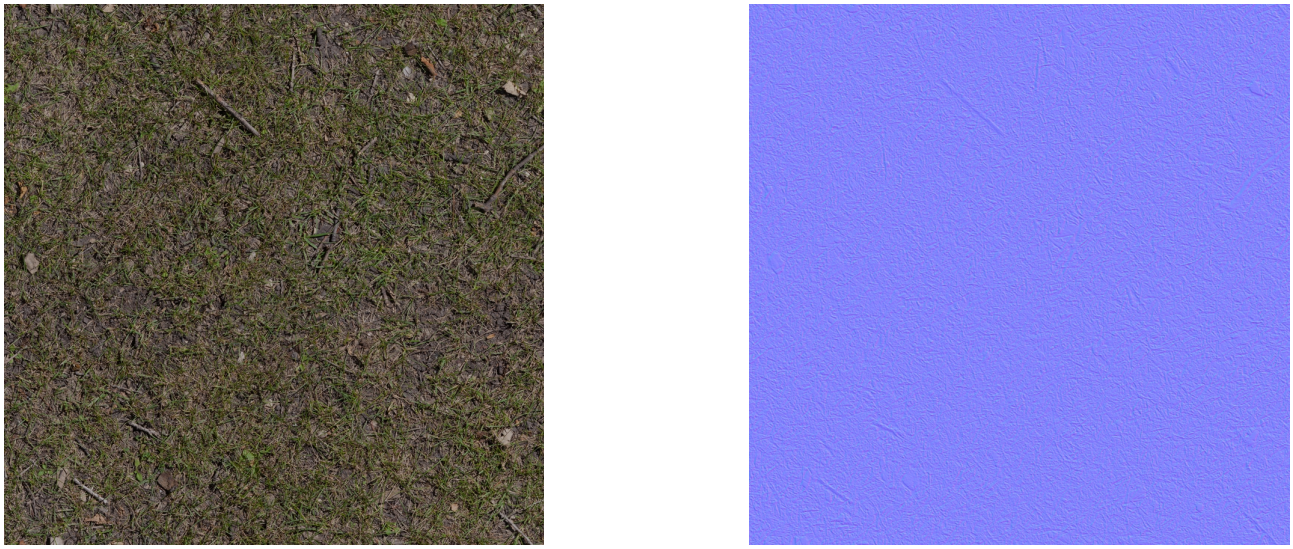
A screenshot of my first ever terrain generated using my game engine in 2017. I used a heightmap to generate the terrain. My engine was very basic at this point.

A screenshot of an OpenGL game I built using my game engine in 2018. Notice how my emphasis was on the terrain, minimap and environmental effects. You may see fog and a directional light imitating the shine from the moon.





The blendmap (left) and heightmap (right) I used to generate the above terrain.



The diffuse texture (left) and the normal map (right) I used for the terrain's background texture.
This texture covered the blue areas of the blendmap.

I calculated my own normals[20], tangents and bi-tangents[23] (necessary when using normal maps for lighting) and researched the Barycentric[19] formula in-depth for terrain collision.

```
/*****************************************************************************************************************
    Function that calculates terrain normals, using finite difference method
    References:
    https://en.wikipedia.org/wiki/Finite_difference_method
    https://www.youtube.com/watch?v=O9v6olrHPwI&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP&index=21
*****************************************************************************************************************/
void Terrain::CalculateNormals()
{
    int index = 0;

    //--- Neighbouring vertices - left, right, bottom and top
    struct { float l, r, b, t; } neighbours = { 0 };

    for (int row = 0; row < m_height; row++) {
        for (int column = 0; column < m_width; colu

            index = (m_height * row) + column;

            //--- We calculate the height of all 4 neighbouring vertices
            neighbours.l = FindHeightAtPoint(column - 1, row);
            neighbours.r = FindHeightAtPoint(column + 1, row);
            neighbours.b = FindHeightAtPoint(column, row - 1);
            neighbours.t = FindHeightAtPoint(column, row + 1);

            //--- Then create the normal from the data generated above
            glm::vec3 normal = glm::normalize(glm::vec3(neighbours.l - neighbours.r, 2.0f, neighbours.b - neighbours.t));

            m_map[index].normal = normal;
        }
    }
}
```

Image above shows my formula for calculating normals using the Finite Difference Method[24] (a fast way to calculate normals, very popular in terrain generation).

```
//--- References:
//--- http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/
//--- https://learnopengl.com/Advanced-Lighting/Normal-Mapping
//--- This could be made a lot cleaner in future - but I kept it simple so that we can see how the terrain is generated

//--- First Triangle
deltaPosition.first     = m_map[vertex.topLeft].position - m_map[vertex.topRight].position;
deltaPosition.second    = m_map[vertex.bottomLeft].position - m_map[vertex.topRight].position;
deltaTexCoord.first     = m_map[vertex.topLeft].textureCoord - m_map[vertex.topRight].textureCoord;
deltaTexCoord.second    = m_map[vertex.bottomLeft].textureCoord - m_map[vertex.topRight].textureCoord;

denominator = 1.0f / (deltaTexCoord.first.x * deltaTexCoord.second.y - deltaTexCoord.second.x * deltaTexCoord.first.y);
tangent     = denominator * (deltaTexCoord.second.y * deltaPosition.first - deltaTexCoord.first.y * deltaPosition.second);
bitangent   = denominator * (-deltaTexCoord.second.x * deltaPosition.first + deltaTexCoord.first.x * deltaPosition.second);

glm::normalize(tangent);
glm::normalize(bitangent);

vertices[index].position        = m_map[vertex.topRight].position;
vertices[index].textureCoord    = m_map[vertex.topRight].textureCoord;
vertices[index].normal          = m_map[vertex.topRight].normal;
vertices[index].tangent         = tangent;
vertices[index].bitangent       = bitangent;
index++;
```

The image above shows how I calculated tangents and bi-tangents for a terrain mesh.

```
/*********************************************************************************************
    BARYCENTRIC COORDINATES - A popular formula used for multi-height terrain collision

    The Barycentric formula is used to return a given point P(x,y) within a triangle
    The formula of which can be found here:
    http://2000clicks.com/MathHelp/GeometryTriangleBarycentricCoordinates.aspx

    There are much simpler ways of calculating the equation, however I wrote
    it out by hand in its simplest form to make it easier to understand.
    A simpler equation:
    https://gamedev.stackexchange.com/questions/23743/whats-the-most-efficient-way-to-find-barycentric-coordinates

    Given vertex points A, B and C of a triangle and a 2D point P(x,y) of an object
    within this triangle, we distinguish the P(x,y) Barycentric coordinates using
    the forumlas below. Barycentric coordinates are usually represented as t1, t2, t3
    l1, l2, l3 or more commonly v, w, u.

    Once we have these coordinates, we multiply them by the height of each
    vertex (A, B, C), which will return us the height of the terrain at that given spot.
*********************************************************************************************/
float Barycentric(const glm::vec3& A, const glm::vec3& B, const glm::vec3& C, const glm::vec2& P) {

    float determinant = (A.x * B.z) + (B.x * C.z) + (C.x * A.z) - (A.z * B.x) - (B.z * C.x) - (C.z * A.x);

    float v = (P.x * B.z) + (B.x * C.z) + (C.x * P.y) - (P.y * B.x) - (B.z * C.x) - (C.z * P.x) / determinant;
    float w = (A.x * P.y) + (P.x * C.z) + (C.x * A.z) - (A.z * P.x) - (P.y * C.x) - (C.z * A.x) / determinant;
    float u = 1.0f - v - w;

    return v * A.y + w * B.y + u * C.y;
}
```

The image above shows the Barycentric formula I wrote in its plainest form. This formula is the simplest way to check for collisions against the terrain.

Since I have tapped into terrain generation, my passion to learn about it has grown. I have been researching terrain and level design for a couple of years now and I believe I am at a stage where I feel confident enough to build a software program around this.

Work to date

It is vital that I complete development of my game engine prior to building the terrain software. I am making good progress with my engine and already have the most important aspects I will need for this project covered and in good working order, these are covered below.

Past Research

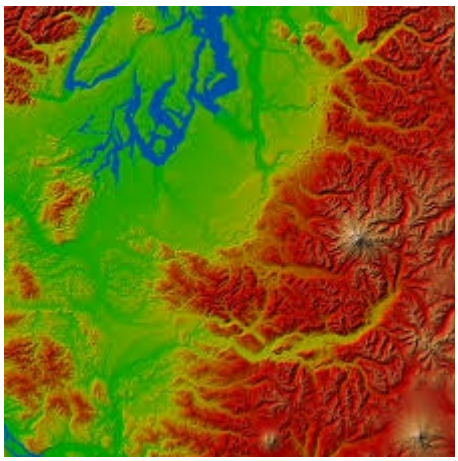
Terrain Optimization

I have discovered that the fastest way to render terrain is by using quad-trees[8] in conjunction with node-based LOD[12], OpenGL's built-in occlusion queries[11] and frustum culling[10]. This will speed up rendering massively, as occlusion queries disable vertices from being rendered if they are not within view or behind other objects. Frustum culling will add additional optimization as it will check the bounding boxes of the terrain prior to each and every vertex, reducing the number of checks being done per frame. Quad-trees combined with node-based LOD make for an extremely fast optimization technique, used to allow rendering of large terrain.

Batch Rendering

I have found a way to render unlimited textures to the scene extremely fast. By using a batch renderer[25], I will give full control of graphics rendering to the shader[26], allowing me to render unlimited textures per frame. A terrain can have multiple textures and to achieve this I use a

blendmap[22] to distinguish where on the terrain to render the other textures. Blendmaps are extremely useful for pathways and roads, etc.



An example of a blendmap. RGBA (red, green, blue and alpha) colour values of blendmap are used to determine where the terrain textures will be placed.

Image credits: https://www.cc.gatech.edu/projects/large_models/ps.html

With one blendmap in place, a terrain can have up to four textures, blended together via the fragment shader to create smooth interpolation between textures. Each texture of the terrain uses the RGBA colour's located on the blendmap to distinguish where it is placed.

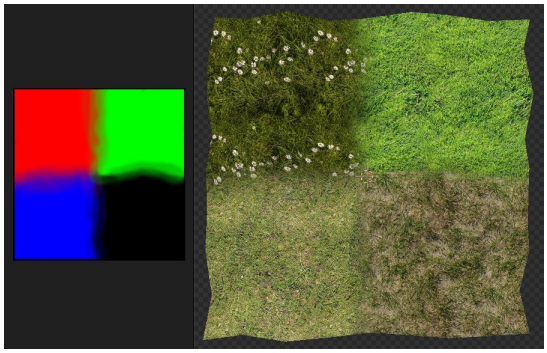


Image above shows terrain texture example. Each texture (commonly named base, red, green and blue) will be rendered at its matching colour value on the blendmap.

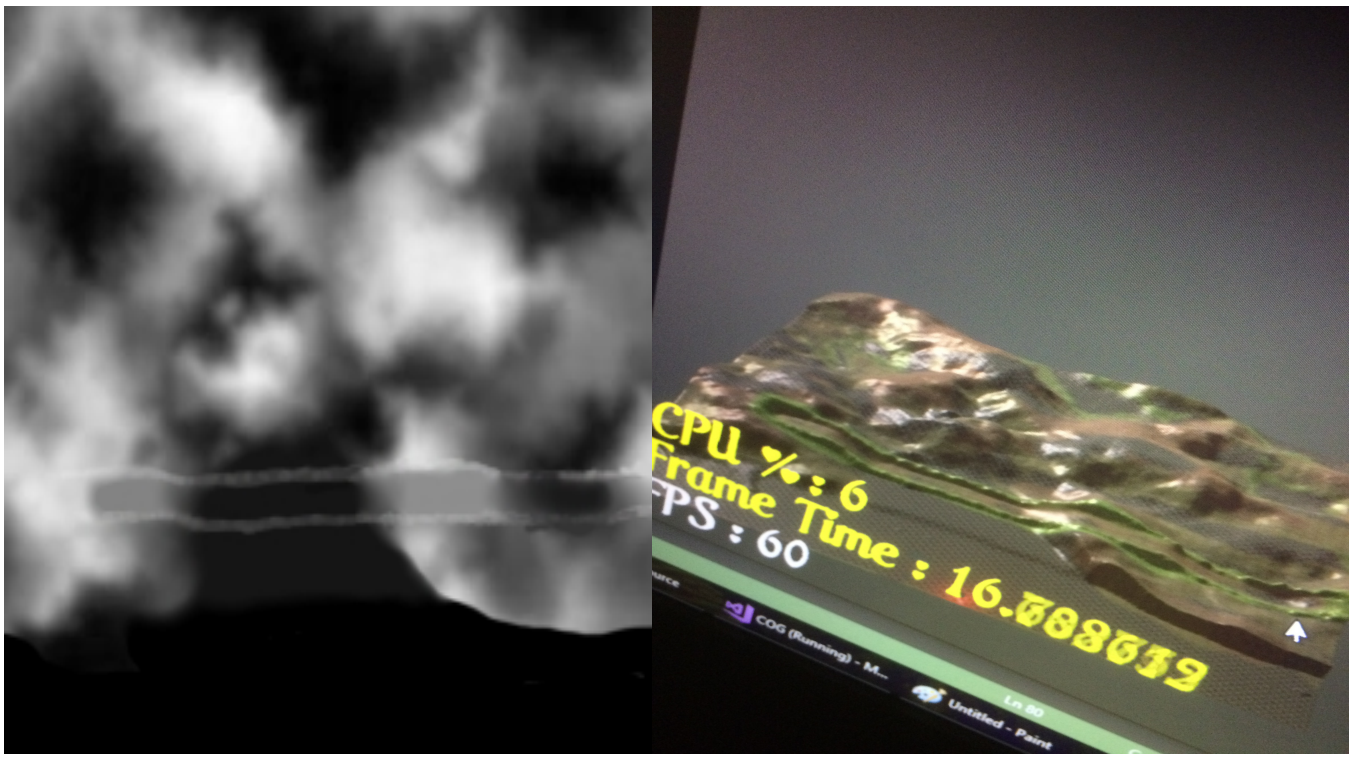
Without batch rendering, I would be limited to four textures per terrain. If I implement batch rendering successfully, I could potentially have multiple blendmaps and multiple textures.

Multi-threading and Daemon Threads

Daemon threads[14], or background threads, will be necessary for fast terrain LOD generation. I have discovered that these are most useful when running background processes. Daemon threads can also be used for loading screens, or even file dialogs. When the user opens the file dialog in the terrain software, they can still edit their terrain in real time. With use of daemon threads, I could accomplish extremely fast loading times and allow the user to work on multiple things at once. I have to be sure I implement them correctly however, or I run the risk of severe program crashes. For this to be successful I would need to keep a reference count[27] (similar to that used in shared pointers[28]) of all daemon threads currently active/running and only once they have finished will I shut down the main program.

Heightmap Generation

I have done a great amount of research into heightmap[2] generation and thus far have created two games with 3D mountainous terrain by using heightmap's. Heightmap's are a greyscale power of two (e.g. 512 pixels in width and 512 pixels in height) image. The colour values in a heightmap are used to determine the height of the terrain. Completely black areas will render flat terrain, whereas bright white areas will render mountains. The colour values in-between are interpolated, giving a smooth and realistic looking terrain mesh. Heightmap's work extremely well if you want a quick, fast terrain without writing an algorithm. I discovered that PNG's[29] work best (BMP's are far too large) - no compression is lost and you get the added bonus of having an alpha channel, should you want the terrain to have transparency (great for levels that are floating mid-hair).



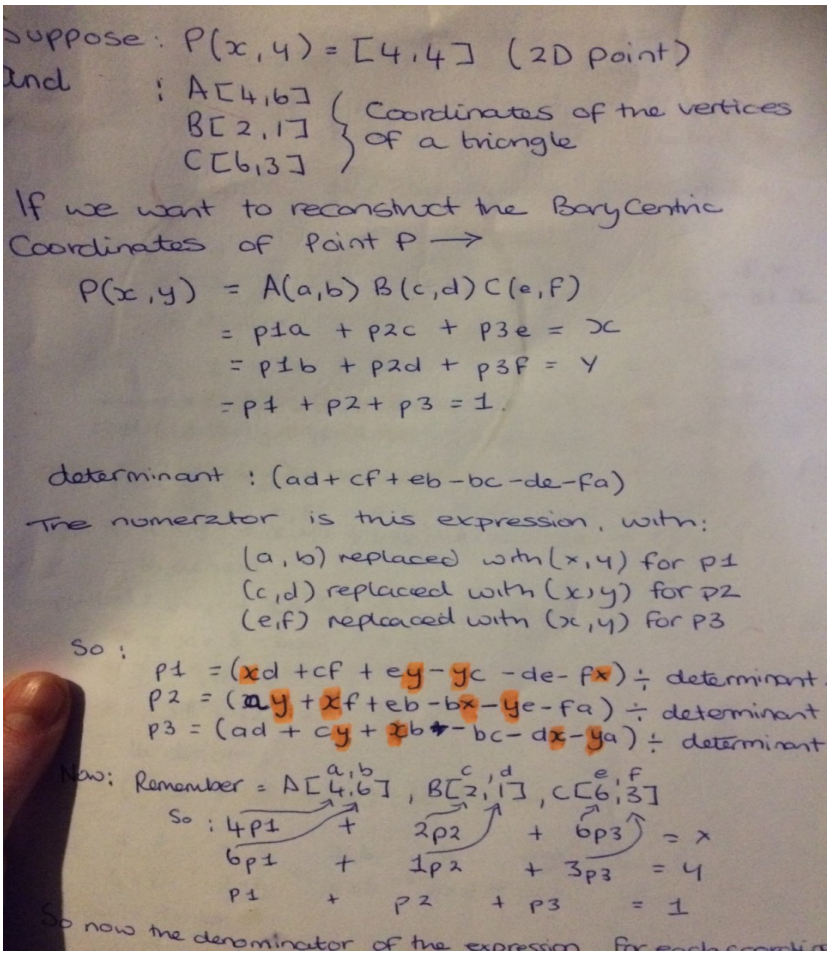
The image above shows a terrain being rendered using a heightmap. The heightmap (left) was modified and altered to fit the game I was making at the time. You may already see a small pathway beginning to take shape in the terrain image (right).

Procedural Terrain Generation

I have found many algorithms online on how to procedurally generate terrain and so I am comfortable this will be a fairly simple task. My favourite algorithm and the one I have decided to use as a basis, is that by games developer ThinMatrix[30]. The only time procedural terrain, in my opinion, becomes more complicated than heightmap's is when you want specific randomisation to occur. For example, if you have a lake in your game, you must make sure you have an algorithm in place so that all lake tiles are placed together, and then maybe interpolate smoothly into the grass tiles. For my terrain software however, I am only interested in generating random terrain.

Barycentric Coordinates

I spent a vast amount of time researching the best way to handle terrain collision. As most open-world terrains are hilly and unpredictable, I need to check every frame where the player has moved to on the terrain and update the players height (i.e. where they are standing) accordingly, so they don't fall through the terrain. A great, fast and easy way of doing this is by using a formula known as barycentric[19][31]. I wrote the formula out by hand, so I could fully understand how it works. In simplistic terms, the formula allows me to find an exact point on a triangle. In terms of terrain, the point on this triangle (within the terrain mesh) would then be the player's new height and this would be updated every frame.



The image above shows my research into the barycentric formula. I wrote it out by hand based on the information found on 2000clicks.com[32]. This was my way of fully understanding one method of terrain collision.

Development

Serialization

I needed serialization[33] to be present in my engine to enable me to load and save user-created terrain files. The added advantage of implementing serialization, is that I can use it for everything in my engine. The benefit of this is that a vast amount of work is done externally, without changing any code, compile times are much quicker and the risk of error is minimal.

```
    @see
        Shutdown()
    ************************************************************************************************************
    bool SDL::Initialize(const std::string& windowAttributes)
    {
        if (SDL_Init(SDL_INIT_EVERYTHING) == -1) { PRETTY_ERROR("{ SDL } SDL failed to initialize"); retu

defined(PRETTY_DEBUG)
        if (!SetAttribute(SDL_GL_CONTEXT_FLAGS, SDL_GL_CONTEXT_FORWARD_COMPATIBLE_FLAG | SDL_GL_CONTEXT_D
            PRETTY_WARNING("{ SDL } Debugging context flags not supported");
        }
if
        if (!Pretty::File::Instance()->Load(windowAttributes, m_Attributes)) { return false; }

        /* Define our attributes before creating the SDL window */
        if (!SetColorBufferBits()
            || !SetDepthBufferBits()
            || !SetStencilBufferBits()
            || !SetMultisampling()
            || !SetContextVersion()
            || !SetContext()
            || !SetDoubleBuffering()) { return false; }

        if (!LaunchWindow())    { return false; }
        if (!LaunchContext())   { return false; }

        EvaluateVerticalSync();
        EvaluateDisplayModes();

        PRETTY_SUCCESS("{ SDL } SDL initialized successfully");

        return true;
```

Image above shows my file management system loading the attributes for the SDL window. It is one line of code for pure simplicity.

I have implemented a strong file management system that uses the light-weight C++ serialization library known as Cereal[17].

My game engine now supports fully working serialization. I am able to use JSON[34], XML[35] and binary[36] files to load and save data. This will be extremely useful for me when I come to write my own terrain parser.

```
class SDL {

private:
    /*! @brief { Serialized } A struct that stores the SDL window attributes. */
    struct srWindowAttributes {
        /*! @brief Allows cereal access to Serialize(Archive& archive). */
        friend class cereal::access;

        std::string title;  /*!< The title of the window. */
        int width;          /*!< The width of the window. */
        int height;         /*!< The height of the window. */
        bool fullscreen;    /*!< Toggles fullscreen on/off, default is off. */
        bool vSync;         /*!< Toggles vSync on/off, default is on. */
        int colorBits;      /*!< The number of bits to use per color channel, default is 8. */
        int depthBits;      /*!< The number of bits to use for the depth buffer, default is 24. */
        int stencilBits;    /*!< The number of bits to use for the stencil buffer, default is 8. */
        int multisamples;   /*!< The number of multisamples to use, default is 8. */
        int version;        /*!< The default OpenGL version to use, default is 4. */
        int subVersion;     /*!< The default OpenGL sub version to use, default is 0. */
        bool coreMode;      /*!< Toggles core mode on/off, default is on. */
        bool doubleBuffer;  /*!< Toggles double buffering on/off, default is on. */

    private:
        /*! @brief Function that contains serialization data for loading and saving. */
        template <class Archive>
        void Serialize(Archive& archive)
        {
            archive(
                PRETTY_NVP(title),
```

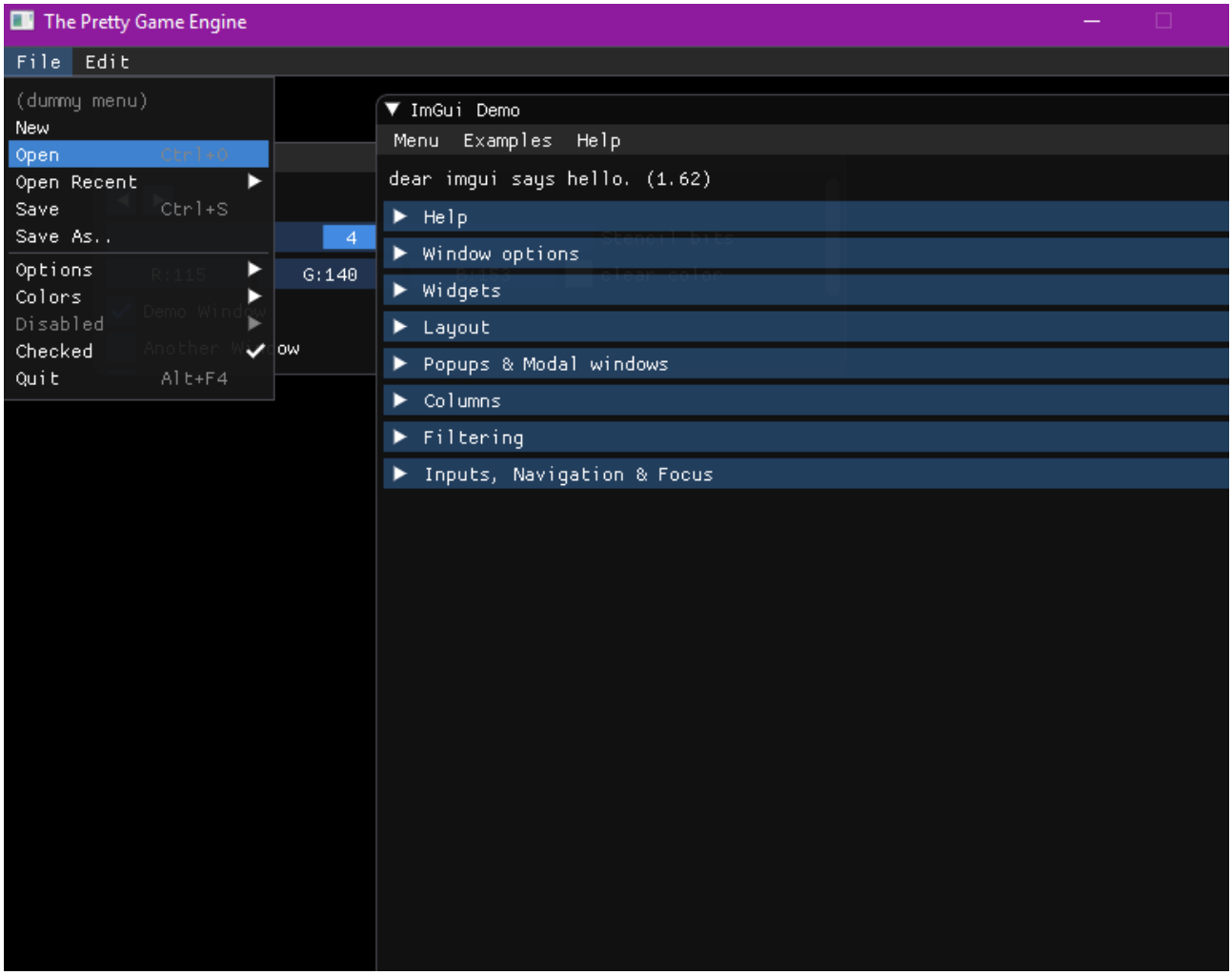
I managed to find a nice, neat way to save and load the data by storing it inside of a C++ struct. The struct is pre-fixed with "sr", so the user knows it can be serialized.

```
{
    "struct Pretty::Interface::SDL::srWindowAttributes": {
        "title": "The Pretty Game Engine",
        "width": 800,
        "height": 600,
        "fullscreen": false,
        "vSync": false,
        "colorBits": 8,
        "depthBits": 24,
        "stencilBits": 4,
        "multisamples": 8,
        "version": 4,
        "subVersion": 0,
        "coreMode": true,
        "doubleBuffer": true
    }
}
```

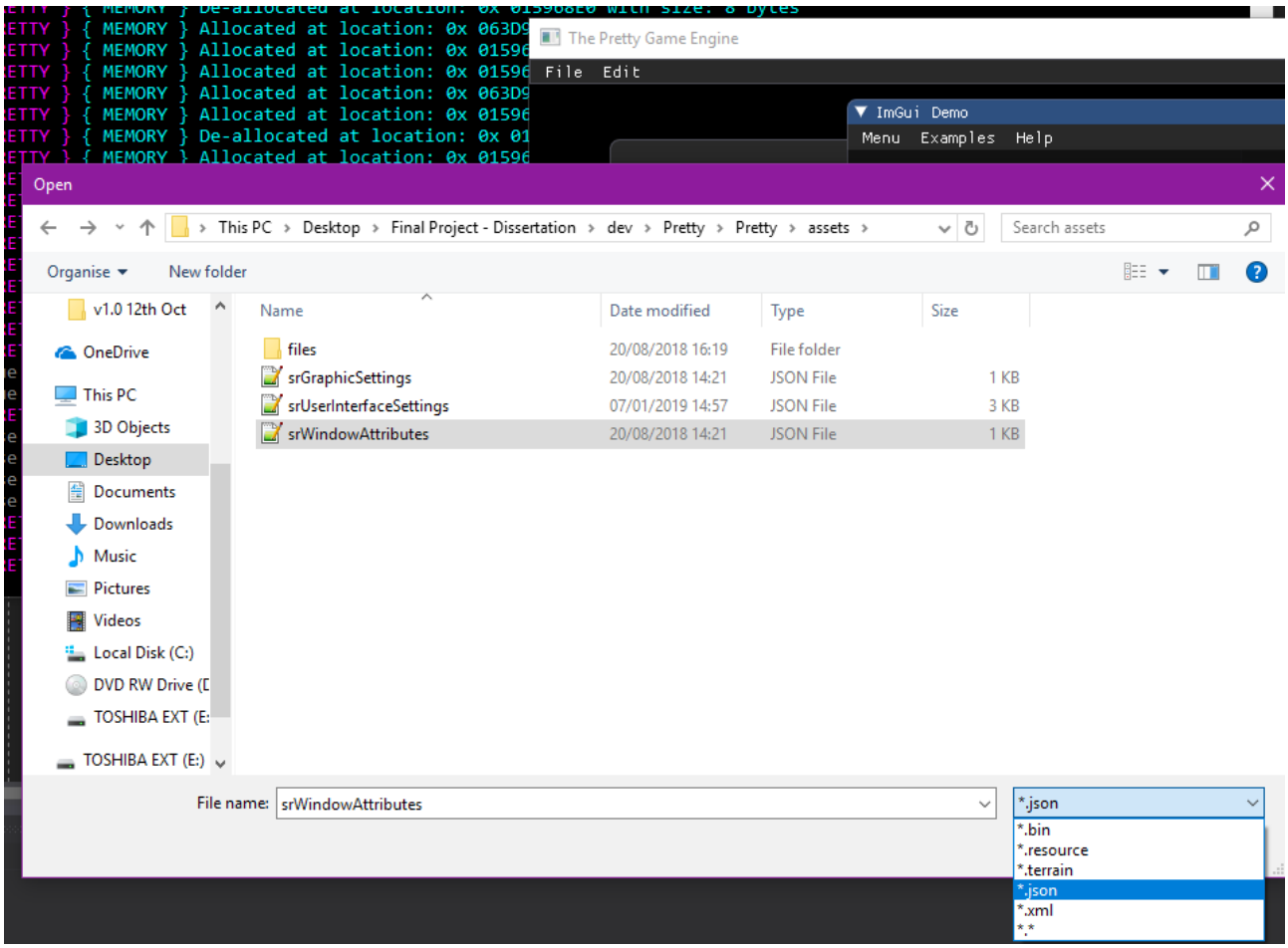
A JSON file storing the SDL window attributes. Data-driven design[37] is now active throughout my entire engine, no data is contained in my code.

File Dialogs

I implemented open and save file dialogs using ImGui[38] and the cross-platform Native File Dialog (NFD)[39] library. It is important that I allow the user to load and save terrain files from the program. I will also need file dialogs to load and save heightmap's.



The image above shows the drop-down menu created using ImGui, which will allow the user to open and save files.



The image above shows the file dialog working, with all necessary extensions in place. The user also has the option to add their own extension(s) to the extension list in the bottom right corner.

Memory Management

I need my game engine to handle memory efficiently and want to have control over allocations and de-allocations. This is so I can keep track of the amount of memory the terrain is using and pre-allocate memory to further speed up the program. I also wanted all memory to have 16-byte alignment[40] by default. This will make data transfers between the central processing unit (CPU) and and graphics card (GPU) in sync. I can also keep track of my memory now and output it to the screen, making sure I have no memory leaks.

```
    @return
        Allocates memory on the heap (done internally).
    ***********************************************************************************************************************/
    void* Allocator::_Allocate(size_t size)
    {
        PRETTY_ASSERT(size < GiB);

        /* Store extra size so we can store the size of this memory allocation before the memory block */
        size_t totalSize    = size + sizeof(size_t);
        uint8_t* result     = (uint8_t*)_aligned_malloc(totalSize, PRETTY_MEMORY_ALIGNMENT);

        /* Set the memory to null */
        memset(result, 0, totalSize);

        /* Store the size of this memory block in the memory allocated so we can grab it later */
        memcpy(result, &size, sizeof(size_t));

    #if defined(PRETTY_DEBUG)
        if (size > MiB) { PRETTY_WARNING("Large allocation (> 1MiB) at location: 0x", (void*)result, "with size:", size, "bytes"); }
        else            { PRETTY_MEMORY("Allocated at location: 0x", (void*)result, "with size:", size, "bytes"); }

        s_Statistics.totalAllocated += size;
        s_Statistics.currentUsed += size;
        s_Statistics.allocationCount++;
    #endif

        /* Now we have stored the size, move result ptr to start of memory block */
        result += sizeof(size_t);

        return result;
    }
```

The image above shows my own memory allocator. I store the size of the memory being allocated also, so I can keep track of my memory allocations and de-allocations and the overall memory my program is using.

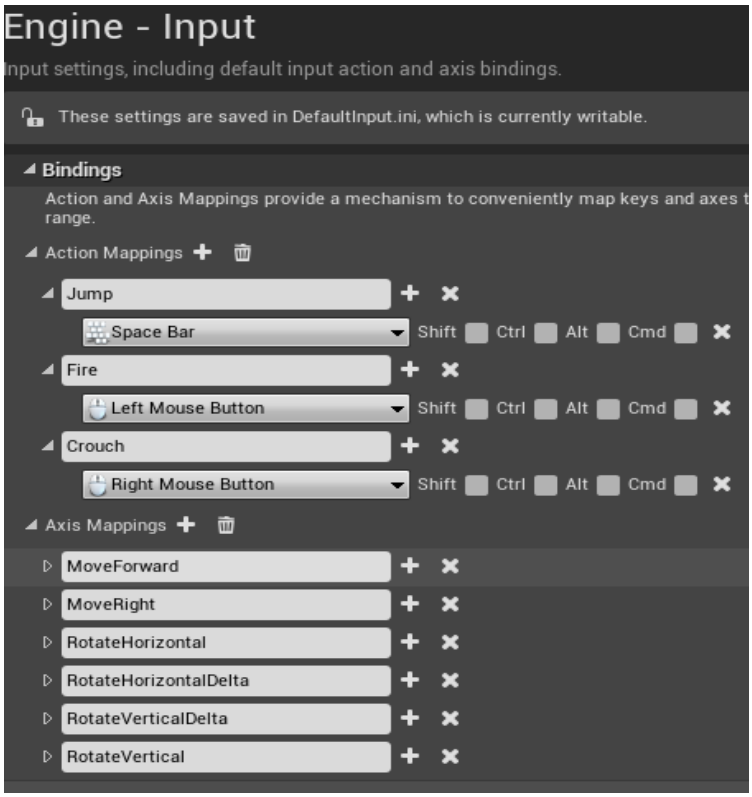
Input Handler

Due to using a user interface (ImGui[38]) that creates a multitude of windows, I needed to have a robust input handler in place that handles different input contexts, depending on what window the user is in. For example, in games such as Grand Theft Auto, different input contexts are used throughout. The player has different input options when walking, as opposed to when they're driving a car. I don't want the user to scroll down using the mouse in an ImGui window, whilst also zooming in to the terrain. To achieve versatility with user input, I implemented an ASR (Action, State, Range) input handler[41]. This works in a similar way to the Unreal Game Engine's[42] input system.

ContextList.json | srMainContext.json | srWindowAttributes.json | srGraphicSettings.json

```
{
    "Value0": {
        "actions": [
            {
                "key": 30,
                "value": {
                    "name": "THIS IS A BUTTON"
                }
            }
        ],
        "states": [
            {
                "key": 31,
                "value": {
                    "name": "THIS IS A STATE"
                }
            }
        ],
        "ranges": [
            {
                "key": 39,
                "value": {
                    "name": "THIS IS A RANGE",
                    "sensitivity": 50.0,
                    "input.min": -1000.0,
                    "input.max": 1000.0,
                    "output.min": -1.0,
                    "output.max": 1.0
                }
            }
        ]
    }
```

The image above shows an input context I have created.



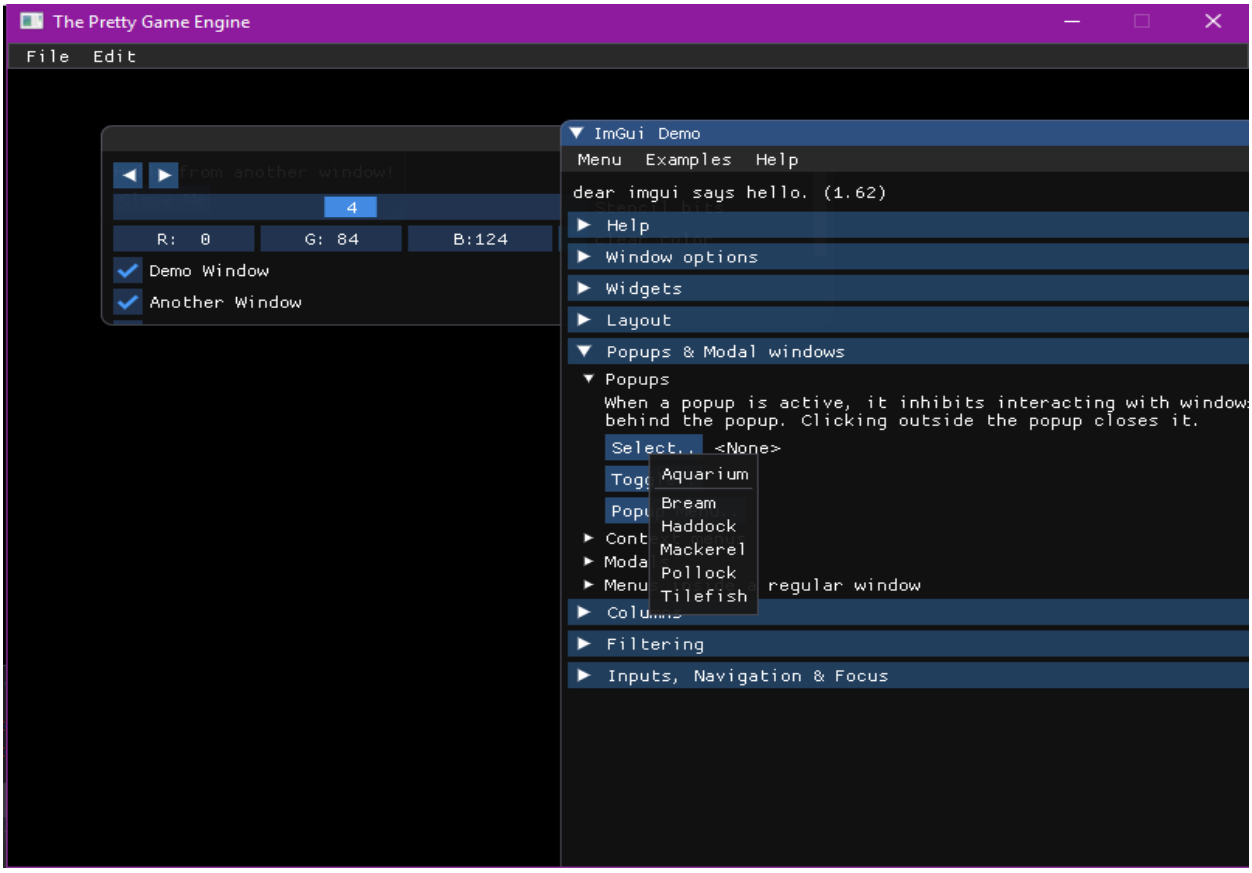
I follow the same design as the Unreal Engine.

The ASR input handler works in two ways. First, we have the input contexts, which store all conversion information between raw input keys and user-defined keys. Secondly, we have an input mapper, which is responsible for controlling what happens when a specific key is triggered.

User Interface

I wanted a simple user interface for the user. I decided to use the already existing ImGui user interface, as building one from scratch would have taken a long time.

ImGui allows me to add buttons, sliders, text fields, drop-down menus, tick boxes and much, much more. This is extremely useful to me, as the user interface will be the most important aspect of my terrain generation tool. I want the user to have a stress-free experience, without any lag or crashes, as well as having a simplistic and clean looking UI to work with.



The image above shows my engine currently. The UI is working well, however I still need to modify it to my liking.

Frustum Culling

I am beginning to implement optimization techniques now, in preparation for rendering the terrain. Frustum Culling[10] will work extremely well with all of the other terrain optimization techniques I have mention in this report.

```
/*************************************************************************************************************************
    A function which checks if a single point is inside the frustum
*************************************************************************************************************************/
bool Frustum::IsPointInside(const glm::vec3& position) {

    //--- Go through all the sides of the frustum
    for (unsigned int side = 0; side < s_maxSides; side++) {
        //--- Calculate the plane equation and check if the point is behind a side of the frustum
        if (m_frustum[side][A] * position.x +
            m_frustum[side][B] * position.y +
            m_frustum[side][C] * position.z +
            m_frustum[side][D] <= 0.0f) {
            //--- The point was behind a side, so it ISN'T in the frustum
            return false;
        }
    }

    //--- The point was inside of the frustum (In front of ALL the sides of the frustum)
    return true;
}


/*************************************************************************************************************************
    A function which checks if a sphere is inside the frustum, giving its center position and radius
*************************************************************************************************************************/
bool Frustum::IsSphereInside(const glm::vec3& centerPosition, float radius) {

    //--- Go through all the sides of the frustum
    for (unsigned int side = 0; side < s_maxSides; side++) {
        //--- If the center of the sphere is farther away from the plane than the radius
        if (m_frustum[side][A] * centerPosition.x +
            m_frustum[side][B] * centerPosition.y +
            m_frustum[side][C] * centerPosition.z +
            m_frustum[side][D] <= -radius) {
            //--- The distance was greater than the radius so the sphere is outside of the frustum
            return false;
        }
    }
```

The image above shows completed source code of my Frustum Culling class. Unit tests have already been completed and it speeds up rendering and game updates massively.

Source Code and Documentation

To allow for fast documentation to be generated, I am using a software called Doxygen[43]. Doxygen generates full documentation automatically, based on the comments written in your code. I am commenting all of my code as I develop my project and will provide full documentation and source code once fully complete.

As this project will eventually be showcased on my professional portfolio, I am determined to have it fully documented. This will also help future game developers re-create and possibly further expand what I have developed.

PrettyBinary

Main Page | Namespaces | Classes | Files | External Links

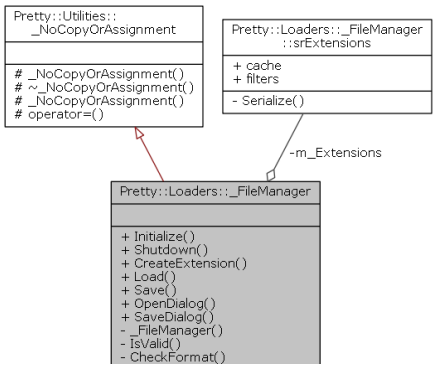
Pretty › Loaders › _FileManager

Classes | Public Member Functions | Private Member Functions | Pri

Pretty::Loaders::_FileManager Class Reference

A singleton class that allows for file loading and saving. **More...**

Collaboration diagram for Pretty::Loaders::_FileManager:



The image above is a screenshot of my current documentation. This image shows my File Manager class information.

A singleton class that allows for file loading and saving.

Warning
NO DIRECT ACCESS

- This class should not be accessed directly. Call **Pretty::File::Instance()** for access.

Encapsulates all cereal and std::fstream procedures and methods, allowing ease of use. When used in conjunction with the **Pretty** Engine, full error checking is provided within the class. Ad

To serialize data, the templated function Serialize must be declared within a class/struct as follows:

```
template <class Archive>
void Serialize(Archive& archive)
{
    archive(PRETTY_NVP(variable));
}
```

You can pass any number of arguments into the archive. The **PRETTY_NVP(T)** macro saves the variable to the file with the string representation of the variable name as its tag.

Construction and Destruction

Call function:

```
Initialize(const std::string& extensionsFilePath)
```

passing in a string to the file path where the list of supported extensions are stored.

Warning
Remember to clean up! Call function:

```
Shutdown(const std::string& extensionsFilePath)
```

before closing the program. Not doing so will mean that any extensions you have created during run-time will not be saved.

Loading and Saving

To save data call:

```
Save(const std::string& filePath, Args&&... args)
```

The image above shows further documentation on my File Manager class. I explain how to initialize and shutdown the object correctly and how to load and save files using my File Manager.

Challenges

Thus far into development my challenges have been quite minimal and I put this down to the level of research I done last year – I had already faced most challenges prior to commencing development of this software.

One of the biggest challenges I have faced and continue to face is optimization. Usually, I build a game without optimization in mind as previously I have made games that are fairly small. However, as you start uploading larger models, especially terrain, combined with extensive graphic effects (for example, normal maps), optimization does become a massive issue.

Thus far I have managed to keep my software extremely fast and still obtain a great frame rate (upwards of 3000 frames per second). In the past I have tackled the problem of optimization with frustum culling[10] only. I am hoping that the optimization techniques I have discovered and covered in this report will further increase render and update times. I am constantly doing unit tests and keep track of my frame rate.

Creating dynamic data in programming is fairly difficult. This becomes a pain when you want to give the user as much control as possible. I had the issue where I wanted the user to be able to have full control over file management. They would be able to create custom extensions, which they could then link to their own custom parser. All their extensions would be saved to a file and be re-usable. This would work nicely with the file dialog I have in place. Unfortunately, after two months of research in this area, I had to give up as it was far too time-consuming to carry on. I did discover a design pattern that would have made this possible, known as the Policy Based design pattern[44], but in the end I decided it was not worth the hassle and not entirely necessary. I have since implemented something simpler. The user can create their own extension names, but must use the parsers I have provided only.

I am using a user interface (UI) library known as ImGui[38]. It is a very light-weight library and looks quite polished. However, there is absolutely no

documentation for it online and so I face a constant struggle trying to figure out how it works. To tackle this I have started experimenting with all its features and keep notes once I discover what something does. This has made me realise even more how important it is to have full, completed documentation for my project.

Time Management

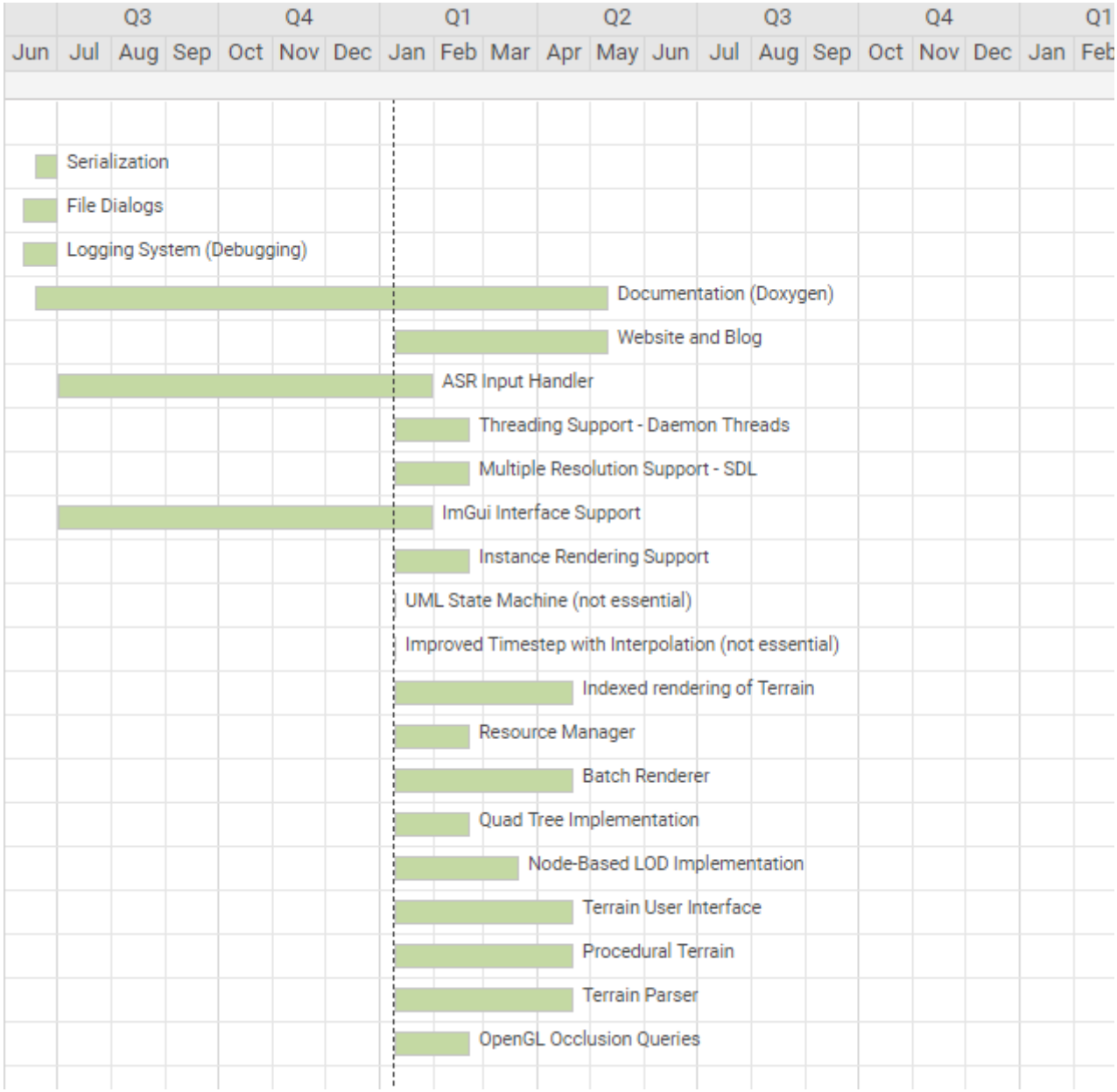
I am constantly tracking my progress by keeping development logs, diaries and bug reports. I have a list of key dates where I must complete certain tasks by. I find this is the best way to meet personal deadlines and it keeps me focused.

In all honesty, I generally do not use Gantt charts, and have created one primarily for this report. This chart outlines my actual development timeline, which is stuck to my massive notice-board above my desk.

If I ever fall behind with my work, I prioritise the most important tasks first. If I spend too much time on a specific task, I will leave it and come back to it later. As long as I complete the core tasks, the additional features can follow at a later date as they are not as important. I aim to have a finished project, even if it is slightly basic. I can always further develop it at a later date.

Remedial Plan

Engine Specific	Task Name	Added	Due Date	Done	Assi... To	Status	Comments
	The Pretty Project Development Timeline			☐			
●	~~Serialization~~	~~06/18/18~~	~~06/30/18~~	☑	~~Kim~~	~~Complete~~	
●	~~File Dialogs~~	~~06/11/18~~	~~06/30/18~~	☑	~~Kim~~	~~Complete~~	~~Threading still to be implemented~~
●	~~Logging System (Debugging)~~	~~06/11/18~~	~~06/30/18~~	☑	~~Kim~~	~~Complete~~	
	Documentation (Doxygen)	06/18/18	05/10/19	☐	Kim	In Progress	
	Website and Blog	01/09/19	05/10/19	☐	Kim	Not Started	Notes, comments and articles/tutorials written but website not online
●	ASR Input Handler	07/01/18	01/30/19	☐	Kim	In Progress	Just to clean up code and do documentation/commenting
●	Threading Support - Daemon Threads	01/09/19	02/20/19	☐	Kim	In Progress	Implement thread pool with reference counting (or use a library)
●	Multiple Resolution Support - SDL	01/09/19	02/20/19	☐	Kim	In Progress	Need buttons and input to test
●	ImGui Interface Support	07/01/18	01/30/19	☐	Kim	In Progress	ImGui set up and working, need to clean up code
●	Instance Rendering Support	01/09/19	02/20/19	☐	Kim	In Progress	
	UML State Machine (not essential)	01/09/19		☐		Not Started	
	Improved Timestep with Interpolation (not esse	01/09/19		☐		Not Started	
	Indexed rendering of Terrain	01/09/19	04/20/19	☐	Kim	Not Started	Will speed up rendering
●	Resource Manager	01/09/19	02/20/19	☐	Kim	In Progress	For keeping track of all GPU data
●	Batch Renderer	01/09/19	04/20/19	☐	Kim	Not Started	For fast texture rendering
	Quad Tree Implementation	01/09/19	02/20/19	☐	Kim	Not Started	For terrain optimization
	Node-Based LOD Implementation	01/09/19	03/20/19	☐	Kim	Not Started	For terrain optimization
	Terrain User Interface	01/09/19	04/20/19	☐	Kim	Not Started	
	Procedural Terrain	01/09/19	04/20/19	☐	Kim	Not Started	
	Terrain Parser	01/09/19	04/20/19	☐	Kim	Not Started	
●	OpenGL Occlusion Queries	01/09/19	02/20/19	☐	Kim	In Progress	For terrain optimization



References

1: Wikipedia, Procedural Generation, 2018, https://en.wikipedia.org/wiki/Procedural_generation

2: Wikipedia, Heightmap, 2018, https://en.wikipedia.org/wiki/Heightmap

3: BiteTheBytes, World Creator, 2019, https://www.world-creator.com

4: SDL, Simple DirectMedia Layer, 2018, https://www.libsdl.org/

5: Wikipedia, Open Graphics Library (OpenGL), 2018, https://en.wikipedia.org/wiki/OpenGL

6: Assimp, The Open-Asset-Importer-Lib, 2018, http://www.assimp.org

7: Wikipedia, Vertex (geometry), 2018, https://en.wikipedia.org/wiki/Vertex_(geometry)

8: Wikipedia, Quadtree, 2019, https://en.wikipedia.org/wiki/Quadtree

9: RasterTek, Tutorial 5: Quad-tree's, 2016, http://www.rastertek.com/tertut05.html

10: Rodriguez, Jorge, Math for Game Developers - Frustum Culling, 2013, https://www.youtube.com/watch?v=4p-E_31XOPM

11: ThinMatrix, OpenGL Tutorial 54: Occlusion Queries, 2017, https://www.youtube.com/watch?v=LMpw7foANNA

12: RasterTek, Tutorial 18: Large Terrain Rendering, 2016, http://www.rastertek.com/terdx10tut18.html

13: Erkenbrach, Leif, Planetary Scale LOD Terrain Generation, 2014, http://leifnode.com/2014/04/planetary-scale-lod-terrain-generation

14: Hong, K, Multi-threaded Programming with C++ - Part A, 2015, https://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php

15: Autodesk, Maya, 2019, https://www.autodesk.co.uk/products/maya/overview

16: ThinMatrix, OpenGL 3D Game Tutorial 29: Mouse Picking, 2015, https://www.youtube.com/watch?v=DLKN0jExRIM

17: Grant, W. Shane and Voorhies, Randolph, Cereal, 2017, https://uscilab.github.io/cereal/

18: Wikipedia, Software Design Pattern, 2019, https://en.wikipedia.org/wiki/Software_design_pattern

19: Wikipedia, Barycentric Coordinate System, 2018, https://en.wikipedia.org/wiki/Barycentric_coordinate_system

20: Wikipedia, Normal (geometry), 2018, https://en.wikipedia.org/wiki/Normal_(geometry)

21: Wikipedia, UV Mapping, 2018, https://en.wikipedia.org/wiki/UV_mapping

22: Red Eclipse, Blendmap, 2016, https://redeclipse.net/wiki/Blendmap

23: OpenGL-Tutorial, Tutorial 13: Normal Mapping, 2017, http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping

24: Wikipedia, Finite Difference Method, 2018, https://en.wikipedia.org/wiki/Finite_difference_method

25: Chernikov, Yan, Ep.9: Ultra-Fast Batch Renderer, 2015, https://www.youtube.com/watch?reload=9&v=ImtWD_9OAeY

26: Wikipedia, Shader, 2018, https://en.wikipedia.org/wiki/Shader

27: Wikipedia, Reference Counting, 2018, https://en.wikipedia.org/wiki/Reference_counting

28: Wikipedia, Smart Pointer, 2018, https://en.wikipedia.org/wiki/Smart_pointer#shared_ptr_and_weak_ptr

29: Wikipedia, Portable Network Graphics, 2019, https://en.wikipedia.org/wiki/Portable_Network_Graphics

30: ThinMatrix, OpenGL 3D Game Tutorial 37: Procedural Terrain, 2016, https://www.youtube.com/watch?v=qChQrNWU9Xw

31: ThinMatrix, OpenGL 3D Game Tutorial 22: Terrain Collision Detection, 2014, https://www.youtube.com/watch?v=6E2zjfzMs7c

32: McRae, Graeme, Barycentric Coordinates, Areal Coordinates, 2012, http://2000clicks.com/MathHelp/GeometryTriangleBarycentricCoordinates.aspx

33: Wikipedia, Serialization, 2018, https://en.wikipedia.org/wiki/Serialization

34: Wikipedia, JSON, 2019, https://en.wikipedia.org/wiki/JSON

35: Wikipedia, XML, 2019, https://en.wikipedia.org/wiki/XML

36: Wikipedia, Binary Number, 2018, https://en.wikipedia.org/wiki/Binary_number

37: Wikipedia, Data-Driven Programming, 2018, https://en.wikipedia.org/wiki/Data-driven_programming

38: Cornut, Omar, ImGui, 2016, http://www.miracleworld.net

39: Labbe, Michael, Native File Dialog, 2017, https://github.com/mlabbe/nativefiledialog

40: Wikipedia, Data Structure Alignment, 2018, https://en.wikipedia.org/wiki/Data_structure_alignment

41: Lewis, Mike, Designing a Robust Input Handling System for Games, 2013, https://www.gamedev.net/articles/programming/general-and-gameplay-programming/designing-a-robust-input-handling-system-for-games-r2975

42: Epic Games, Unreal Engine, 2019, https://www.unrealengine.com

43: Heesch, van Dimitri, Doxygen, 2018, http://www.doxygen.nl

44: Wikipedia, Modern C++ Design, 2018, https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design

Alexandrescu, Andrei, Modern C++ Design: Generic Programming and Design Patterns Applied, 2001, Addison Wesley

Chernikov, Yan, The Cherno Project Series, 2016, Self-published
www.youtube.com/TheChernoProject

De Vries, Joey, Learn OpenGL, 2015, Self-published
http://learnopengl.com

Mitchell, Shaun, SDL Game Development, 2013, Packt Publishing Limited

RasterTek, DirectX 11 Terrain Tutorials, 2016, Self-published
http://www.rastertek.com