CE301 Final Report

2.5D Tile based game with Procedurally Generated Content and engine

Student: Michael Allport

Registration Number: 1802882

Supervisors: Dr Michael Fairbank

Second Assessor: Dr Ghonbing Gu

Degree Course: Computer Science

Acknowledgements

Special thanks go to my supervisor, Dr Michael Fairbank, for giving continual guidance throughout the planning and feature development of this project. Without his support and encouragement through the difficulties, and that final push, this project would not be possible.

I would also like to give thanks to Dr Vishwanathan Mohan, Professor Anthony John Vickers, my second assessor Professor Dongbing Gu, and everyone at University of Essex who contributed to CE301 module. It was with their works that enabled production of this project.

Thanks also go to all sources acknowledged in this report's references section, and the creators of all libraries used.

Additionally, I would like to pay thanks to Pankaj Shah, Richard Farrar, and everyone who contributed to my teachings on my Access to Higher education diploma. It was with their guidance and encouragement that helped me develop the strong learning skills applied and extended throughout this module.

Final thanks go to my partner, Jurgita Armonaite, who has given me continual support throughout University and this project. Many nights were spent bouncing ideas around with her and asking her thoughts and input on all aspects of this project.

Abstract

Numerous games are built and developed in industry using a suite of game independent features enabling the creation of their product. This has given rise to the term ‘Game Engine’ in 1990’s, and the release of main-stream engines see use in the production of many AAA rated games. This project explores some of the common features of such suites resulting in the creation of an engine independent of game logic. A rendering engine, event handling subsystem, logger, window context, and GPU bound asset manager is implemented.

Using the utilities provided in the game engine, a simple game is developed. The Survival game genre is explored, with an emphasis on producing a virtual world utilizing Procedural Content Generation. Noise based methods, Diamond Square and Perlin Noise, are implemented and used to create terrain of the world. Poisson-Disc sampling is used and implemented to fill the world with content such as trees, shrubs, grass, twigs etc.

Other required game features are implemented to enable the player to interact with the world. Ray Plane Intersection is implemented to project the 2-dimensional mouse position into a 3-dimensional rendered world. Game objects ID tags are encoded into uniquely identified RGB colours and rendered upon a hidden screen enabling identification of objects under the mouse by their texture patterns.

Contents

List of symbols and abbreviations

Term	Description
OpenGL	The specification upon which API's are created to manipulate graphics. This enables the communication to the graphics card, sending data to memory buffers, associating GLSL shader programs to perform operations on the GPU, and numerous other applications.
API	Application Programming Interface
GLSL	OpenGL Shading Language. The programming language used in creating GPU shading programs.
GPU	Graphics Processing Unit (graphics card).
PCG	Procedural Content Generation. Draws upon algorithmic Methods to automatically create game content.
NDC	The three-dimensional coordinate system upon which all geometry exists when rendered to screen. Range of x, y, and z axis are in range -1 to 1.
Quad	A 2D or 3D geometry structure consisting of a flat plane square made with four vertices incorporating two triangles.
Sampling	The process of selecting a set of positions existing in n-dimensional space.
Noise	The generation methods of points produced for sampling purposes with stochastic characteristics.
UID	Unique Identifier

Chapter 1 – Preface

1.1 Context

In game development, there is a clean distinction between game design and logic elements and systems which enable the developer to achieve their goals [1, 34]. In the 1990's this separation became clear when the term Game Engine was introduced to describe a suite of these game-independent features. Doom [2], a first-person shooter game, was amongst the first to make that distinction, with subsystems such as three-dimension renderer, audio, and collision detection subsystems [1]. Valve's Source Engine was designed for the game Half-Life [3], which incorporated physics, character animation, AI subsystems etc. and was extended to usage in Half-Life 2, Dark Messiah of Might and Magic [4]. The necessity for this independent platform saw two mainstream game engines, Unity [29] and Unreal Engine [30], which have been used to develop many AAA game titles such as [31, 32, 33].

This product employs and adapts the designs featured in three open source game engine related products, Hazel Engine [5], Ethereal Engine [6], and rendering modules highlighted in [7]. Direct code is rarely used within implementation, rather using them as a basis and expanding upon with other external resources [1, 34, 9, 46].

The engine will be the basis enabling production of a basic game. Survival games see players engaged in a virtual world surrounded with challenges which can be overcome by collecting resources in order to survive. Don't Starve [35] solely focuses on a player gathering resources to build bases and survive the hostile environment through challenging weather systems. This game, and many other such games, employs Procedural Content Generation using algorithms to automatically create various types of game content [19, 37]. Noise functions such as Perlin noise [27] are used to create textures, terrain, landscape, or place game content [37]. Borderlands [33] uses PCG methods to generate a wide array or in-game items such as weapons, shields, armour etc. Minecraft [39], a voxel-based survival game, uses PCG methods to create an endless world where content is generated as the player progresses through the world [38]. Many of the methods used are stochastic giving the user a new experience each time played. Korn et al.,[37] found users of PCG generated content preferred "changing nature of the game environments" over content over statically made content. Furthermore, seeds can be used in the random number generators driving PCG methods such that worlds can be reproduced, or shared with other players [19, 38].

A basic game will be produced using the game engine to imitate the style found in Don't Starve [35]. PCG methods will be used to generate a world and fill it with content. All game design has been created from new building upon knowledge gained in CE218 – Game Programming module. The only external references made are listed in this report for PCG methods, Object detection, and ray plane intersection.

1.2 Sustainability

Energy consumption is a key consideration in the problem of tackling global climate change [40, 41]. In the modern world our energy needs are only increasing over time reaching "unprecedented territory" [42]. Many solutions are being explored to battle the depletion of fossil fuels and the overall affect consumption levels are having on the world [43]. In 2015 the Paris agreement was released, giving international awareness global-warming and our effect on climate change [44, 45].

This project aims to take the most effective approaches to its algorithms. A frame limit is set to limit the amount of clock cycles used towards the users monitor refresh rate, such that no unnecessary processing will be done to exceed refresh rates.

1.3 Legal, Ethical, and Intellectual property

This project makes use of several open source libraries to which a combined copy of all licenses are available here. The project itself, and any custom textures made, is also included within this document, licensed under the MIT License.

Chapter 2 – Key Technical achievements

2.1 Rendering

One of the core elements to numerous game engines [29, 50, 5, 6, 7] is a graphics subsystem, provided the developer the ability to graphically represent and render their game world to the player. Such graphics systems are built utilizing hardware API's [1] allowing for communication of data to the GPU. Common libraries exist such as DirectX [50], Metal [52], OpenGL [53], or the more modern Vulkan [51] for this purpose. All libraries offer similar subset of features, creating GPU bound memory buffers and shader programs in order to manipulate and render 2D and 3D geometry to screen [1 34]. Whilst some libraries are platform specific, DirectX [50] a Windows product compatible with Windows systems and Metal [52] an Apple product for use on Macs, the latter [53, 54] are portable multi-platform libraries. OpenGL has been chosen for the purpose of this Game Engine for its portability and depth of external documentation [5, 7, 8, 9, 46].

2.1.1 Rendering pipeline

Game engine includes a low-level graphics rendering pipeline, where 2-dimensional and 3-dimensional geometry is rendered on screen through use of OpenGL. The main usage of this engine is to provide rendering of quads in either colour or with a textured background from an external image with texture asse classes.

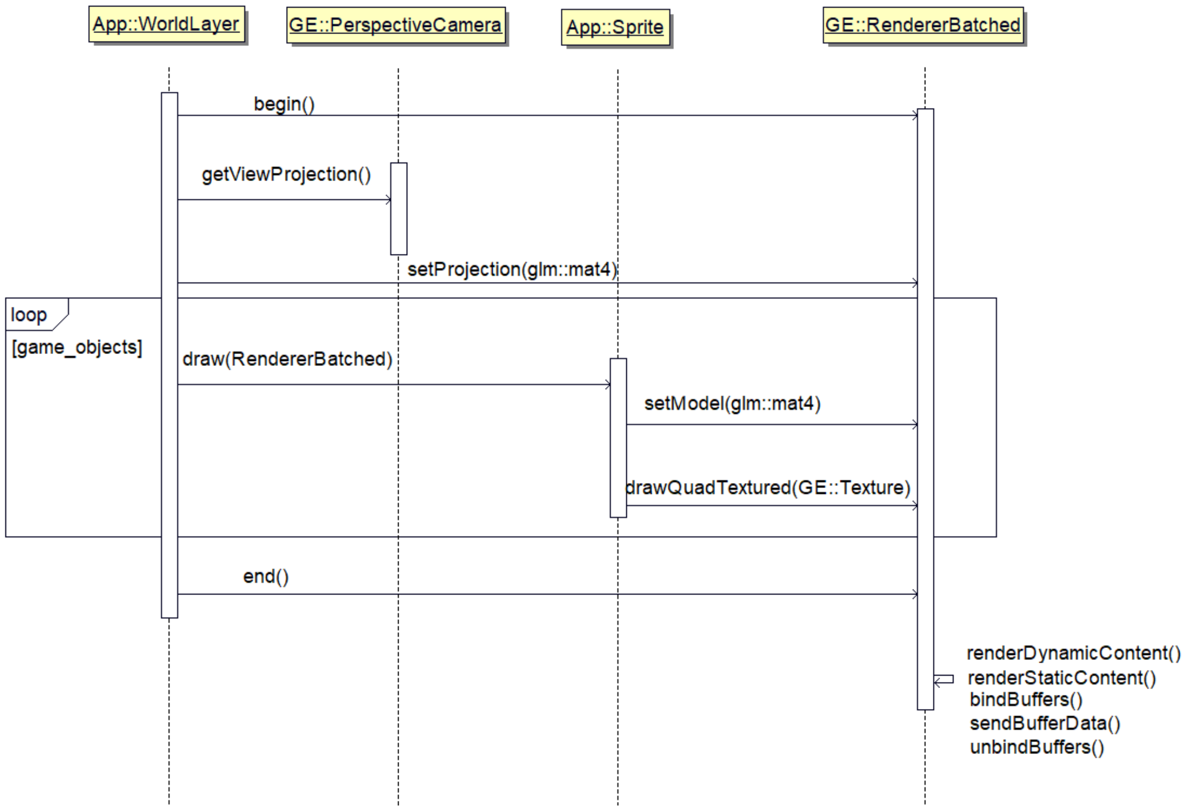
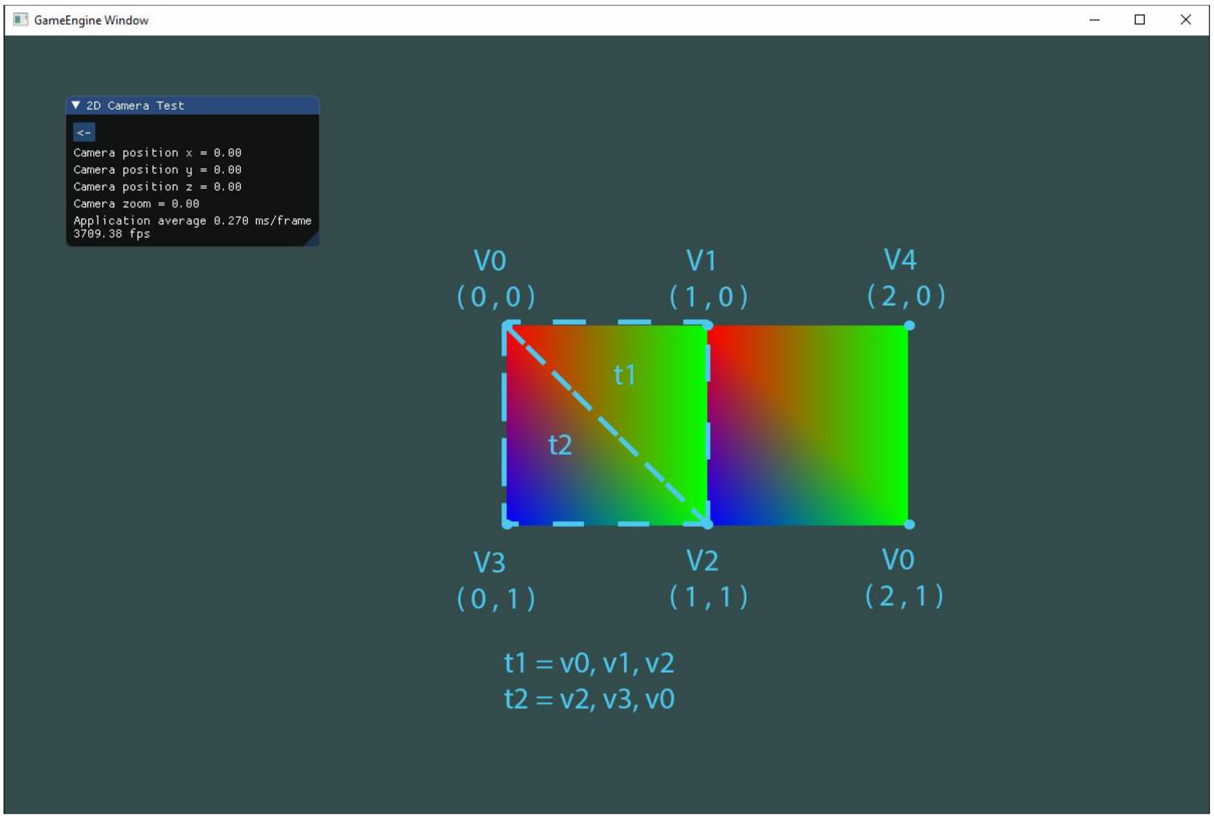


Fig1, Render cycle made to GE::RendererBatched

Fig1 shows a communication diagram of a render cycle executed from the App (game) to the renderer. All geometry information for the shapes to be rendered are cached in the renderer alongside what textures or colours are to be interpolated over for each pixel. At the end of each cycle, the OpenGL GPU memory buffers are bound, geometry information sent to the GPU, and a draw call made.

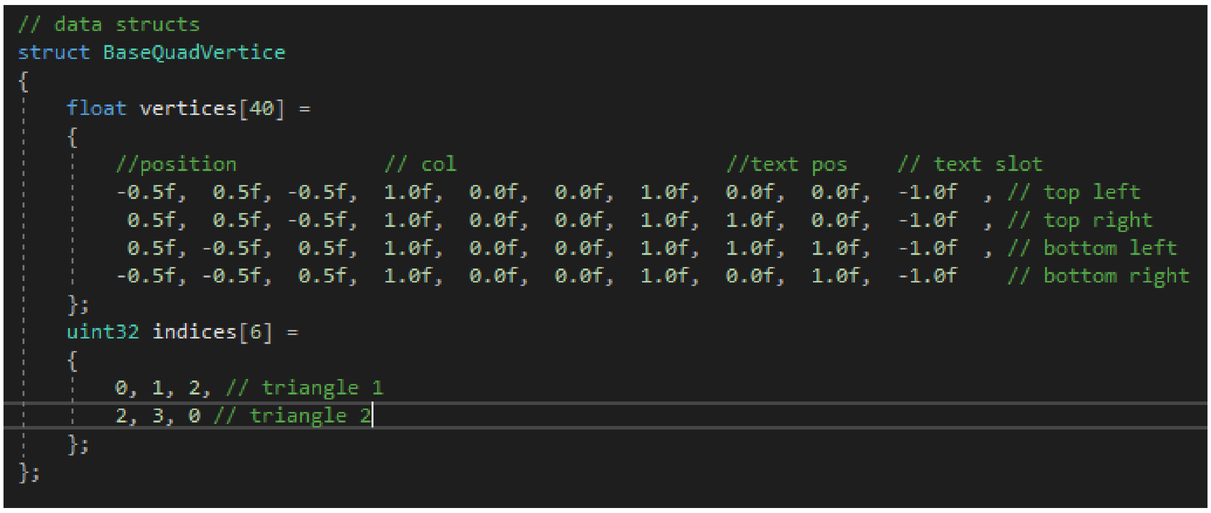
2.1.2 2D/3D Rendering geometry



```
Camera position x = 0.00
Camera position y = 0.00
Camera position z = 0.00
Camera zoom = 0.00
Application average 0.270 ms/frame
3709.38 fps
```

Fig 2 – Rendering of 2no 2D quads to screen with random blended colouring

Fig 2 shows the basic rendering of two quads to screen. For each vertex (corner) of the quad, data of its position, texture position, colour, and texture ID is sent to GPU memory. Each vertex is assigned an index identifying its point in memory, Fig3. A triangle may then be formed of indices {0, 1, 2}, and a square may be formed of two triangles with indices {0, 1, 2, 2, 3, 0}



```
// data structs
struct BaseQuadVertice
{
    float vertices[40] =
    {
        //position           // col                        //text pos     // text slot
        -0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,  0.0f,  -1.0f  , // top left
         0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  1.0f,  0.0f,  -1.0f  , // top right
         0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  1.0f,  1.0f,  -1.0f  , // bottom left
        -0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,  1.0f,  -1.0f    // bottom right
    };
    uint32 indices[6] =
    {
        0, 1, 2, // triangle 1
        2, 3, 0 // triangle 2
    };
};
```

Fig 3 – Base quad data structure with 10 floats pertaining to 1 vertex (corner) of a quad, and indices identifying which set of 10 floats make a triangle

The Renderer class maintains a base quad upon which all drawn quads are transformed from Fig2. This comprises a 1x1x1 unit sized quad centred about 0,0,0 axis as per Fig4. When a render call is made, the positions are transformed by a model matrix set by the user to either translate its position, transform scale, size, or perform any other matrix transformations as required, Fig5. This allows the rendering of quads of any dimension, colour, or texture in a vector coordinate system of the programmers choosing.

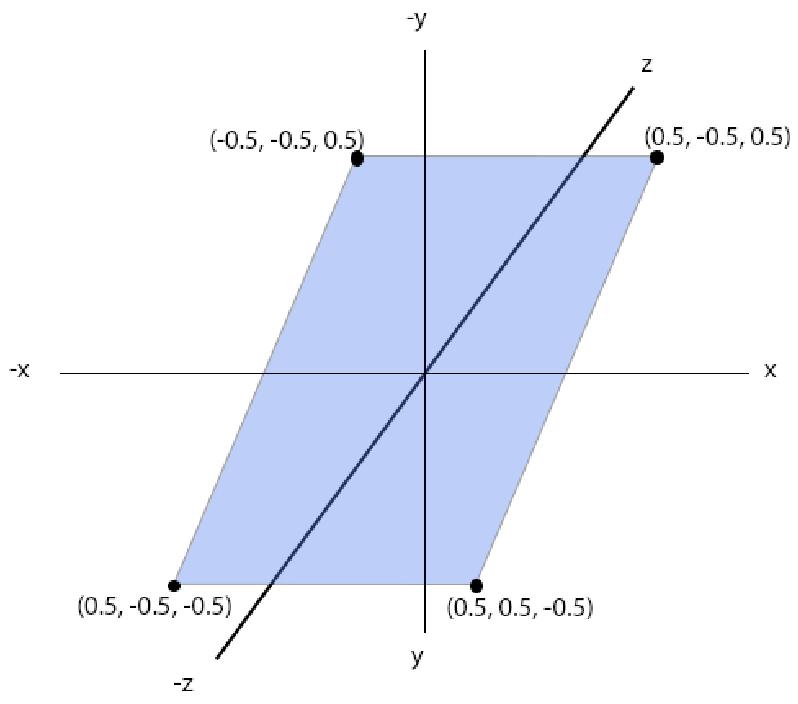
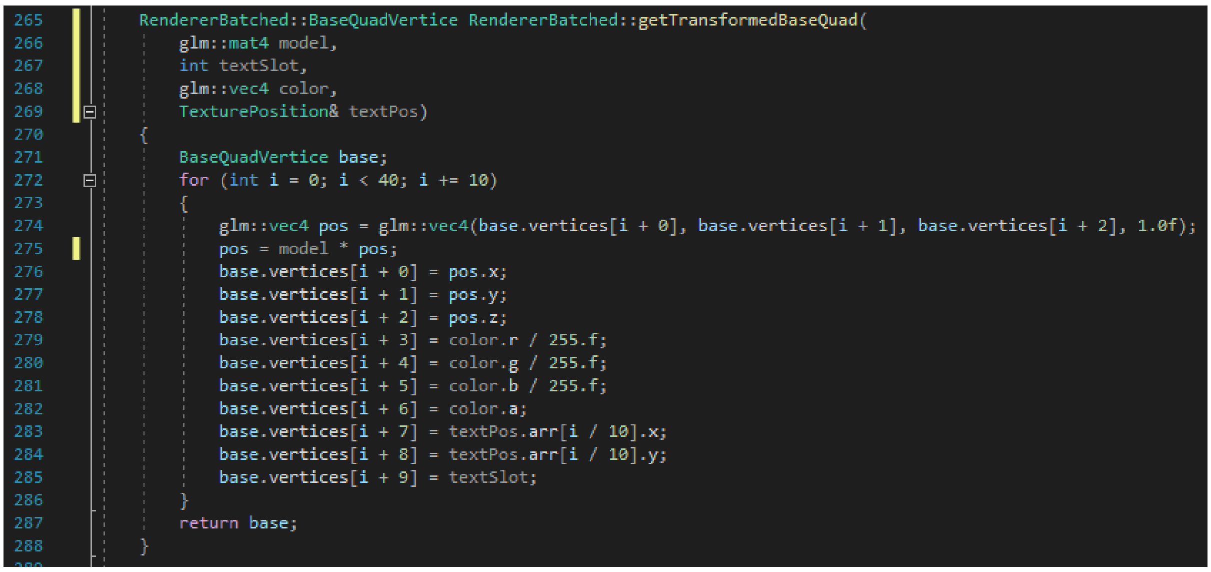


Fig4 – Base quad vertices in 3-dimensional space



```
265  RendererBatched::BaseQuadVertice RendererBatched::getTransformedBaseQuad(
266      glm::mat4 model,
267      int textSlot,
268      glm::vec4 color,
269      TexturePosition& textPos)
270  {
271      BaseQuadVertice base;
272      for (int i = 0; i < 40; i += 10)
273      {
274          glm::vec4 pos = glm::vec4(base.vertices[i + 0], base.vertices[i + 1], base.vertices[i + 2], 1.0f);
275          pos = model * pos;
276          base.vertices[i + 0] = pos.x;
277          base.vertices[i + 1] = pos.y;
278          base.vertices[i + 2] = pos.z;
279          base.vertices[i + 3] = color.r / 255.f;
280          base.vertices[i + 4] = color.g / 255.f;
281          base.vertices[i + 5] = color.b / 255.f;
282          base.vertices[i + 6] = color.a;
283          base.vertices[i + 7] = textPos.arr[i / 10].x;
284          base.vertices[i + 8] = textPos.arr[i / 10].y;
285          base.vertices[i + 9] = textSlot;
286      }
287      return base;
288  }
```

Fig5 – BaseQuadVertice initialized line 271, matrix transformation applied line275 where model is a 4x4 transformation matrix, and position updated lines 276-278

The geometry data undergoes a series of matrix transformations [46]. Initially the vertices exist in local space, centred about 0,0,0 Fig4. A model transformation is applied performing any scaling/rotating as required to represent the object locally, and to also to translate its origin to the worlds coordinate system Fig6-1. The view transformation is then applied translating objects relative to the cameras position, such that all vertices are centred about the cameras position, Fig6-2. A projection transformation is then applied converting vertices to OpenGL's NDC system, Fig6-4.

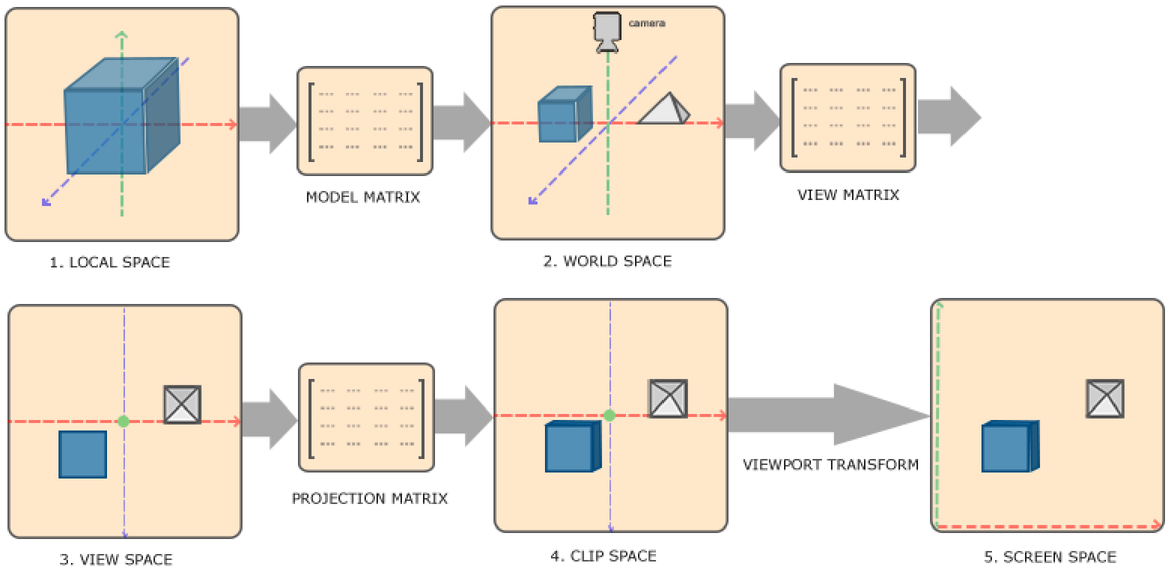
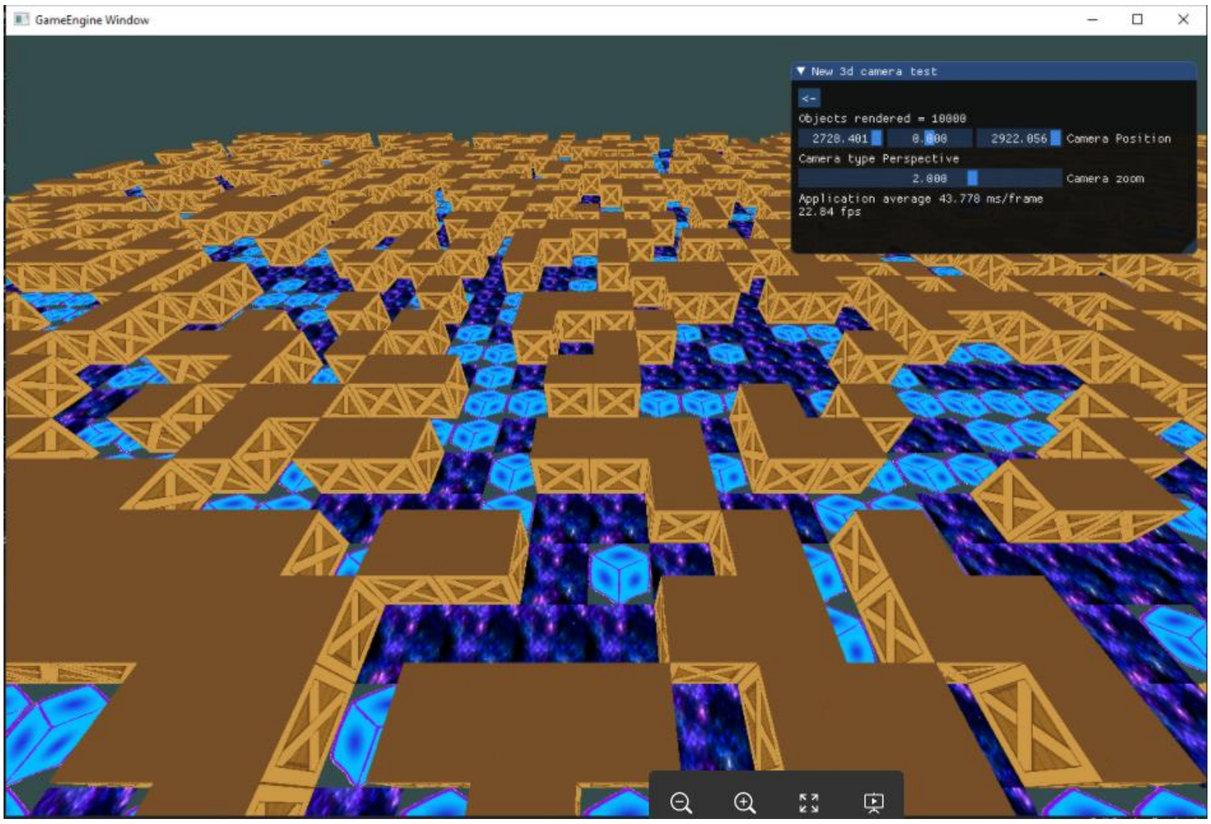


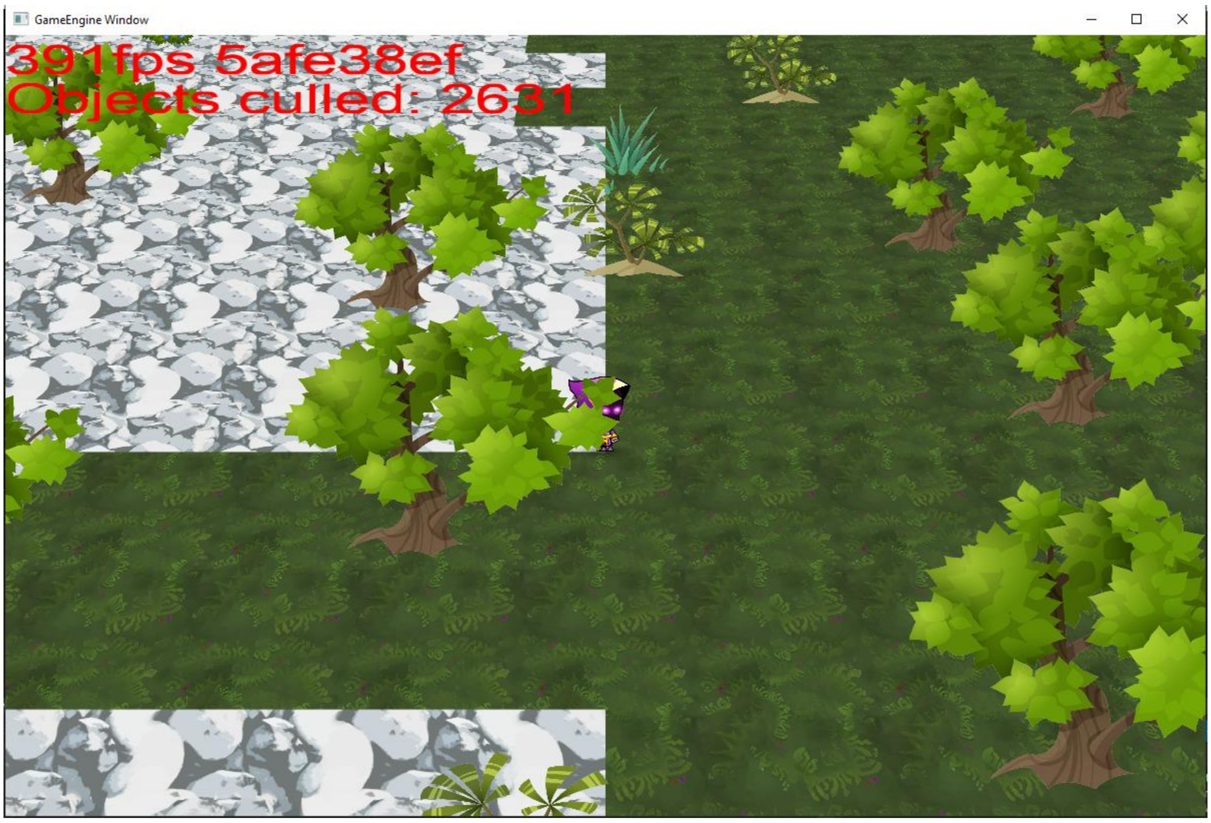
Fig6 – Coordinate system transformations [46]

Furthermore, use of perspective projection transformation allows for rendering of multiple quads in 3-dimensions with perspective, Fig7. This ultimately allowed for the floor within the game to be fully rendered in 3-dimensions, whilst using a custom projection matrix to render sprites in 2-dimensions, Fig8.



```
Objects rendered = 10000
2728.401 | 0.000 | 2922.056 Camera Position
Camera type Perspective
2.000 Camera zoom
Application average 43.778 ms/frame
22.84 fps
```

Fig7 – Rendering of numerous textured cubes, sets of 6 quads, with a 3d perspective projection



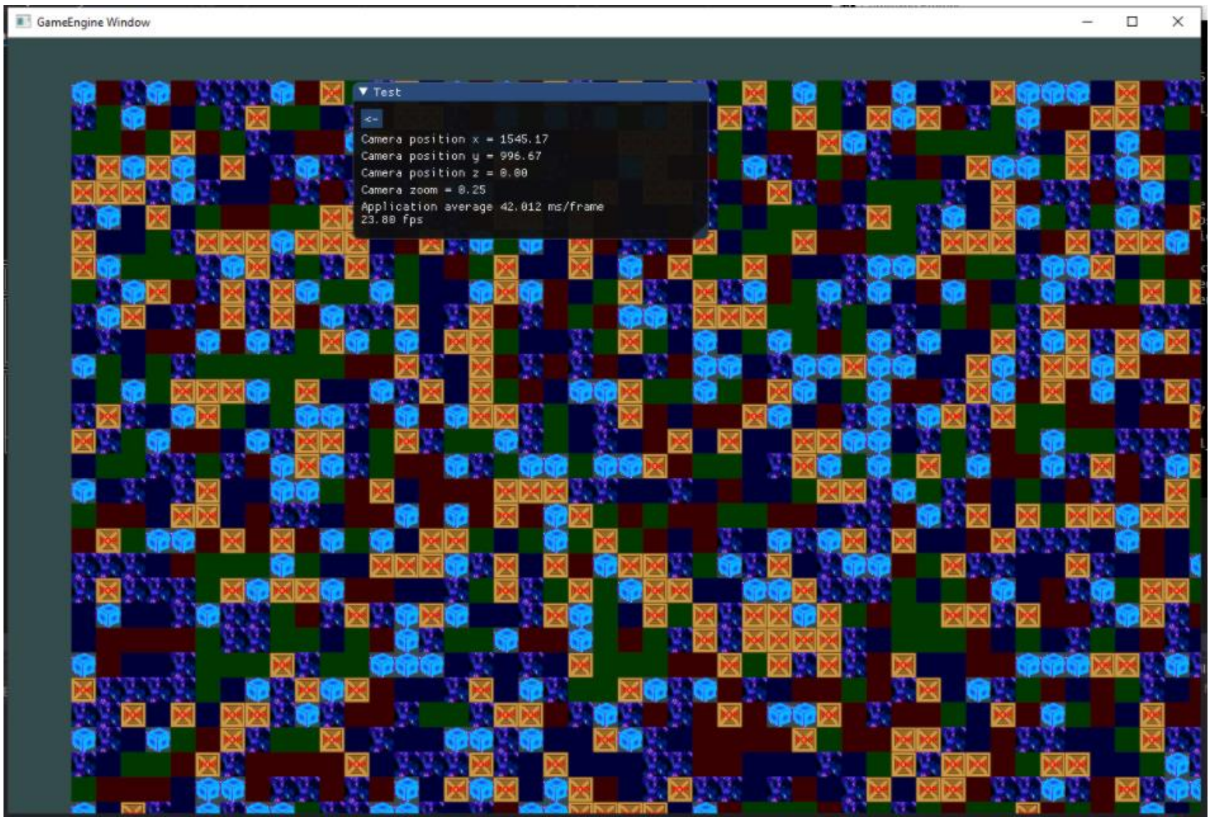
```
391fps 5afe38ef
Objects culled: 2631
```

Fig8 – Rendering from final version of the game, floor rendered with perspective projection and sprites (character & objects – trees, shrubs) rendered with custom model projection nullifying perspective

2.1.3 Optimisations

Batched rendering

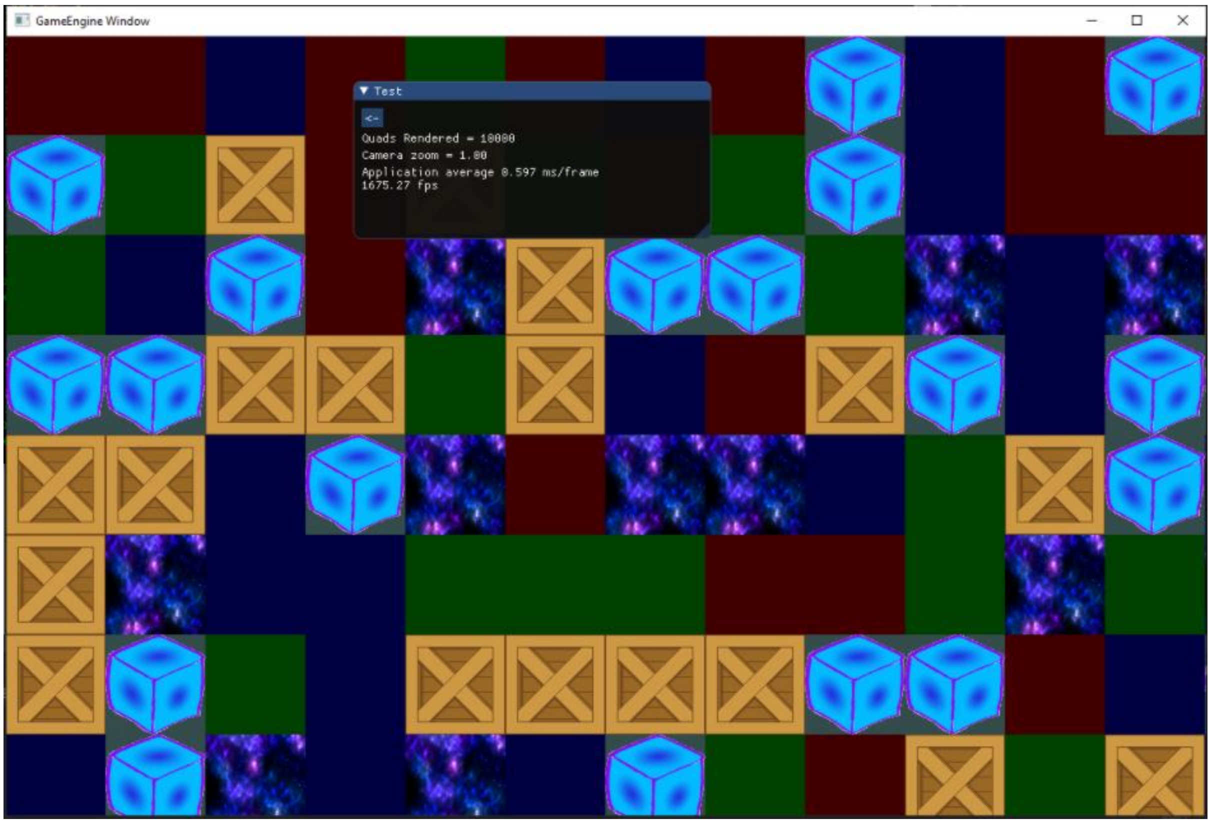
Initially, in early development version 0.9.9, rendering of quads proved to be a cumbersome task. For every object in the game data of each quad was individually being sent to the GPU and drawn to screen. When the number of objects in game increased to 10,000 this had a dramatic effect on the GPU running at frame rates of approx. 24 fps, fig9.



```
Camera position x = 1545.17
Camera position y = 996.67
Camera position z = 0.00
Camera zoom = 0.25
Application average 42.012 ms/frame
23.80 fps
```

Fig9 – 10,000 quads rendered with single draw call per quad, 25fps

Hazel engine [5] employs a caching technique, whereby a maximum number of quads, typically ten thousand, are saved in memory between each draw call. When the maximum number of quads is reached, a draw call is made sending all quads data in one batch. Optimisations were made to the Renderer class adopting this method, though direct implementation is original. We further extended this method to allow caching of static geometry, objects whose positions and render data did not change between frames. Static objects data were sent to the GPU on initialization, and merely referenced each frame they were required to be drawn. This was an in-depth overhaul which saw improvements of over 1500 frames per seconds increase, fig10.



```
Quads Rendered = 10000
Camera zoom = 1.00
Application average 0.597 ms/frame
1675.27 fps
```

Fig10 – Batched rendering of 10,000 quads at 1675 frames per second

<u>Frustrum culling</u>

Culling objects is the process of eliminating objects not within the viewing frustrum of the camera from the rendering process, Fig11.

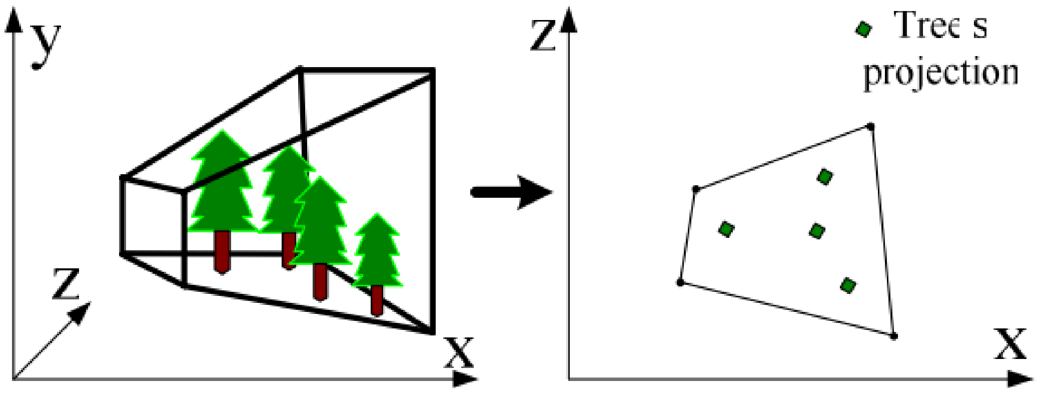
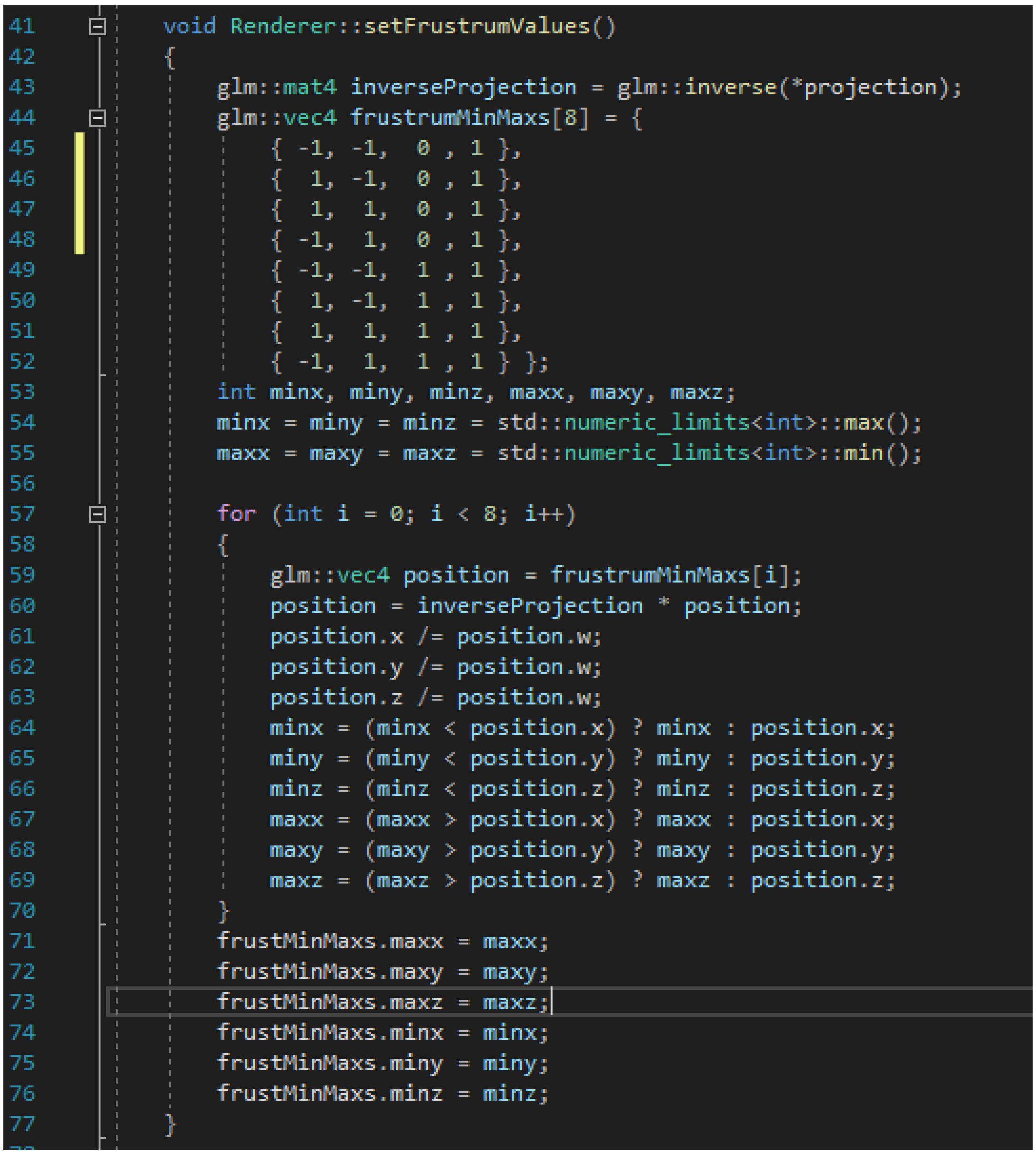


Fig22 viewing frustrum of a 3d projection [7], items in the frustrum to be drawn, any items outside to be culled

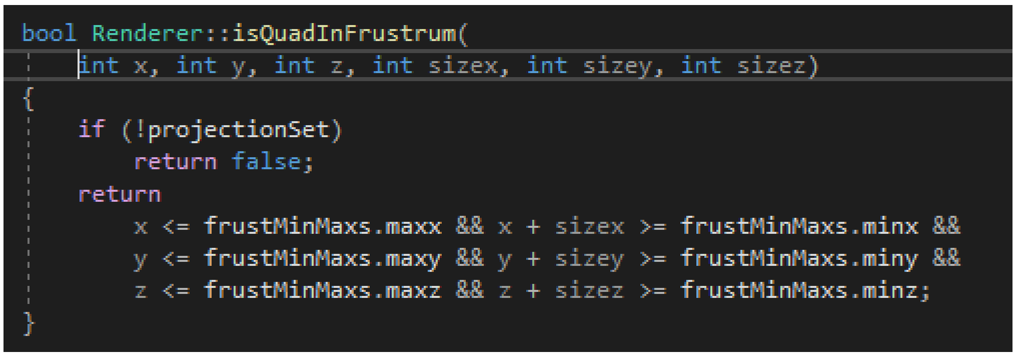
Wang et al., [7] proposed a method to obtain vector points that are inside the frustrum by projecting a point from world space into normalized device to compare for culling. However, the inverse is achieved in [17] and adopted in the renderer. Knowing the minimum and maximum normalized device coordinates in OpenGL post projection, -1 < x < 1, -1 < y < 1, 0 < z < 1, the inverse projection matrix can be applied on all corners of the frustrum to obtain an AABB in world coordinates, Fig23



```cpp
void Renderer::setFrustrumValues()
{
    glm::mat4 inverseProjection = glm::inverse(*projection);
    glm::vec4 frustrumMinMaxs[8] = {
        { -1, -1,  0 , 1 },
        {  1, -1,  0 , 1 },
        {  1,  1,  0 , 1 },
        { -1,  1,  0 , 1 },
        { -1, -1,  1 , 1 },
        {  1, -1,  1 , 1 },
        {  1,  1,  1 , 1 },
        { -1,  1,  1 , 1 } };
    int minx, miny, minz, maxx, maxy, maxz;
    minx = miny = minz = std::numeric_limits<int>::max();
    maxx = maxy = maxz = std::numeric_limits<int>::min();

    for (int i = 0; i < 8; i++)
    {
        glm::vec4 position = frustrumMinMaxs[i];
        position = inverseProjection * position;
        position.x /= position.w;
        position.y /= position.w;
        position.z /= position.w;
        minx = (minx < position.x) ? minx : position.x;
        miny = (miny < position.y) ? miny : position.y;
        minz = (minz < position.z) ? minz : position.z;
        maxx = (maxx > position.x) ? maxx : position.x;
        maxy = (maxy > position.y) ? maxy : position.y;
        maxz = (maxz > position.z) ? maxz : position.z;
    }
    frustMinMaxs.maxx = maxx;
    frustMinMaxs.maxy = maxy;
    frustMinMaxs.maxz = maxz;
    frustMinMaxs.minx = minx;
    frustMinMaxs.miny = miny;
    frustMinMaxs.minz = minz;
}
```

Fig12, obtaining AABB frustrum values in world coordinate system to perform AABB collision check



```cpp
bool Renderer::isQuadInFrustrum(
    int x, int y, int z, int sizex, int sizey, int sizez)
{
    if (!projectionSet)
        return false;
    return
        x <= frustMinMaxs.maxx && x + sizex >= frustMinMaxs.minx &&
        y <= frustMinMaxs.maxy && y + sizey >= frustMinMaxs.miny &&
        z <= frustMinMaxs.maxz && z + sizez >= frustMinMaxs.minz;
}
```

Fig13, AABB check against frustrum min/max's

With the bounding box of the frustrum in world coordinates, a simple AABB collision check is applied to an input object's coordinates in world space, Fig13. This saw an approximate framerate increase of fifty percent when the camera is not fully zoomed out Fig's 14 & 15. However, the framerate of all objects is proportional to the number of objects rendered. If Fig15 had double the objects, the framerate would be significantly lower

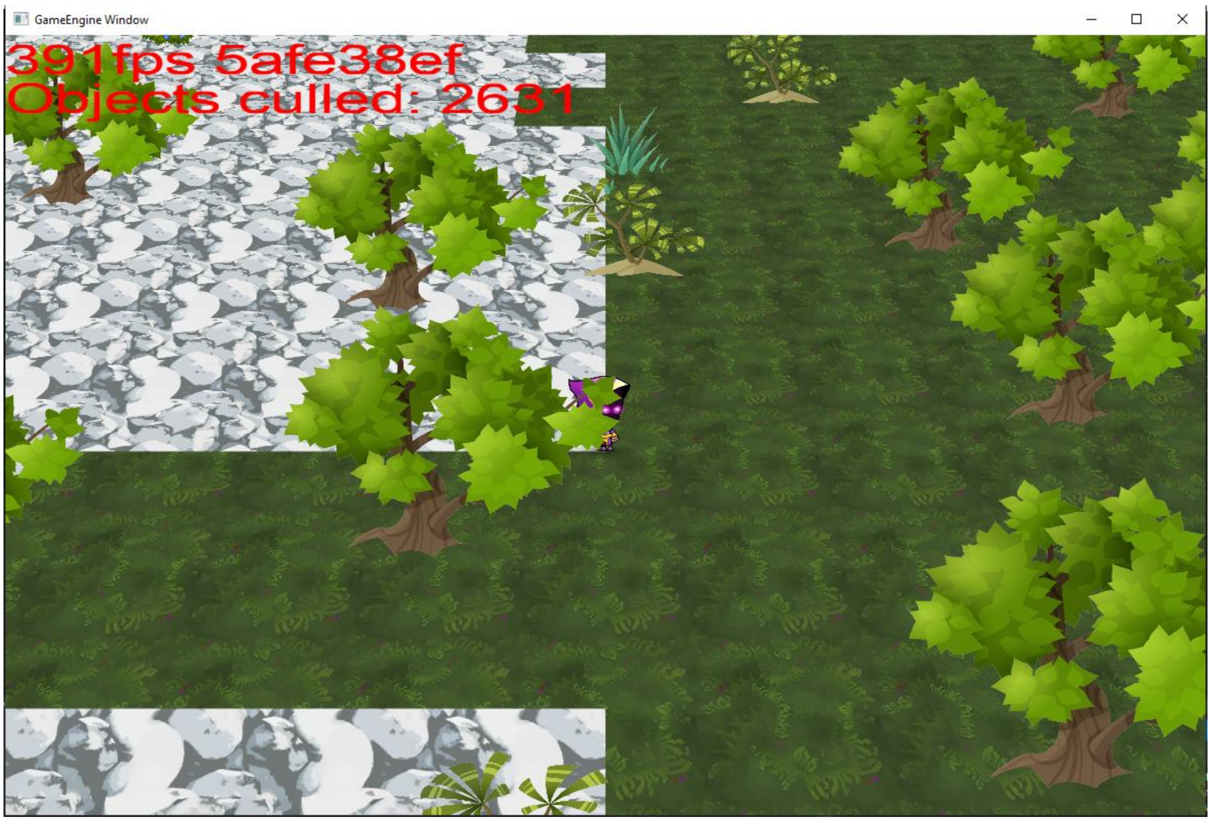


Fig14, Rendered frame with camera zoomed close to character culling 2631 objects and 391fps

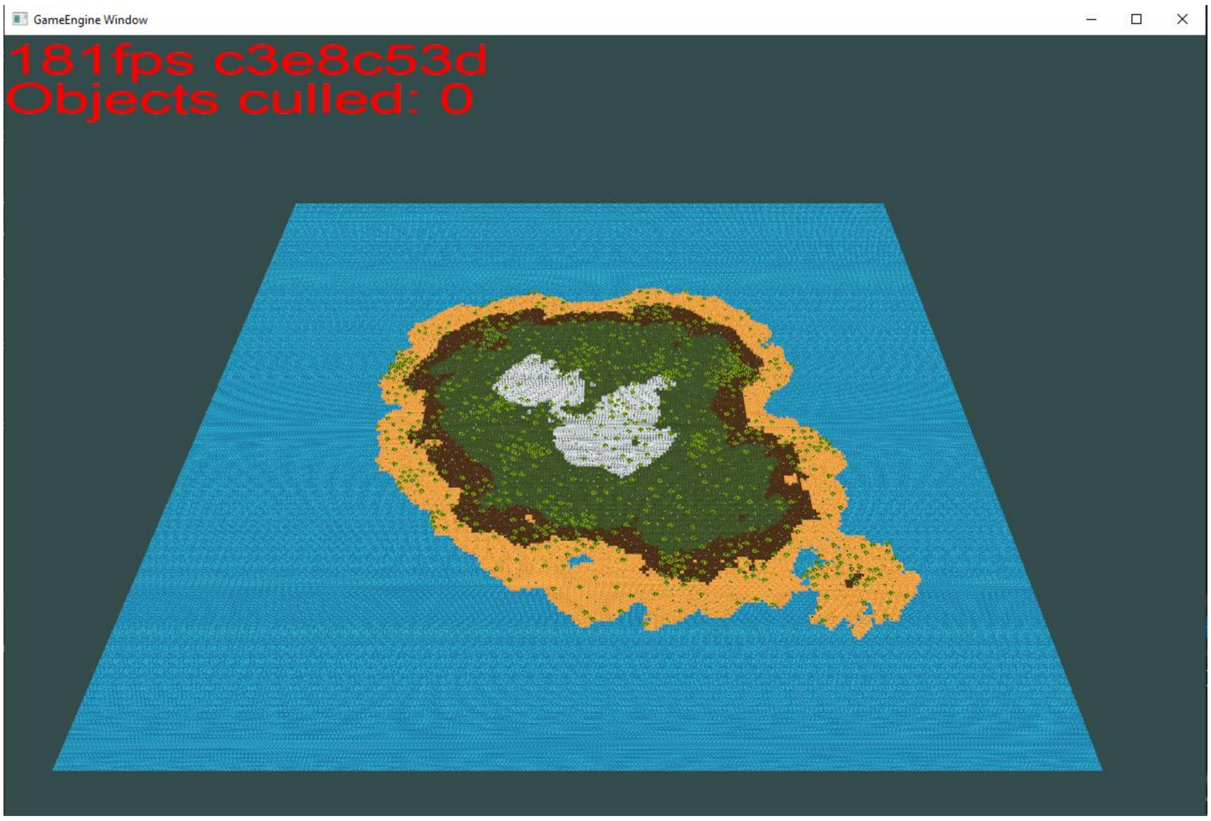


Fig15, Rendered frame with all objects in frustrum, 0 culled and 181 fps

2.2 PCG World Generation

Noise based functions are popular PCG methods which have a wide array of applications for modelling natural phenomena [19]. They are functions which act as random number generators, but rather than being completely random they show pseudo-random characteristics, parameterized to generate noise patterns [55]. There applications have seen wide usage such as Atmospheric Cloud modelling [54], marble texture generation [27], generating tracks in a runner-based game [56].

This game focuses on the Diamond-Square noise algorithm, and produces a new implementation based upon the abstract description provided in [19]. We also implement an adapted form of Perlin noise highlighted in [57], where the only modifications to the code are conversion to C++ and the use of a seeded random-number generator to populate the permutation table.

2.2.1 Diamond-Square terrain generation

Diamond-square algorithm is a stochastic process for generating a set of points in 2-Dimensional space, first suggested by Fournier et al, [21]. This is a noise generation method belonging to fBm family. Mandelbrot et al, [20] first identified fBm as families of "Gaussian random functions" which show the interdependence between points and their distance. Two points in n-dimensional space are suggested to show a strong interdependence in random chance phenomena [20]. This is achieved with area sub-division. First, assigning the outermost corners of the space normalized random values, and performing subdivision of the space to take an average of the surrounding areas [21].

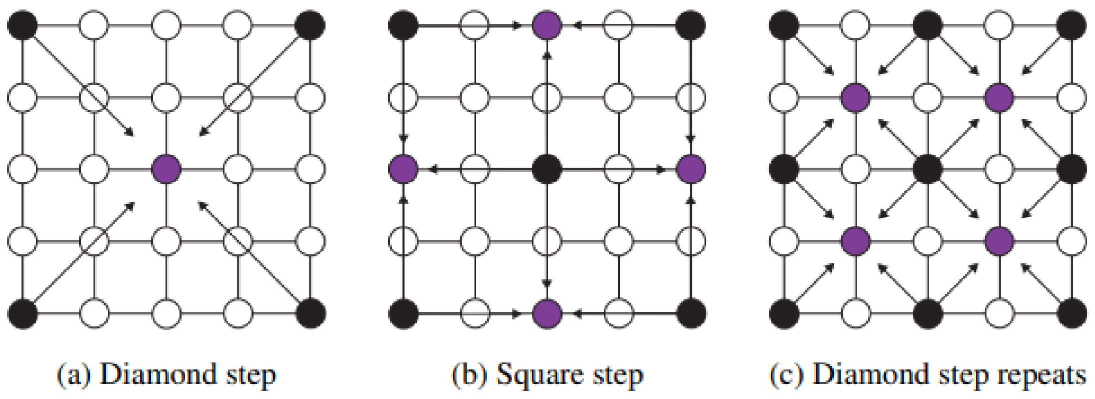


Fig16, Diamond square steps illustrated [19]

Subdivision of areas is shown in Fig16 for both diamond and square steps. Pseudo code for the algorithm is as follows:

```
1. Assign random value to each corner
2. Attain centroid, assign its value the average of corners plus some random roughness value
   (square step, Fig16 a)
3. For each adjacent point (diamond step, Fig16 b):
    a. Assign average of centroid and adjacent points, plus random roughness value
4. Sub divide area
5. For each sub area, repeat steps 2 – 5 whilst dividing roughness value by 2, until all points are
   assigned (Fig16 c)
```

Higher values of the roughness division value (2 in pseudo code) give rise to greater stochastic characteristics. Conversely lower values give a closer relation between neighbours, having a smoother gradient of values.

Fig17 shows the resultant sample map from Diamond Square, with values closer to 1 appearing white and values closer to 0 black. This map is used much like a height map, but instead of using height values to raise and lower geometry we use them to partition ranges to associated tile types. Values 0 – 0.2 are water, 0.2 – 0.4 sand, 0.4 – 0.6 dirt, 0.6 – 0.8 grass, and > 0.8 mountain, Fig18

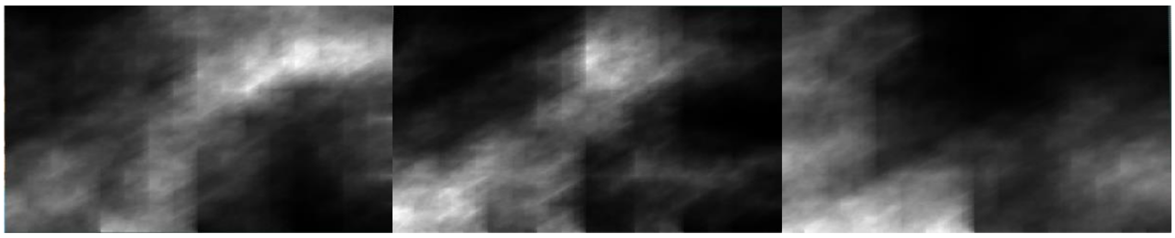


Fig17, three noise maps generated with Diamond Square algorithm

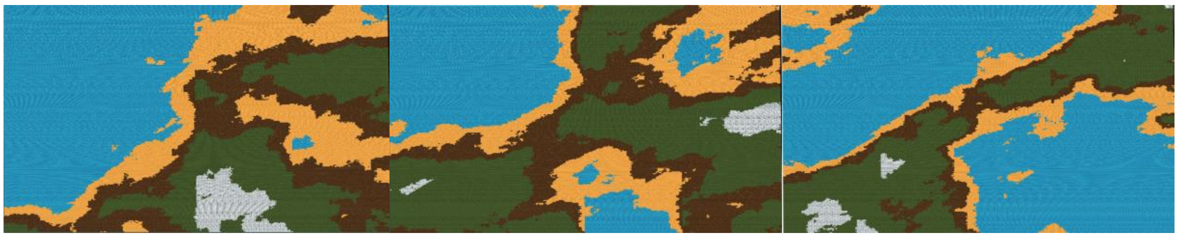


Fig18, three noise maps post association to height types using custom-made textures – blue water, yellow sand, brown dirt, green grass, and white mountain

The resultant height association produces a uniform range of stochastic regions, which alone is desirable in certain games. However, in survival games, the player is often isolated in a smaller region such as an island. For this we use post-processing techniques identified in [22], offering several methods to perform morphological transformations. Mainly, areas closer to the edge are lowered, areas closer to the centre are raised, but other transformations are applied also. Results, Fig19, produce terrains which have island characteristics.

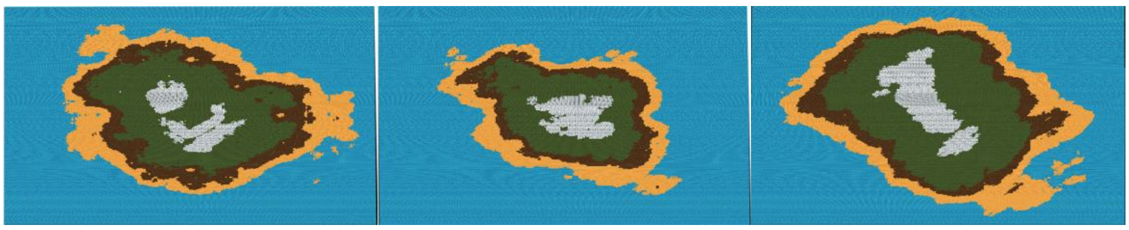


Fig19, three maps following transformations to create island shaped terrain

While Diamond-Square proved to be simple to implement, it does have short comings. Due to the sub-division of areas in noise generation, all maps are required to be of size $2^n + 1$ in size. This results in the number of operations having $O(2^n)$ complexity, np hard. At map sizes > 512 the map generation process became highly inefficient with respect to time. In addition to this, the size of noise map needs to be known in advance, and all points require generation in advance of sampling. Alternative approaches, such as Perlin noise, do not have these shortcomings; but do have a far greater learning curve to understand.

2.2.2 Poisson Disc Sampling

Sampling is a particularly useful technique used for generating random points in n-dimensional space [28]. It has many application domains including graphics, imaging, and geometry processing [58]. Poisson-disc sampling is a stochastic sampling method of generating tightly packed points in Euclidean space in which no two are within a minimum distance [23, 24]. We use this an adapted form of this sampling technique to produce positions for item placement in game.

General Pseudo code for the algorithm is as follows:

```
1. Let minDist be the minimum distance between points, numTries be the number of attempts
   to find additional points
2. Initialize frontier, pointsMade to be empty sets of points
3. Generate point of random position in space and add to frontier and to pointsMade
4. While frontier has members:
    a. Pick random point from frontier as p1
    b. For 0 until numTries
        i. Generate point at random angle, and of distance minDist plus random *
           minDist to p1 as p2
        ii. If p2 outside of space or exists a point in pointsMade within distance minDist:
                Continue next iteration
        iii. Else:
                Add point to pointsMade and to frontier
    c. Remove p1 from frontier
```

Fig20 Shows the approach for generating a point, where max distance is min distance * 2. A random angle is chosen between 0-360º and a random radius is generated between min distance and max distance. The final output of a Poisson-Disc sampling is shown in Fig21

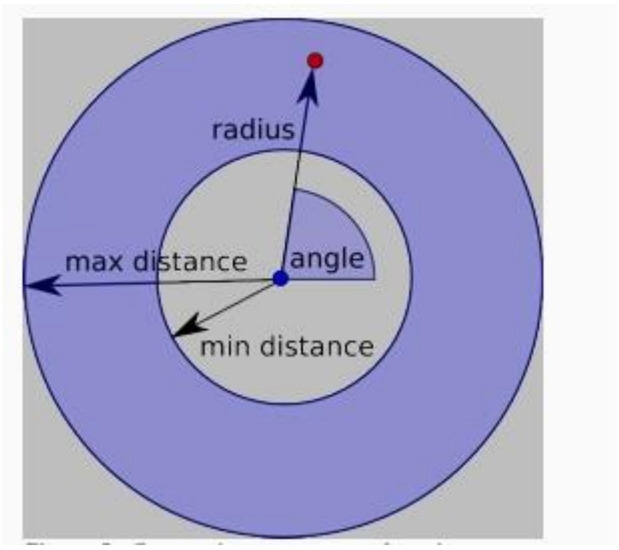


Fig20 characteristics of a points generated about a chosen point in Poisson disc sampling [26]

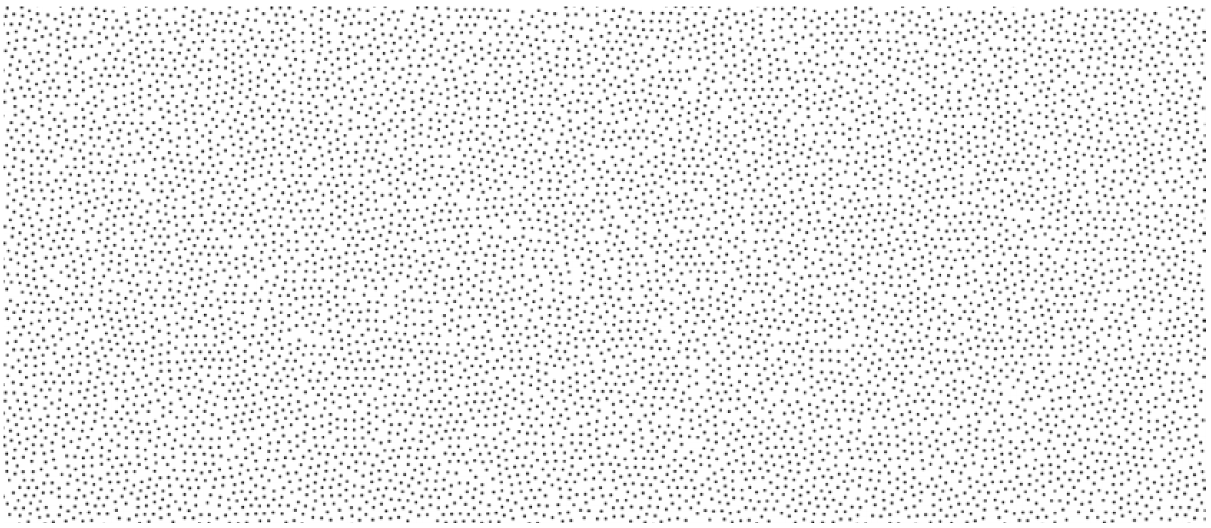


Fig21 Points generated with Poisson disc sampling [28]

Dwark et al., [25] highlight a variable density approach of Poisson-disc sampling where a parameterized function governs the minimum and maximum distance between points, allowing for varying density of the points generated. An example of an externally sourced variable density Poisson-Disc sampling can be seen in Fig22.

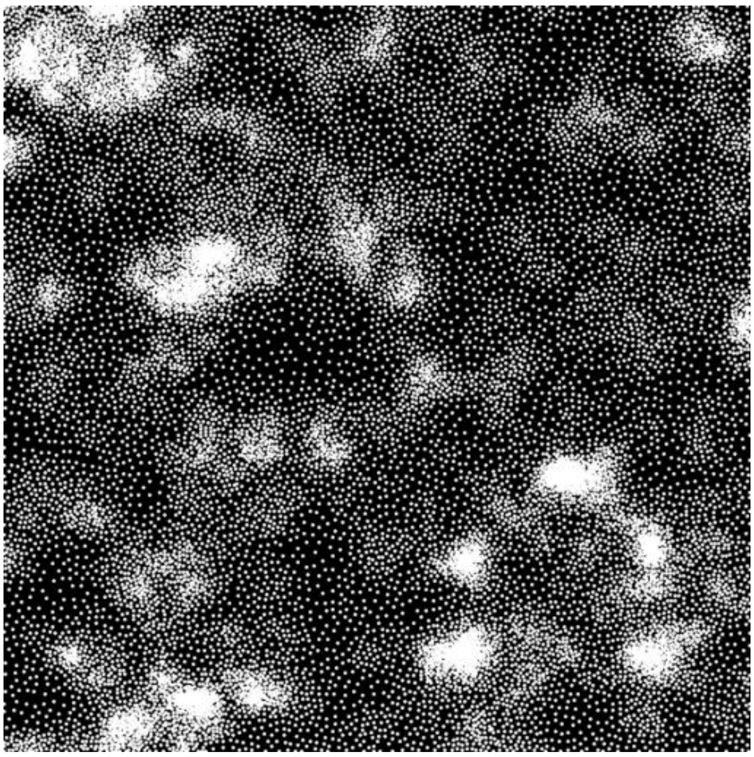


Fig22, points generated using Variable Density Poisson-disc sampling [26]

We use Perlin noise [27] for the function to govern distance for its ability to generate high frequency blue noise. We also introduce a distance modifier parameter, such that minimum distance of points becomes minDist = minDist + (1 – noise) * distMod. The output achieved is shown in Fig23, where objects can be seen to be clustered where noise levels are high, and sparce where noise levels are low.

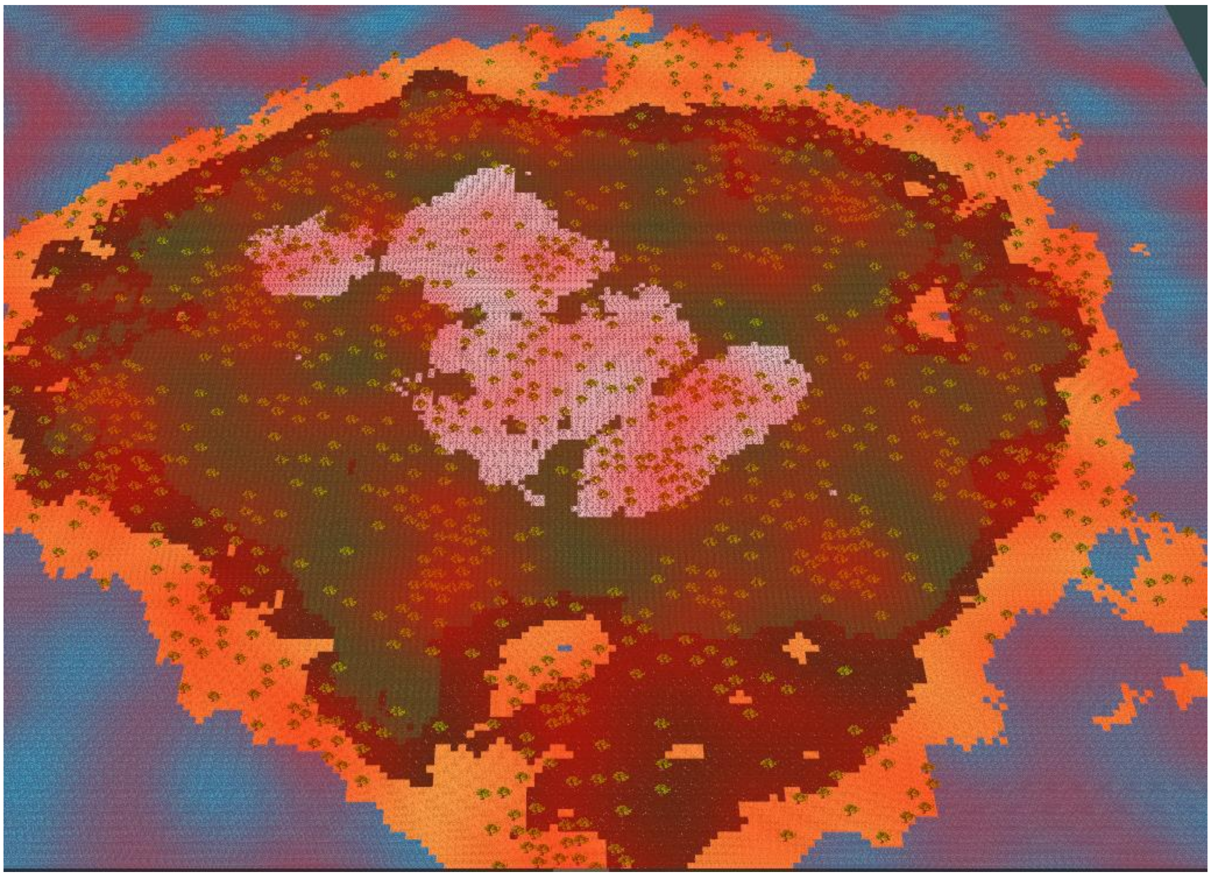


Fig23, points generated with Variable Density Poisson-disc sampling shown as trees (green blobs) using Perlin's noise to govern density. Noise is overlayed in red with higher values of noise having higher opacity

2.3 Miscellaneous features

2.3.1 Object Selection

To enable the player to interact with the virtual world and its environments it is important to detect what game object exists underneath the mouse's position on screen. We employ a technique detailed in [47]. The premise is that each object in game is assigned a UID. The ID is then converted into it a 24-bit representation, and each 8-bit sections are obtained by the bitwise AND function and a left wise bit shift, e.g Fig24 to obtain green channel. The final colour is encoded as Fig25

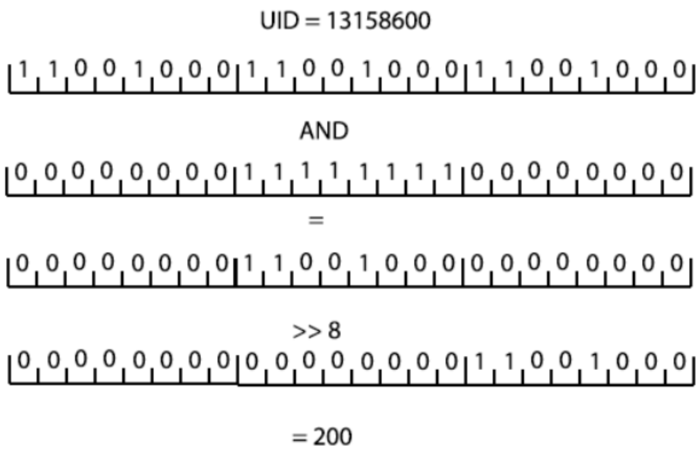


Fig24, bitwise encoding of UID to green channel of rgb colour

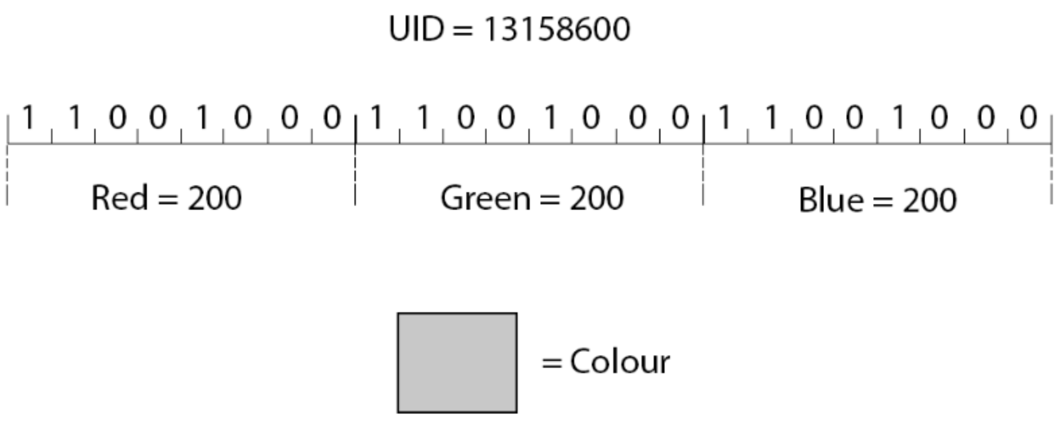


Fig25, UID converted into 24-bitstring, RGB-channels separated to 8-bitstring, resultant colour achieved

With the UID's converted into and RGB colour representation, the objects are then rendered to a hidden screen not visible by the player, Fig26.

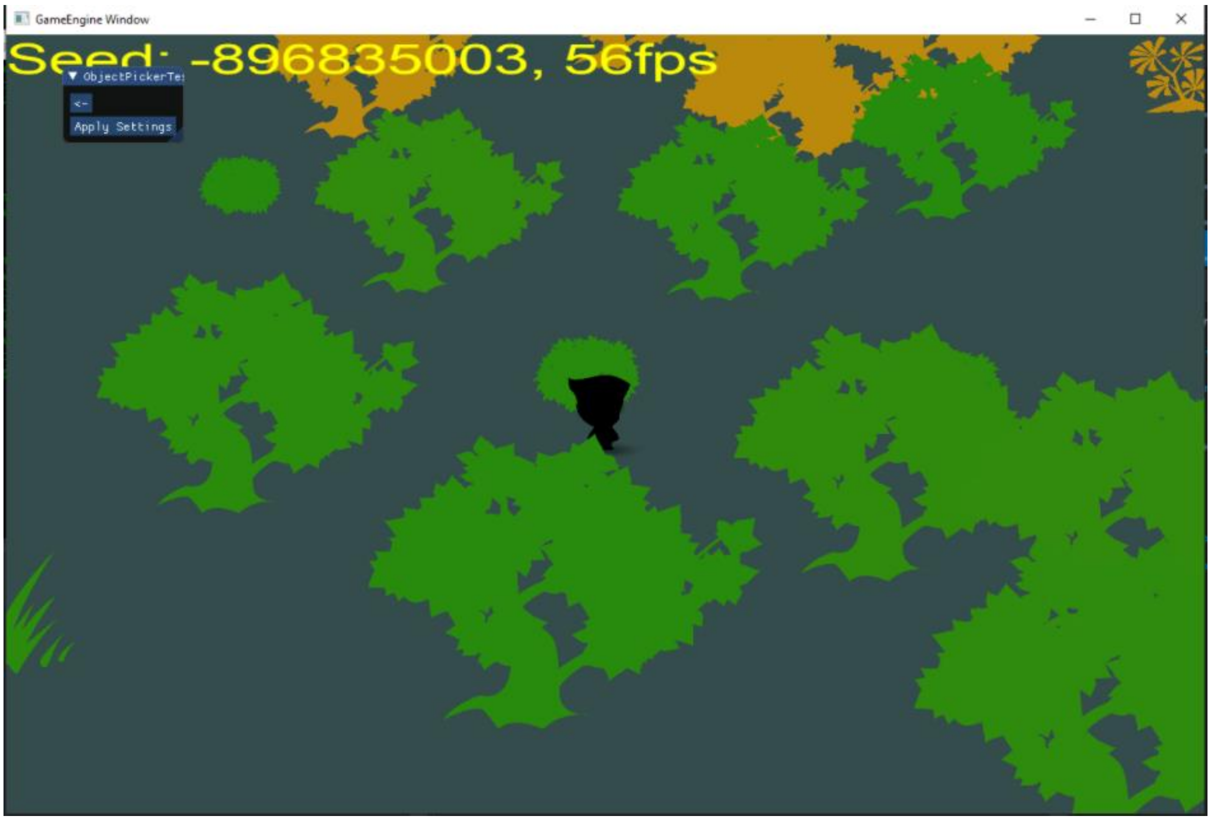


Fig26, objects on screen rendered with their UID encoded to a 24-bit RGB colour

OpenGL employs a method 'glReadPixels' which enables attaining the red, green, blue, and alpha values of the pixel under the mouse position. Having attained values, we then re-assemble the 24-bit UID Integer using reverse operations of Fig24 to determine which object is under the mouse. Fig27 shows the identification of an object under the mouse with object highlighted using a stencil renderer to brighten the textures value.

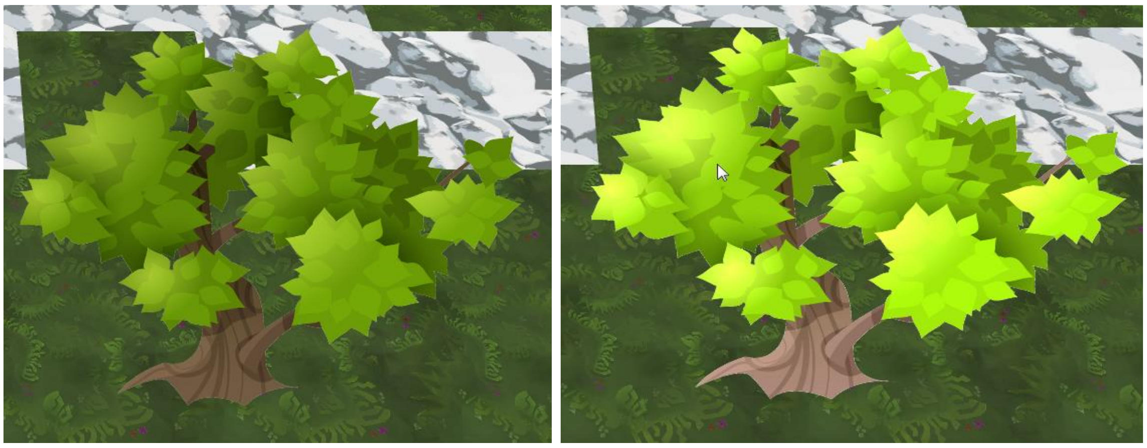


Fig27, tree before and after identification with tree highlighted

This technique has one drawback, only 2^24 objects can be encoded. This could be improved upon by including the alpha channel in the encoding giving 2^32 possible objects. Alternatively, rather than assigning every object an ID, we could assign IDs to only those objects which are rendered on screen. It would be highly unlikely that 2^24 objects would be rendered on screen at a single point in time. However, the method provided is substantial enough for the size of this project.

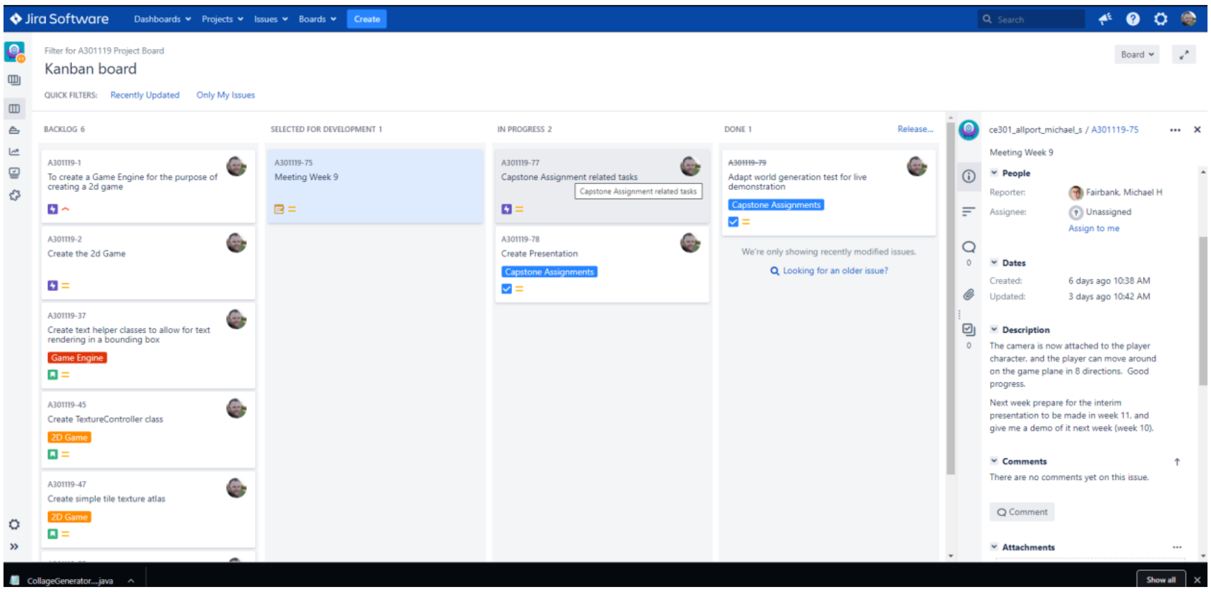
Chapter 3 – Full technical documentation

Project Technical documentation - Table of Contents

Chapter 4 – Project Planning

4.1 Methodology

This project employed the Agile Framework during development using the Kanban system. Jira Software was used to assist in this project management for its use of ticketing system and agile rich features. Kanban helped to visualize features and tasks with its swim lanes providing a good overview of what features and issues were being undertaken, Fig24.



BACKLOG 6

A301119-1
To create a Game Engine for the purpose of creating a 2d game

A301119-2
Create the 2d Game

A301119-37
Create text helper classes to allow for text rendering in a bounding box
Game Engine

A301119-45
Create TextureController class
2D Game

A301119-47
Create simple tile texture atlas
2D Game

SELECTED FOR DEVELOPMENT 1

A301119-75
Meeting Week 9

IN PROGRESS 2

A301119-77
Capstone Assignment related tasks

A301119-78
Create Presentation
Capstone Assignments

DONE 1

~~A301119-79~~
Adapt world generation test for live demonstration
Capstone Assignments

We're only showing recently modified issues.
Looking for an older issue?

ce301_allport_michael_s / A301119-75
Meeting Week 9

People
Reporter: Fairbank, Michael H
Assignee: Unassigned
Assign to me

Dates
Created: 6 days ago 10:38 AM
Updated: 3 days ago 10:42 AM

Description
The camera is now attached to the player character, and the player can move around on the game plane in 8 directions. Good progress.

Next week prepare for the interim presentation to be made in week 11, and give me a demo of it next week (week 10).

Comments
There are no comments yet on this issue.
Comment

Attachments

Fig24, overview of Kanban board with issues selected for development in use of swim lanes

Additionally, three epic issues were created to help categorize features being developed – Game Engine, 2.5D Game, and Assignment Related tasks.

Weekly meetings were set in place with my supervisor in which features, and goals were set. This enabled continuous momentum throughout the duration of the project and resulted in a weekly rollout of features. Most weeks saw individual features being developed, with few occasions where more complex features took two weeks to develop such as the world generation.

Overall project momentum can be seen in Fig25 showing task completion over the course of the product. Over 120 tasks have been completed during the duration of this task

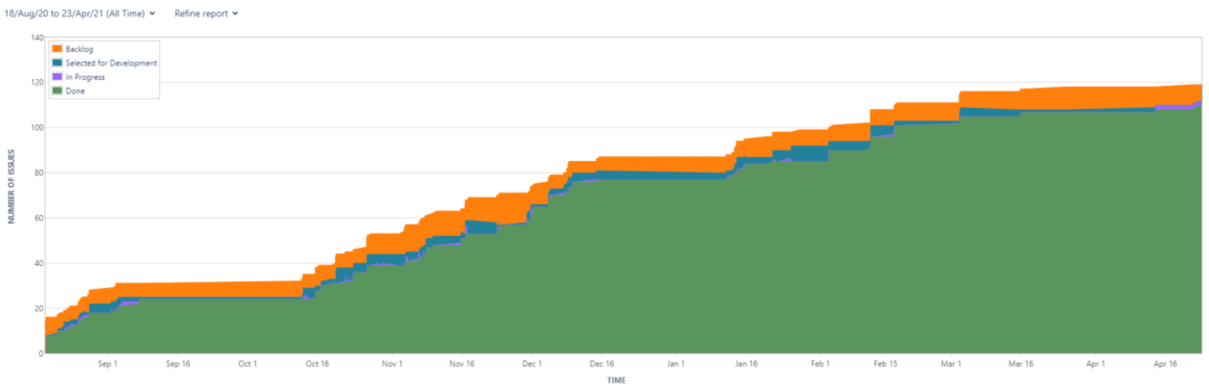


Fig25, Cumulative Flow diagram showing tickets raised, completed, selected for development over time

Fig 26 shows the Control Chart for the project. The average duration of tasks completion were 7 days, with many features being completed much sooner. Larger epics/stories saw much larger completion time, up to 60 day+ mark.

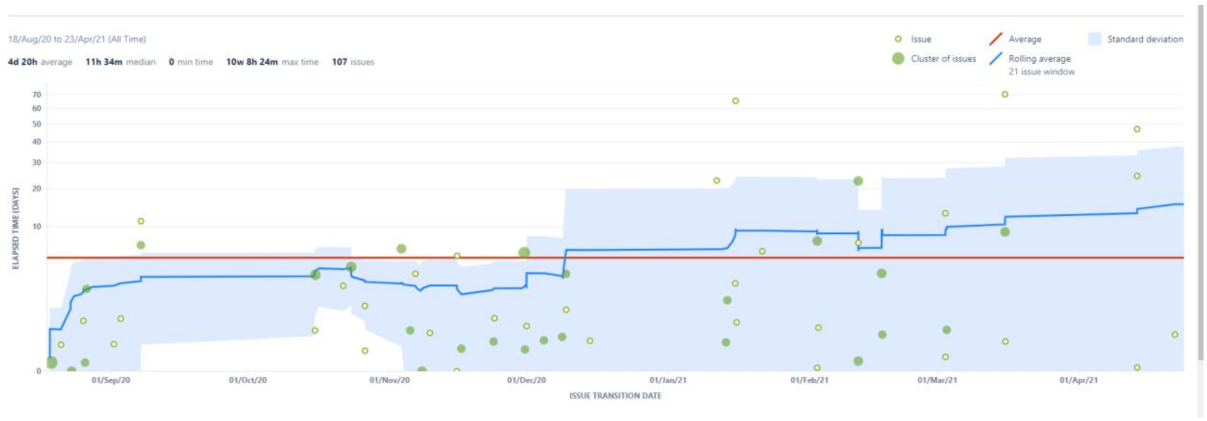
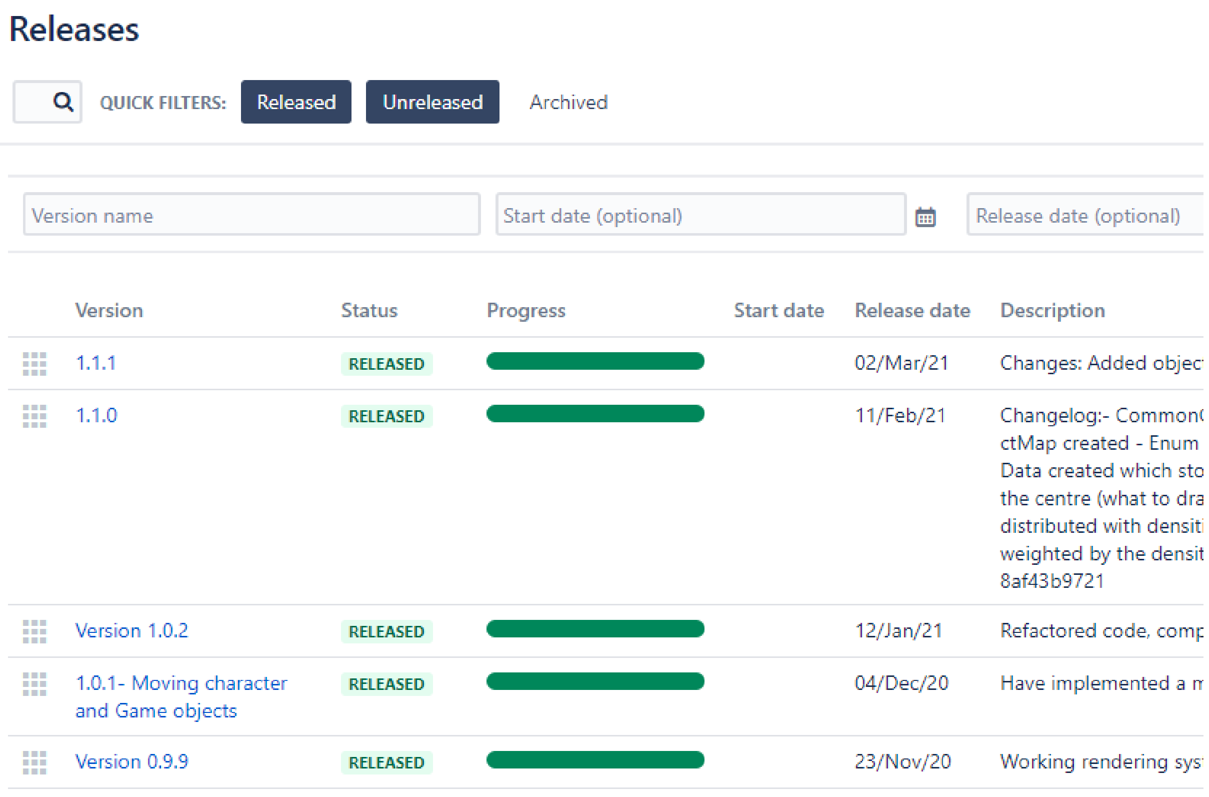


Fig27, Control Char for the project, with hollow circles showing issue completion, filled circles being a cluster of issues, red line being average completion time, and blue line being efficiency

Moreover, Jira provided the ability to release issues in a batch of versions associated with software versions. This allows for an oversight of issues and stories completed within the release of a software version, Fig27 and Fig28



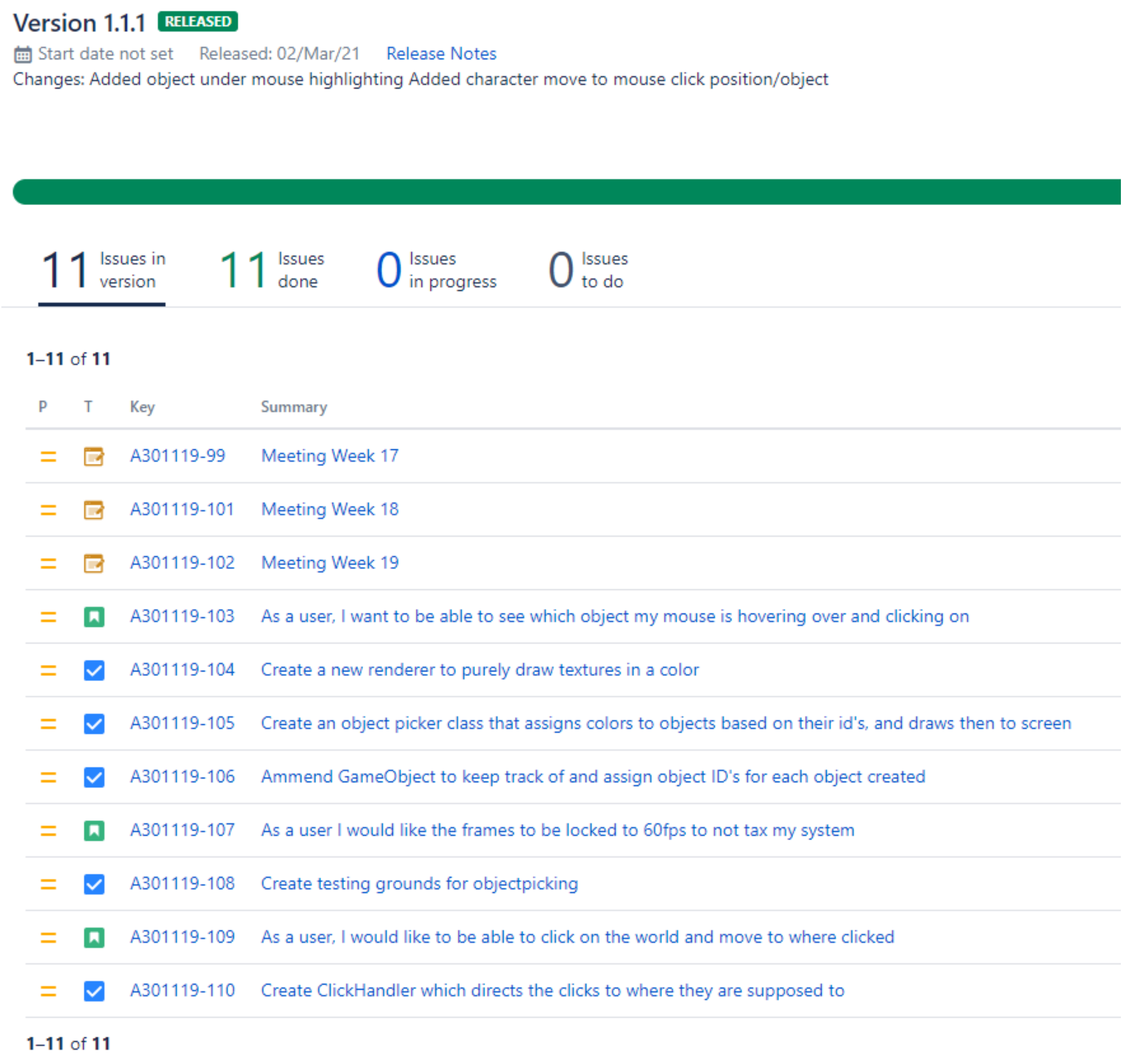
Releases

QUICK FILTERS: Released Unreleased Archived

Version name | Start date (optional) | Release date (optional)

	Version	Status	Progress	Start date	Release date	Description
	1.1.1	RELEASED	🟩		02/Mar/21	Changes: Added objec
	1.1.0	RELEASED	🟩		11/Feb/21	Changelog:- Common(ctMap created - Enum Data created which sto the centre (what to dra distributed with densit weighted by the densit 8af43b9721
	Version 1.0.2	RELEASED	🟩		12/Jan/21	Refactored code, comp
	1.0.1- Moving character and Game objects	RELEASED	🟩		04/Dec/20	Have implemented a m
	Version 0.9.9	RELEASED	🟩		23/Nov/20	Working rendering sys

Fig27, Use of version releases overview


Version 1.1.1 RELEASED

Start date not set   Released: 02/Mar/21   Release Notes

Changes: Added object under mouse highlighting Added character move to mouse click position/object

11 Issues in version   11 Issues done   0 Issues in progress   0 Issues to do

1–11 of 11

P	T	Key	Summary
=		A301119-99	Meeting Week 17
=		A301119-101	Meeting Week 18
=		A301119-102	Meeting Week 19
=		A301119-103	As a user, I want to be able to see which object my mouse is hovering over and clicking on
=	✓	A301119-104	Create a new renderer to purely draw textures in a color
=	✓	A301119-105	Create an object picker class that assigns colors to objects based on their id's, and draws then to screen
=	✓	A301119-106	Ammend GameObject to keep track of and assign object ID's for each object created
=		A301119-107	As a user I would like the frames to be locked to 60fps to not tax my system
=	✓	A301119-108	Create testing grounds for objectpicking
=		A301119-109	As a user, I would like to be able to click on the world and move to where clicked
=	✓	A301119-110	Create ClickHandler which directs the clicks to where they are supposed to

1–11 of 11

Fig28, List of Stories and Issues achieved within version 1.1.1

4.2 Risk Management

Agile methodologies suggest employing prioritization methods in a project lifecycle. One such method is MoSCoW [48, 49]. This breaks tasks and features down into four categories – “Must have”, “Should have”, “Could have”, and “Won’t have”.

From the projects offset our targets were to create a 2.5d survival game. To create a game using C++ we would need a subset of features to help achieve this such as rendering, responding to use input. It was also required to employ procedural content generation to create our world. It was desirable to have 5 minutes of gameplay.

Initially all requirements would have been “Must Haves”, and risks were assessed weekly. One risk came to fruition where near the end of the project it was realised we wouldn’t attain 5 minutes of gameplay. This target was reassessed, and focus shifted on enriching the features already developed.

4.3 Version Control

GitLab was used continuously through this project, seeing 195 commits with two branches used, Fig25. When a fundamental feature was being released, the test branch was used to ensure changes had no effect on the system overall. Using version control software allowed for ease of rollbacks when new features and bugs appeared. This also prevented loss of data when switching systems.

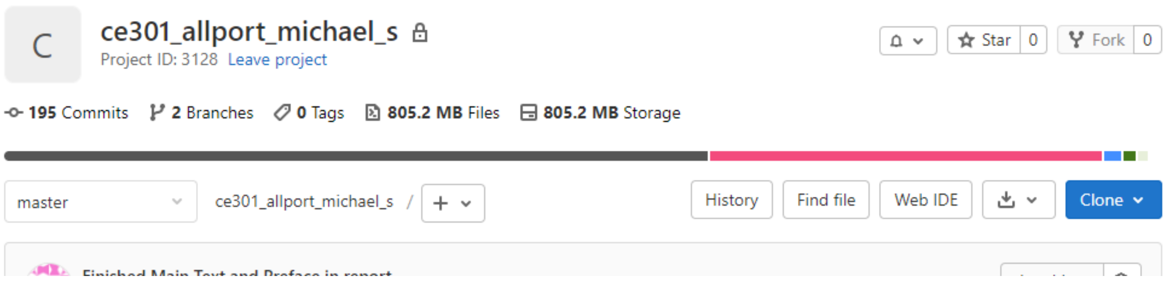


Fig24, git overview showing 195 commits and two branches

4.4 Reflection

The use of Agile methodologies proved to be highly beneficial in this project. Features were categorised based on the system being worked on, Game Engine, Game, or Assignment tasks. Use of stories allowed for a feature to be broken into numerous tasks to achieve the goals. The Kanban board allowed oversight of what tasks were being selected for development, in progress, and the subset of features that had been completed during the version of the software.

Weekly workloads were discussed between the developer and supervisor, reflecting upon work achieved throughout the previous week and the benefit they had towards the overall goals of the project. The project saw a good momentum of work being done throughout, with each week developing features which contributed to the software aims.

Version control was invaluable during the project. When bugs were experience, it was possible to roll back the software to before the introduction of a bug to examine their exact cause. Additionally, when a large feature was being introduced it was possible to create a new working branch such that the feature did not affect the overall structure of the code before combining said feature into the master branch.

Chapter 5 – Discussion

5.1 Conclusion

This report set out the context and premises for the project. The origins of 'Game Engines' was explored, and it was found that there are systems in game development that are totally independent of game logic. Some general game engines in industry were found, alongside several open source engines which became the focus of the design for this projects engine. Several subsystems were implemented – an Event subsystem, a Rendering engine, GPU based asset management, Logging, and the Base Application which provided window context. The rendering process was highlighted showing how basic geometry in drawn to screen. Optimisation were also made enhancing the performance of the system.

Additionally, we explored the survival game genre to which became the focus of our game's design. Noise based Procedural Content Generation methods were utilized, which saw custom implementation of the Diamond Square algorithm from its abstract high-level description to generate terrain. We used an adapted variable density Poisson-Disc sampling method to choose locations to spawn objects. A basic 2.5d game was presented which had a PCG world generator for both terrain and object placement within the world of trees, shrubs, twigs, and other resources.

Furthermore, we employed additional game design elements to enable the player interacting with the world. Objects UID's were encoded in order to determine the object pointed to under the mouse.

Concluding the report was a discussion of the project management tools used for this product. We looked at the Agile methodologies employed with the Kanban system. Features of the product were split between epics, user stories, and an issue ticketing system. We also saw the use of Version Control System GitLab and its use in this project.

5.2 Future Works

The PCG methods explored here are only a small subset of this area. Where we have presented a basic form of terrain generation, this could easily be examined in more depth. There exist methods of simulating real-world corrosion which would further add detail to a map such as rivers and streams. Furthermore, these methods could be easily be extended to a real 3D application such that noise would be used for heightmaps of elevations.

While a basic game has been presented here, it is far from being an actual game where the player can utilize the resources available. Non-playable characters could be introduced to challenge the user. Health bar systems and hunger systems could be introduced which are commonly found in survival games. There are many aspects of game design that could be introduced to expand this project.

If extending the game, it would also be beneficial to extend the game engine. A sound manager should be made providing the use of audio within the game, be it music or in-game effects. A physics engine could be made, alongside collision-detection system. The FontRenderer has partial implementation in the current state and all text is rendered on one line. This could be extended using an algorithm which places a string of text in a bounding box, automatically adjusting the texts width accordingly and allowing for scrolling of the text if oversized.

5.3 Problems faced

One of the major problems face was the unrealistic scope, first highlighted by supervisor Dr Michael Fairbank in the offset of planning. This type of project in industry is typically undertaken by teams of developers, alongside individuals of other professions such as Art Design, Animation, Story developers etc. While the product is satisfactory, it does not comprise a game where there is an end goal for completion. This made it rather difficult to meet all the requirements set out. In hindsight, a rogue-like game with PCG would have been a more attainable goal. However, the challenge was welcomed.

A further problem faced was a lack of attention to stopping and noting what had been achieved between features. My mindset was more toward "This feature has been done, now onto the next feature". It would have been more beneficial to have a break between features for work on documentation and what had been achieved.

Additionally, there were times in development where my GitLab commits were not as frequent as they could have been. Work would be undertaken on a large set of functionalities without committing. Many times, the functionality being worked on proved to be working in manual testing, and then something would happen that would break the feature. Because no committing had been done in between, it was difficult to trace where the break happened. In future projects I will be working with the commit often mindset.

Chapter 6 - References

[1] J. Gregory, *Game Engine Architecture, Third Edition*, 3rd ed. Boca Raton, FL: CRC Press Taylor & Francis Group, 2019.

[2] id Software, Doom, http://www.idsoftware.com/, April 2021. [Online]. [Accessed: 15- Apr- 2021].

[3] Valve Corporation, Half-Life, https://www.valvesoftware.com/ April 2021. [Online]. [Accessed: 15- Apr- 2021].

[4] P. Petridis et al., "Game Engines Selection Framework for High-Fidelity Serious Applications", International Journal of Interactive Worlds, pp. 1-19, 2012. Available: 10.5171/2012.418638

[5] Y. Chernikov, *Hazel Engine*. 2021. [Online]. Available: https://github.com/TheCherno/Hazel [Accessed: 15-Apr- 2021]

[6] N. Ivanov, *Ethereal Engine*. 2021. [Online]. Available: https://github.com/volcoma/EtherealEngine [Accessed: 15-Apr- 2021]

[7] S. Sharma, *No Title, OpenGL C++ Tutorials*. 2021. [Online]. Available: https://github.com/Headturna/OpenGL-C---Tutorials [Accessed: 15-Apr- 2021]

[8] de-Vries, J., 2020. *Learn Opengl, Extensive Tutorial Resource For Learning Modern Opengl*. [online] Learnopengl.com. Available at: <https://learnopengl.com/> [Accessed 12 October 2020].

[9] Kessenich, J., Sellers, G. and Shreiner, D., 2017. Opengl Programming Guide. 9th ed. Crawfordsville, Indiana: Pearson Education, Inc.

[10] Geelnard, M. and Löwy, C., 2020. *GLFW*. Available: https://www.glfw.org/

[11] Herberth, D., 2020. *Glad*.

[12] Riccio, C., 2020. *GLM*. G-Truc.

[13] Melman, G., 2020. *Spdlog*.

[14] Barrett, S., 2020. *Stb*.

[15] D. Turner, R. Wilhelm and W. Lemberg, *FreeType*. 2020.

[16] Chunhua Wang, Huijuan Xu, Hongfei Zhang and Dong Han, "A fast 2D frustum culling approach", 2010 2nd International Conference on Computer Engineering and Technology, 2010. Available: 10.1109/iccet.2010.5485876

[17] "How to get the frustum bounding box?", GameDev.net, 2021. [Online]. Available: https://gamedev.net/forums/topic/321595-how-to-get-the-frustum-bounding-box/3070895/. [Accessed: 20- Apr- 2021].

[19] N. Shaker, J. Togelius and M. Nelson, Procedural content generation in games. Cham: Springer, 2016.

[20] B. Mandelbrot and J. Van Ness, "Fractional Brownian Motions, Fractional Noises and Applications", SIAM Review, vol. 10, no. 4, pp. 422-437, 1968. Available: 10.1137/1010093

[21] A. Fournier, D. Fussell and L. Carpenter, "Computer rendering of stochastic models", Communications of the ACM, vol. 25, no. 6, pp. 371-384, 1982. Available: 10.1145/358523.358553

[22] A. Patel, 2021. [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/. [Accessed: 21- Apr- 2021].

[23] M. Ebeida, A. Davidson, A. Patney, P. Knupp, S. Mitchell and J. Owens, "Efficient maximal poisson-disk sampling", ACM Transactions on Graphics, vol. 30, no. 4, pp. 1-12, 2011. Available: 10.1145/2010324.1964944

[24] K. White, D. Cline and P. Egbert, "Poisson Disk Point Sets by Hierarchical Dart Throwing", 2007 IEEE Symposium on Interactive Ray Tracing, 2007. Available: 10.1109/rt.2007.4342600

[25] N. Dwork, C. Baron, E. Johnson, D. O'Connor, J. Pauly and P. Larson, "Fast variable density Poisson-disc sample generation with directional variation for compressed sensing in MRI", Magnetic Resonance Imaging, vol. 77, pp. 186-193, 2021. Available: 10.1016/j.mri.2020.11.012

[26] H. Tulleken, "Poisson Disk Sampling", Dev.Mag, 2021. [Online]. Available: http://devmag.org.za/2009/05/03/poisson-disk-sampling/. [Accessed: 22- Apr- 2021].

[27] K. Perlin, "Improving noise", ACM Transactions on Graphics, vol. 21, no. 3, pp. 681-682, 2002. Available: 10.1145/566654.566636

[28] D. Dunbar and G. Humphreys, "A spatial data structure for fast Poisson-disk sample generation", ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06, 2006. Available: 10.1145/1179352.1141915

[29] Unity Technologies, “Unity”, 2021. [Online]. Available: https://unity.com/. [Accessed: 22- Apr- 2021].

[30] "Unreal Engine | The most powerful real-time 3D creation platform", Unreal Engine, 2021. [Online]. Available: https://www.unrealengine.com/en-US/. [Accessed: 22- Apr- 2021].

[31]"FINAL FANTASY VII Remake", Ffvii-remake.square-enix-games.com, 2021. [Online]. Available: https://ffvii-remake.square-enix-games.com/en-gb. [Accessed: 22- Apr- 2021].

[32] "Hellblade: Senua’s Sacrifice", Hellblade.com, 2021. [Online]. Available: https://www.hellblade.com/. [Accessed: 22- Apr- 2021].

[33] "Borderlands", Borderlands.com, 2021. [Online]. Available: https://borderlands.com/en-US/. [Accessed: 22- Apr- 2021].

[34] D. Eberly, 3D game engine design. Amsterdam [etc.]: Elsevier, 2007.

[35] "Don't Starve | Klei Entertainment", Klei.com, 2021. [Online]. Available: https://www.klei.com/games/dont-starve. [Accessed: 22- Apr- 2021].

[36] T. Mahlmann, J. Togelius and G. Yannakakis, "Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types", Applications of Evolutionary Computation, pp. 93-102, 2011. Available: 10.1007/978-3-642-20525-5_10

[37] O. Korn, M. Blatz, A. Rees, J. Schaal, V. Schwind and D. Görlich, "Procedural Content Generation for Game Props? A Study on the Effects on User Experience", Computers in Entertainment, vol. 15, no. 2, pp. 1-15, 2017. Available: 10.1145/2974026

[38] R. de Pontes and H. Gomes, "Evolutionary Procedural Content Generation for an Endless Platform Game", 2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), 2020. Available: 10.1109/sbgames51465.2020.00021.

[39] "Minecraft", Minecraft.net, 2021. [Online]. Available: https://www.minecraft.net/en-us/. [Accessed: 22- Apr- 2021].

[40] S. Sorrell, "Reducing energy demand: A review of issues, challenges and approaches", Renewable and Sustainable Energy Reviews, vol. 47, pp. 74-82, 2015. Available: 10.1016/j.rser.2015.03.002

[41] A. Omer, "Energy use and environmental impacts: A general review", Journal of Renewable and Sustainable Energy, vol. 1, no. 5, p. 053101, 2009. Available: 10.1063/1.3220701

[42] I. Staffell and S. Pfenninger, "The increasing impact of weather on electricity supply and demand", Energy, vol. 145, pp. 65-78, 2018. Available: 10.1016/j.energy.2017.12.051

[43] J. Lieberei and S. Gheewala, "Resource depletion assessment of renewable electricity generation technologies—comparison of life cycle impact assessment methods with focus on mineral resources", The International Journal of Life Cycle Assessment, vol. 22, no. 2, pp. 185-198, 2016. Available: 10.1007/s11367-016-1152-3

[44] R. Clémençon, "The Two Sides of the Paris Climate Agreement", The Journal of Environment & Development, vol. 25, no. 1, pp. 3-24, 2016. Available: 10.1177/1070496516631362

[45] "The Paris Agreement", 2021. [Online]. Available: https://unfccc.int/process-and-meetings/the-paris-agreement/the-paris-agreement

[46] J. de Vries, Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. 2020.

[47] "Picking with an OpenGL hack", Opengl-tutorial.org, 2021. [Online]. Available: http://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/picking-with-an-opengl-hack/. [Accessed: 23- Apr- 2021].

[48] M. Hammad, I. Inayat and M. Zahid, "Risk Management in Agile Software Development: A Survey", 2019 International Conference on Frontiers of Information Technology (FIT), 2019. Available: 10.1109/fit47737.2019.00039 [Accessed 24 April 2021].

[49] S. Hatton, "Choosing the Right Prioritisation Method", 19th Australian Conference on Software Engineering (aswec 2008), 2008. Available: 10.1109/aswec.2008.4483241 [Accessed 24 April 2021].

[50] "DirectX graphics and gaming - Win32 apps", Docs.microsoft.com, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/directx. [Accessed: 28- Apr- 2021].

[51] "Khronos Vulkan Registry - The Khronos Group Inc", Khronos.org, 2021. [Online]. Available: https://www.khronos.org/registry/vulkan/. [Accessed: 28- Apr- 2021].

[52] "Metal - Apple Developer Documentation", Developer.apple.com, 2021. [Online]. Available: https://developer.apple.com/documentation/metal/. [Accessed: 28- Apr- 2021].

[53] "The OpenGL R Graphics System: A Specification (Version 4.6 (Core Profile) - October 22, 2019)", Khronos.org, 2021. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf. [Accessed: 28- Apr- 2021].

[54] M. Zamri and M. Sunar, "Atmospheric cloud modeling methods in computer graphics: A review, trends, taxonomy, and future directions", Journal of King Saud University - Computer and Information Sciences, 2020. Available: 10.1016/j.jksuci.2020.11.030

[55] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony Derose, George Drettakis, et al.. A Survey of Procedural Noise Functions. Computer Graphics Forum, Wiley, 2010, 29 (8), pp.2579-2600. 10.1111/j.1467-8659.2010.01827.x hal-00920177

[56] A. Ginting, K. Sari, C. Fadhilah, R. Yusra, D. Hartama and M. Zarlis, "Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game", Journal of Physics: Conference Series, vol. 1255, p. 012064, 2019. Available: 10.1088/1742-6596/1255/1/012064

[57] K. Perlin, "Improved Noise reference implementation", Mrl.cs.nyu.edu, 2021. [Online]. Available: https://mrl.cs.nyu.edu/~perlin/noise/. [Accessed: 28- Apr- 2021].

[58] L. Wei, "Parallel Poisson disk sampling", ACM SIGGRAPH 2008 papers on - SIGGRAPH '08, 2008. Available: 10.1145/1399504.1360619