OÉ Gaillimh
NUI Galway

Game World Creation using Procedural Generation Techniques

Ryan Clarke
4BCT Computer Science and Information Technology
National University Ireland, Galway
April 2018

Table of Contents

Illustration Index

List of Acronyms

PCG – Procedural Content Generation
GMS – Game Maker Studio
CTO – Chief Technology Officer
2D – Two-dimensional
3D – Three-dimensional
IDE – Integrated Development Environment
GLSL – OpenGL Shader Language
GML – GameMaker Language
RTS – Real-time Strategy
PRNG – Pseudo-random Number Generation
ECEF – Earth-centered, earth-fixed
RGBA – Red, Green, Blue, Alpha
FPS – Frames Per Second

Acknowledgements

I would like to thank my project supervisor David O'Sullivan. I would also like to extend my thanks to Finn Krewer of 9th Impact for the insight into the game development world.

Introduction

My project will be a focus on using procedural generation techniques and algorithms to produce the game world for the player to interact with. Procedural generation is becoming an exceedingly important topic for game developers all over the world, especially independent developers who do not have the resources to create high quality content within an appropriate time frame.

The target of this project is to develop a system that implements a game world algorithmically instead of manual creation and placement of objects into the game world.

It is important to acknowledge the vast amount of information on the internet pertaining the topic of PCG. With the development of engines accessible to the average developer without intricate knowledge of the underlying technical aspects of the engine, forums for discussion on implementations within that engine have been created. Other sites such as Stack Overflow[1], Wikipedia[2] and the Procedural Content Generation Wiki[3] can provide a detailed explanation or general overview of topics related to game development.

The video game industry has become one of the biggest industries in the world. In 2017, the worldwide market revenue of the video game industry (including media related to video games) was estimated to be worth $105 billion[4]. The advent of digital distribution platforms has made selling a game much simpler and cheaper[5]. This has led to a boom of ‘indie’ games created by small teams without a publisher.

From a personal perspective I have been playing games since I was young. I started playing with game development when I was twelve years old learning scripting language in game creation systems. The internet really helped provide the information necessary to accomplish the goals I wanted in very simple games.

Thus the motivation behind this project is creating games from an independent developer perspective. Where an independent game developer is lacking a publisher providing an extensive budget or a dedicated team for each aspect of the game.

The report will be laid out as follows:

Chapter 1 will be a brief overview of the research and game design involved in this project. There will also be discussion on the scope of this project.

Chapter 2 will be the implementation of the project.

Chapter 3 will be a discussion of the technical issues and analysis of the project.

Chapter 4 will discuss future work and give a conclusion for the project.

Preliminary Research

Upon meeting and discussing the project with my supervisor David O’Sullivan he referred me to talk with Finn Krewer, CTO of 9th Impact[6]. 9th Impact are a game development studio based in Galway. Primarily a mobile game developer who work with rights owners to develop official games based on TV shows. Their latest game *“Danger Mouse: The Danger Games”* based on the British animated television show *Danger Mouse*, is an award-winning multiplayer racing game[7].

I engaged with Finn via email and set up a meeting with him on campus. The focus on the meeting was to investigate on a game related topic I wanted to base my project on. In the meeting Finn took me through their Danger Mouse game. In the game the player can play against players from around the world; racing characters from the TV show. The player can build a ‘deck’ to use against other racers to take the lead. One of the interesting things from the game I noticed was the map. The games racing mechanic is very similar to the old mobile game Temple Run, where the player has to dodge incoming obstacles. I asked Finn how the map and obstacles are generated. He told me that the map was made of set pieces stitched together and using strategic turns on the race track to give the illusion of a defined map. Then the obstacles are randomly placed on these set pieces. Since mobile devices often have limited memory and computing performance this set piece design allows for the game to load in these objects on demand without placing everything within memory.

This meeting was very helpful in deciding my project. I wanted to make a game that generates a map algorithmically without any placement by the developer. In terms of using set pieces to create the game world there is the popular game series *Diablo* which contains maps made of set pieces that connect together and on these maps are random quest gives and chests. For complete terrain generation an example is *Minecraft*. *Minecraft* builds its map using a noise generator and applying a user inputted seed. For this project I went with the latter due to the former requiring a lot of work creating the set pieces. Unless the design of the connections to each set piece is simplistic (for example a door) then many variations will need to be created for each connection.

Minecraft is an example of 3D procedural generation. For 2D map generation there are games such as the *Age of Empires* series. It creates its maps randomly for every new match played.

Chapter 1 (Game Design)

Game Design

This section will outline the overall game design of the project and the parts that are within the scope of the project. The games theme and genre help to define the map that the game needs.

Game Genre

The genre for the result of this project will be the real-time strategy genre. This genre has some defining characteristics, most notably is the ability to maneuver their units to enforce control over parts of the map. The maps usually contain specific points to attack or defend resources and build. The games graphics can either be in the 2D or 3D format and could possibly have a top-down or 3/4 view (Illustration 1.).

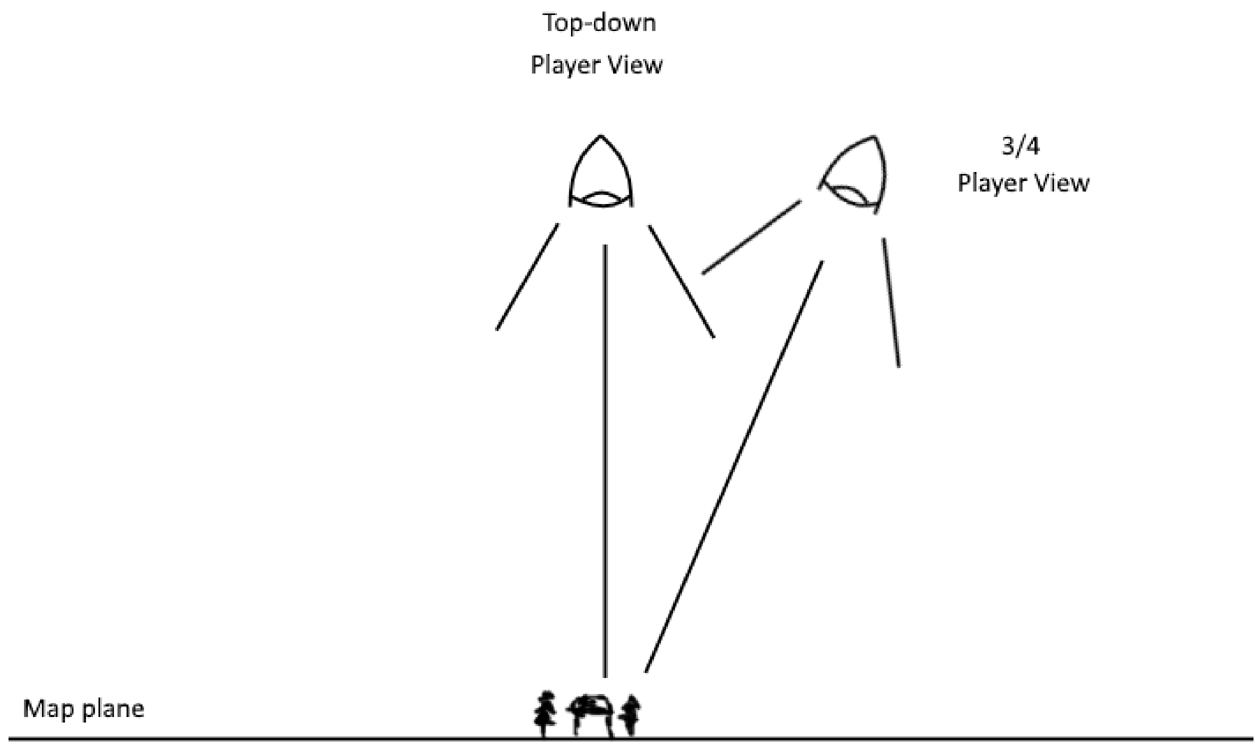
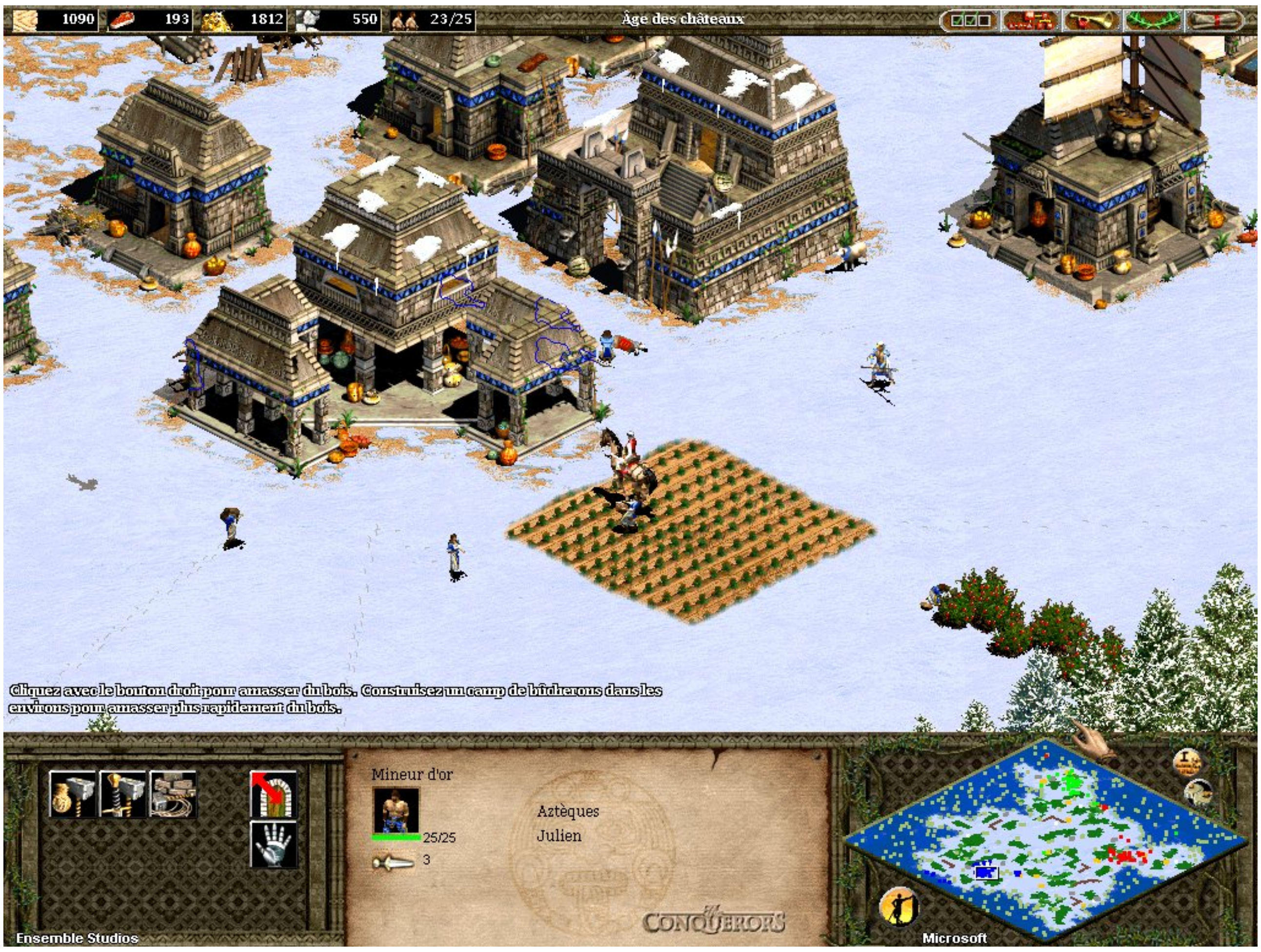


Illustration 1: Top-down/Three-quarter view

These views give the player an overwatch position on the actions within the game.
In 2D games using the 3/4 view the sprites and images used within the game are drawn with a skewed perspective giving the illusion of a 3/4 view (example: Illustration 2).



Cliquez avec le bouton droit pour amasser du bois. Construisez un camp de bûcherons dans les environs pour amasser plus rapidement du bois.

Illustration 2: Age of Empires II 3/4 isometric perspective

The maps within these games are often randomly generated with specified parameters based on the users choice. The randomness to the maps gives players a new but similar experience every time.

I am a fan of this genre and played many games within it. I've always wanted to try developing my own real-time strategy game and this is the main inspiration for this project. Some of my favorite games within the genre are the *Age of Empires* series, the *Starcraft* series and *Company of Heroes*.

Knowing the game genre is important for the next step which is picking the engine. Some game engines are designed for specific genres in mind.

Game Engine

With the rise of ‘indie’ game development there has been a steady increase in low-cost, free and open-source game engines. Picking a game engine relevant to a game can be difficult due to the extensive amount of options. There’s also a learning curve to each engine as many have their own IDE, scripting language and core programming language. Some of the engines I looked at are the following:

Unreal Engine
- 2D/3D Engine
- C++, older version can use UnrealScript (< Unreal Engine 4)
- Free to use, commercial has a royalty system
- 5% royalty on gross revenue after $3000 per game per calender year[8]
- Continually developed by Epic Games

The Unreal Engine has become quite popular in recent years with independent developers due to it’s move from a subscription-based model to a royalty based model. Some notes to make are it’s primarily a 3D engine with 2D implemented via the Paper2D system.

Unity
- 2D/3D Engine
- C#, older versions have access to UnityScript (JavaScript variant)
- Free to use, subscription based commercial model
- Free up to $100,000 annual gross revenue otherwise monthly subscription[9]
- Continually developed by Unity Technologies

Probably the strongest competitor to the Unreal Engine. Well known for its ease of use between independent developers.

Godot
- 2D/3D Engine
- C#, C++ and its own scripting language GDScript
- Free, under MIT License
- Open-source, community driven development[10]

Completely open-source and supports multiple target platforms. Has its own IDE and has strong documentation to back it.

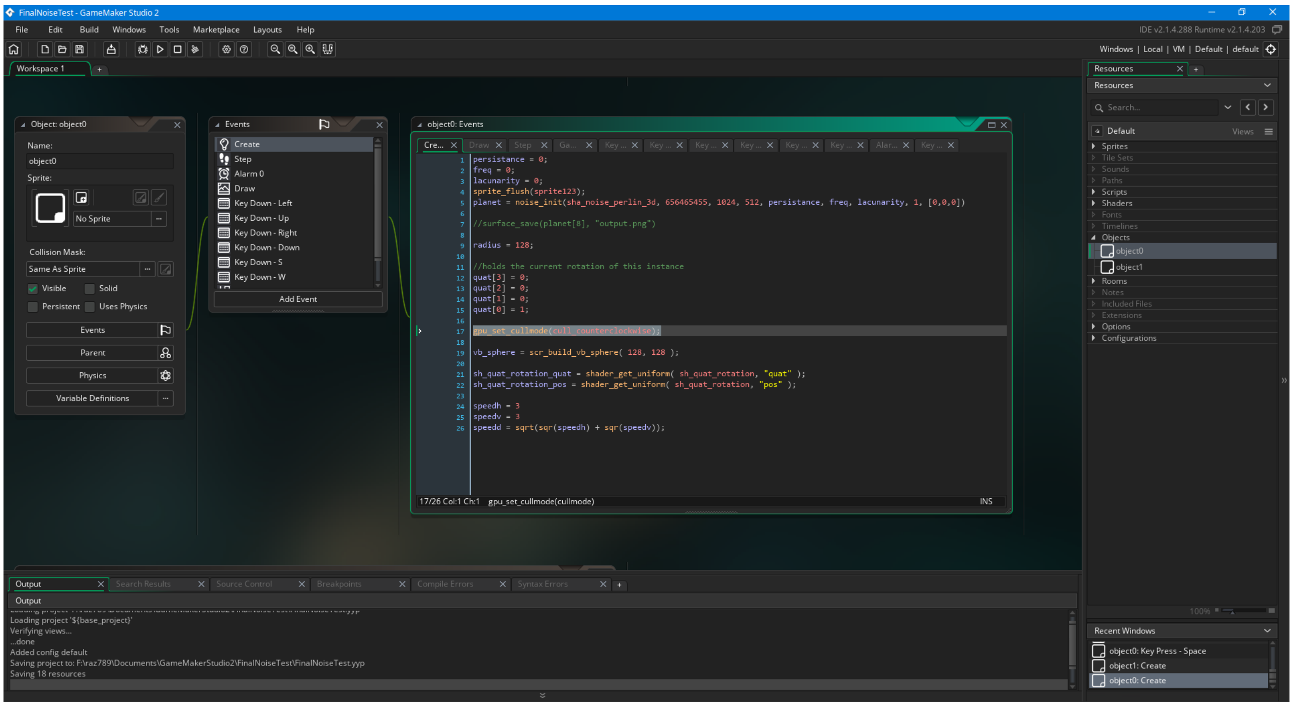
GameMaker: Studio
- 2D/3D Engine
- Uses its own scripting language GameMaker Language or drag and drop visual programming
- One-time license fee for each target platform, desktop $99[11]
- Continually developed by YoYoGames

Limited 3D functions. I have experience with this engine and have used it in the past to make small games. It’s programming language is very easy and its built-in functions are well documented and descriptive.

Each of these engines have a great community where you can ask for help.

For my game I wanted to stick to 2D. This is simpler to work with and reduces the complexity in generating maps. Objects within the game would only need to apply the X-axis and Y-axis and there would be minimal camera work. Also because of the time needed to be spent learning a new engine I wanted to go with sometime I have had previous experience with. For this project I will be using GameMaker: Studio. I had previously purchased a license for this project in hopes of exploring game development as a hobby. For the purposes of this project I did not be using any of GMSs built-in drag and drop visual scripting feature. This project has been developed using GML and the OpenGL Shader Language.

GMS is usable by complete beginners and advanced users for rapidly developing full or prototype games.



```
persistance = 0;
freq = 0;
lacunarity = 0;
sprite_flush(sprite123);
planet = noise_init(sha_noise_perlin_3d, 656465455, 1024, 512, persistance, freq, lacunarity, 1, [0,0,0])

//surface_save(planet[8], "output.png")

radius = 128;

//holds the current rotation of this instance
quat[3] = 0;
quat[2] = 0;
quat[1] = 0;
quat[0] = 1;

gpu_set_cullmode(cull_counterclockwise);

vb_sphere = scr_build_vb_sphere( 128, 128 );

sh_quat_rotation_quat = shader_get_uniform( sh_quat_rotation, "quat" );
sh_quat_rotation_pos = shader_get_uniform( sh_quat_rotation, "pos" );

speedh = 3
speedv = 3
speedd = sqrt(sqr(speedh) + sqr(speedv));
```

```
17/26 Col:1 Ch:1    gpu_set_cullmode(cullmode)    INS
```

```
Loading project '${base_project}'
Verifying views...
...done
Added config default
Saving project to: F:\raz789\Documents\GameMakerStudio2\FinalNoiseTest\FinalNoiseTest.yyp
Saving 18 resources
```

Illustration 3: GameMaker: Studio 2 IDE

The IDE contains features expected of a modern game engine IDE (Illustration 3). On the right we have the resources list. On the bottom is the output providing information on errors. Centered is our workspace where objects and code can be created and modified. Games can be run via a virtual machine which converts the code to bytecode or compiled. The IDE also contains a debugger that allows you to view the game variables in real-time.

For the rest of this project I will be making reference to events within GMS. GMS objects contain events that are executed in a certain order. Not all events have to be active.
The three most important events are as follows:

The Create Event: run the moment an instance of an object is created.
The Step Event: run every step where one there is an equal number of steps per second based on the frame rate of the game. E.g. 60 frames per second will have 60 steps per second.
The Draw Event: Same as the step event except that it is the only event that can draw to the screen.

Any other events are variations of the above.

Game Theme and Setting

The implementation of the game mechanics is beyond the scope of this project but it is important as it decides what the maps need to contain.

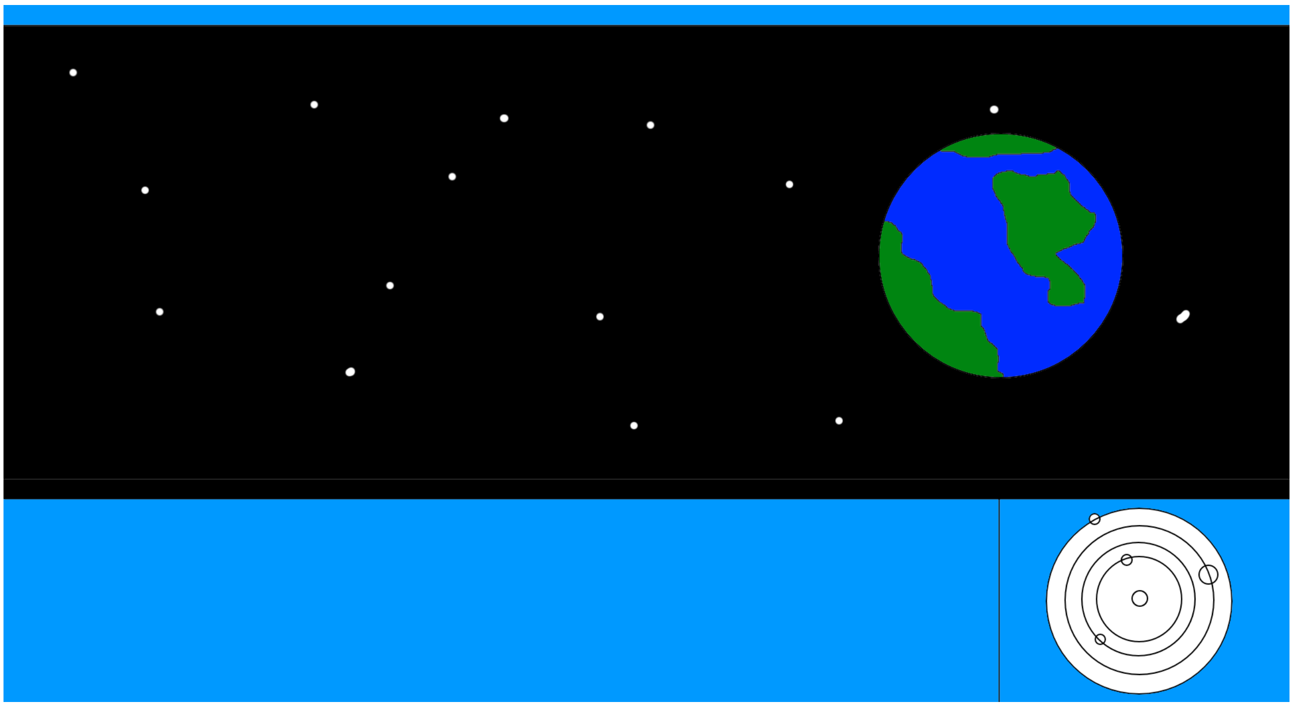


Illustration 4: Early idea on finished product

The finished product I imagine would have each player contained to a solar system. This solar system would contain a number of planets and each planet would contain a number of resources. Players would control some or all of the system they're in and can build structures on their planets to create units to use in battle with another player.

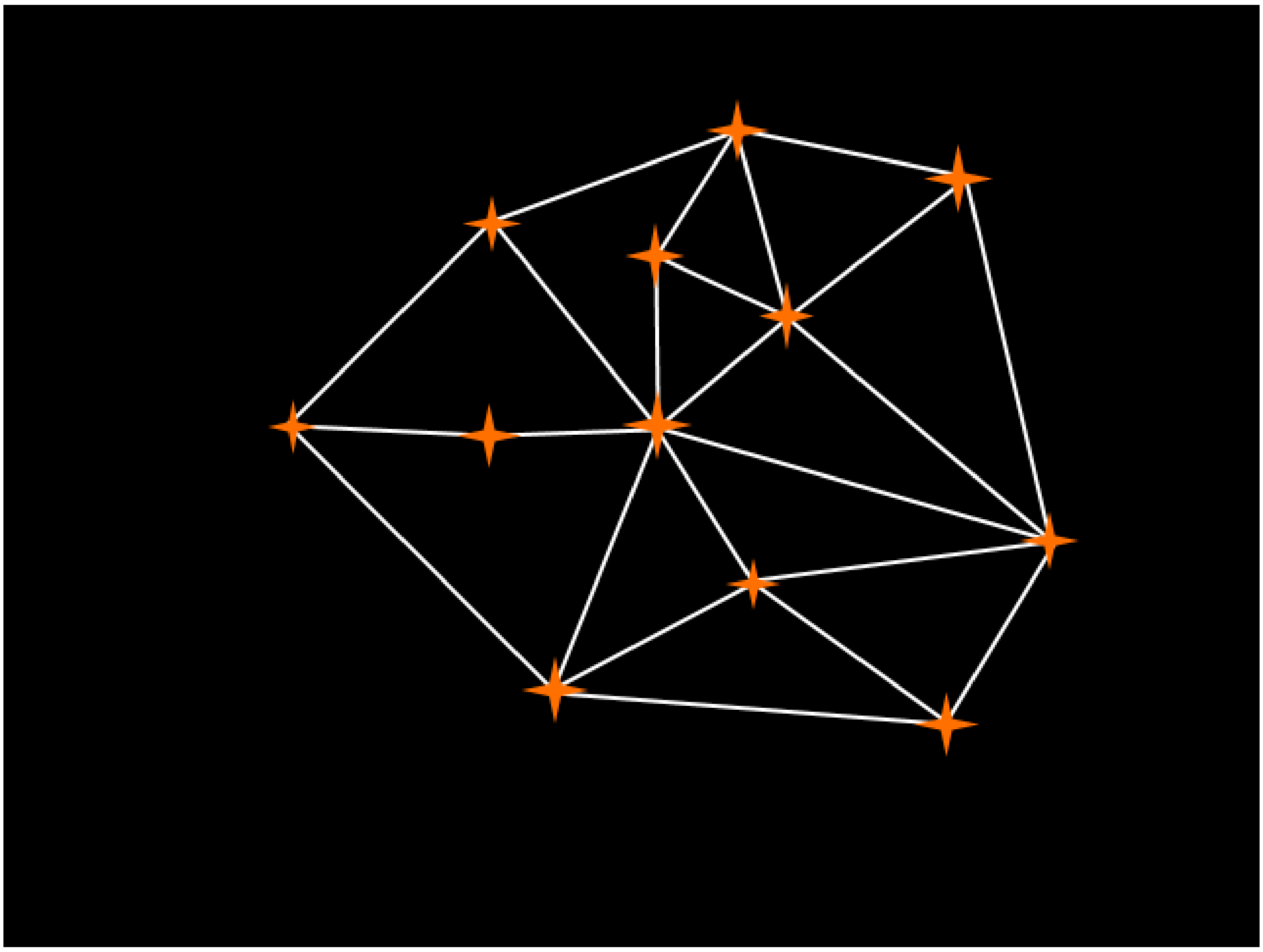


Illustration 5: Galaxy star system and its connections

The base game idea is that players fight for control of the galaxy. Galaxy connections (Illustration 5) would be facilitated by a ‘gate’ within the system which allows the player to transfer their units to another system.

With this I wanted to create a random map generator that creates a small galaxy with a number of solar systems and their connections.

Requirements

The requirements of this project are for a developer adding onto the project.

Seed Support

The creation of the map needs to use a ‘seed’. This seed will allow the map to be reproduced on consecutive runs. For the purpose of demoing this project the seed should be able to be inputted by the user before the map is created.

Galaxy – Star System
The map of the galaxy should have a definable number of points or systems with connections to each. The map should be drawn to the screen. The connections and their angle should be passed to the solar system.

Solar System
The solar system should have a definable middle point where the sun it placed. Orbiting this point are a random number of planets at different random speeds. Past this should be the gate connections placed depending on their angle of the connection.

Planets
The planets need to rotate on their own axis. Graphically the planets should be visually distinct. These are the main hub for the player to create units and structures.

Gates
The gates on the system edges should allow the user to navigate to another system. Need to be placed intelligently so that navigating to them is easy.

Chapter 2 (Implementation)

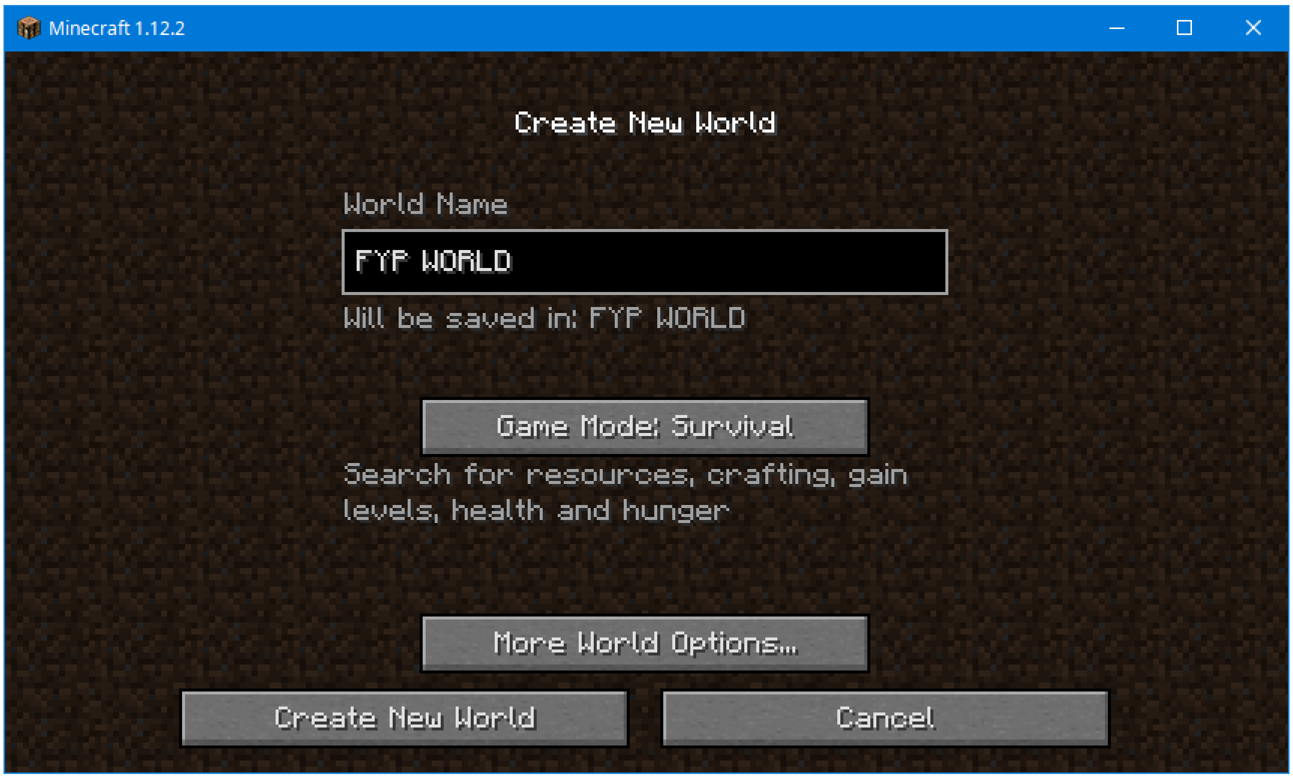
Implementation

This section will be a focus on the implementation of the defined requirements in Chapter 1. Each part of this chapter will discuss the research on that specific topic and then how it was implemented in my project.

Seed support

In game development there is extensive use of random numbers to produce random results. Although generally these numbers aren’t truly random but rather pseudo-random. That is, when the random numbers outputted follow a specific sequence of a specific length. They are not truly random but based on an initial state or value called a seed. This deterministic sequence is important for producing procedural generated maps. The reproducibility allows maps to be shared as was the case in the 1984 video game Elite where due to memory limitations at the time the game was supplied with a seed number and then the game content was procedurally generated[12]. This meant that any player who purchased a copy of the game would have the same procedurally generated world based on this seed. A modern example of a game shipped with a single seed is No Man’s Sky.

In more modern games the use of seeds and procedural generation for creating whole maps is declining due to the increasing memory and computational power of modern computers. Although this does not mean it’s completely disappeared in modern video games. One of the most popular video games in the last decade Minecraft, makes use of seeds and PRNG to generate it’s terrain.


Minecraft 1.12.2

Create New World

World Name

FYP WORLD

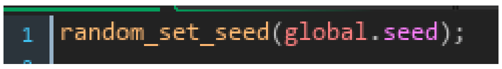
Will be saved in: FYP WORLD

Search for resources, crafting, gain levels, health and hunger

Illustration 6: Minecraft seed input

Upon the selection of creating a new world the user is prompted with the choice to enter a seed of their choosing (Illustration 6). This seed can be a hexadecimal character. The seed is then converted to an integer using Java's built-in hashCode() function. This function returns an integer than can the be used for comparisons. In the case of Minecraft this integer is used for the initial state of the random functions. Although this is not without it's issues, seed collision is possible. This is where even though two separate strings were entered by the player the generated world is the same.

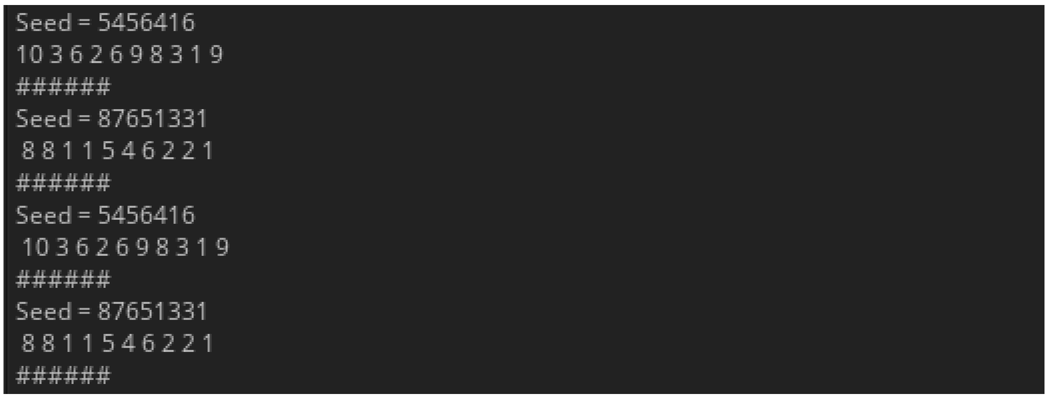
For my project I used GMS built-in random functions. These functions can be seeded manually and randomly by GMS (Illustration 7).



```
random_set_seed(global.seed);
```

Illustration 7: Setting the seed in GML

A quick test in GMS gives the following result when switching between seeds and switching back(Illustration 8).



```
Seed = 5456416
10 3 6 2 6 9 8 3 1 9
######
Seed = 87651331
 8 8 1 1 5 4 6 2 2 1
######
Seed = 5456416
 10 3 6 2 6 9 8 3 1 9
######
Seed = 87651331
 8 8 1 1 5 4 6 2 2 1
######
```

Illustration 8: PRNG

As can be seen the resulting numbers produce the same result given an identical input seed.

GMS PRNG is suitable for this project due to the fact I only need one PRNG generator. If you need more than one you would need to implement your own PRNG generator. There are a few good algorithms out there that have been well tested. One of the most popular is the Mersenne Twister. Implementing your own PRNG has downsides especially in GMS. GMS projects are single-threaded and follow a flow of execution and freeze in this executions will cause a complete slowdown of the game. So if I was implementing my own I'd make sure it's fast and not necessarily of high quality. GMS PRNG is not cryptographically secure thus if this is something that is needed for a game a developer would have to one themselves.

GMS does not have a built-in hashCode function so I implemented my own method of converting a string to an integer. Initially I tried using the ASCII values of the characters but this had a glaring issue. Seed collision was common in this method.

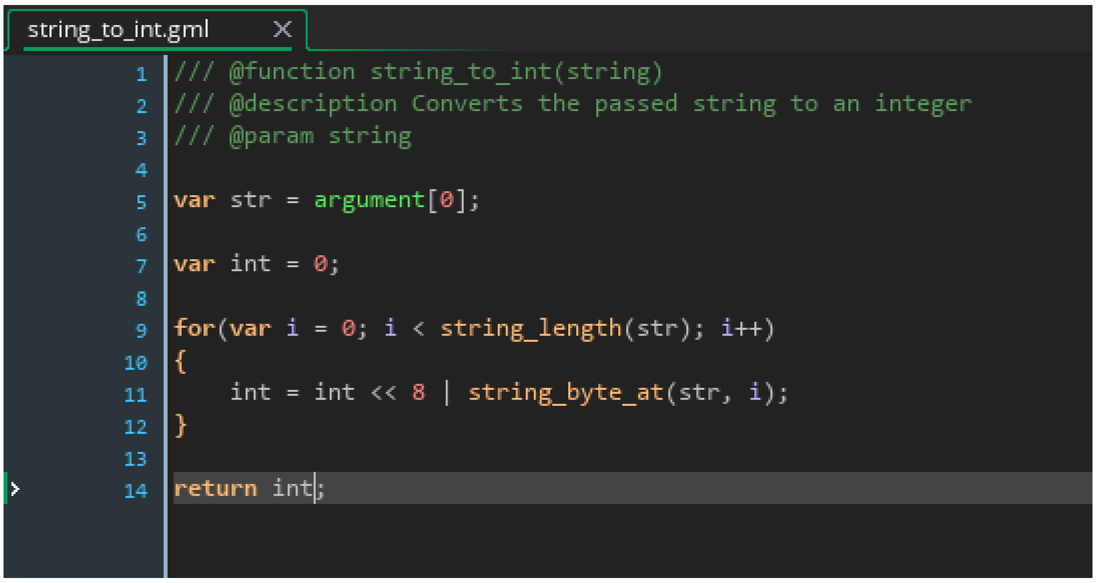
For example the string "hello":

h – 104	e – 101	l – 108	l – 108	o - 111

Adding the numbers together: 532

and the string "kbjno" add up to the same value. Even jumbling the word hello returns the same result.

To prevent this I used a different method of bit-shifting the binary representation of each character into a 64-bit integer.



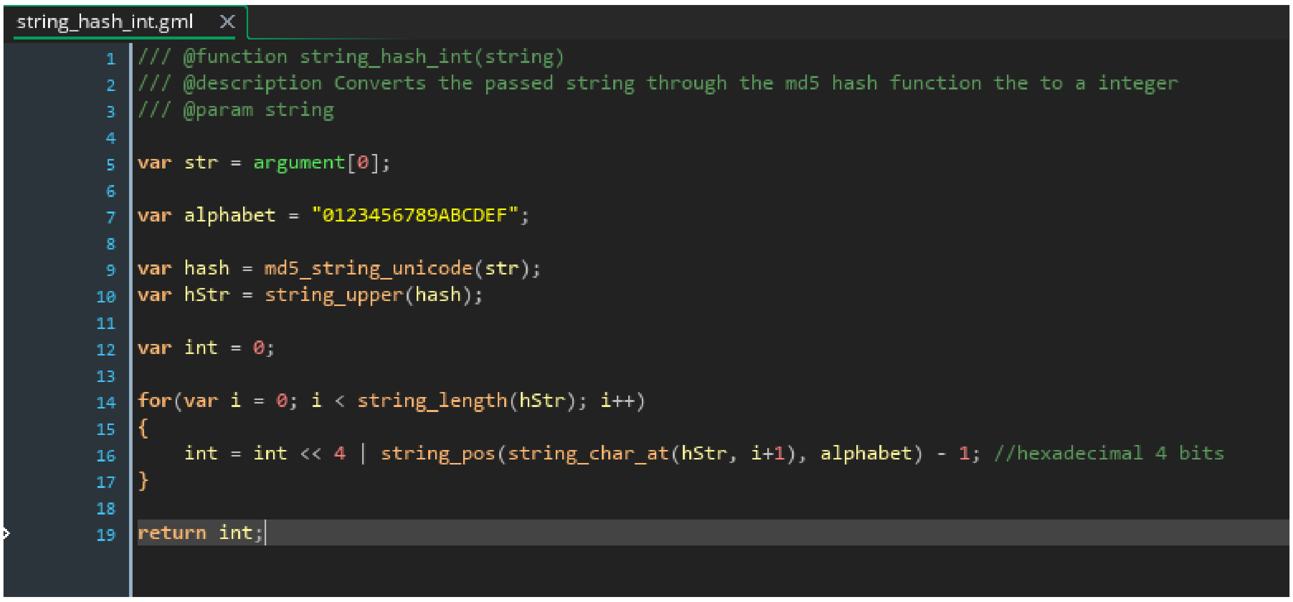
string_to_int.gml

```
/// @function string_to_int(string)
/// @description Converts the passed string to an integer
/// @param string

var str = argument[0];

var int = 0;

for(var i = 0; i < string_length(str); i++)
{
    int = int << 8 | string_byte_at(str, i);
}

return int;
```

Illustration 9: Bit shifting the string into the an integer

The code above(Illustration 9) simply returns the byte at a specific character position and shifts it into the variable call “int”. There is only one problem with this method. The string is limited to 8 characters otherwise there is a loss of information and the chance of string collision.

Although for the purposes of this project 8 characters is more than enough to get a suitable seed integer I wanted to try another method.



string_hash_int.gml

```
/// @function string_hash_int(string)
/// @description Converts the passed string through the md5 hash function the to a integer
/// @param string

var str = argument[0];

var alphabet = "0123456789ABCDEF";

var hash = md5_string_unicode(str);
var hStr = string_upper(hash);

var int = 0;

for(var i = 0; i < string_length(hStr); i++)
{
    int = int << 4 | string_pos(string_char_at(hStr, i+1), alphabet) - 1; //hexadecimal 4 bits
}

return int;
```

Illustration 10: md5 Hashing

GML comes with a pre-implemented hashing function. This method(Illustration 10) uses the md5 hashing algorithm to produce a hexadecimal string. This string is then converted to an integer by once again bit shifting. Since we're using hexadecimal format one hexadecimal character is only 4 bits. This gives us twice the string size without information loss(on the md5 output). But there's still a chance of seed collision as the md5 hashing function is well known to eventually encounter one.

With the former method the original string is kept intact within the 64-bit integer while with latter method we completely lose the original string. Md5 hashes are not reversible. The solution to this is to just simply save the seed as a string separately to its integer counterpart.

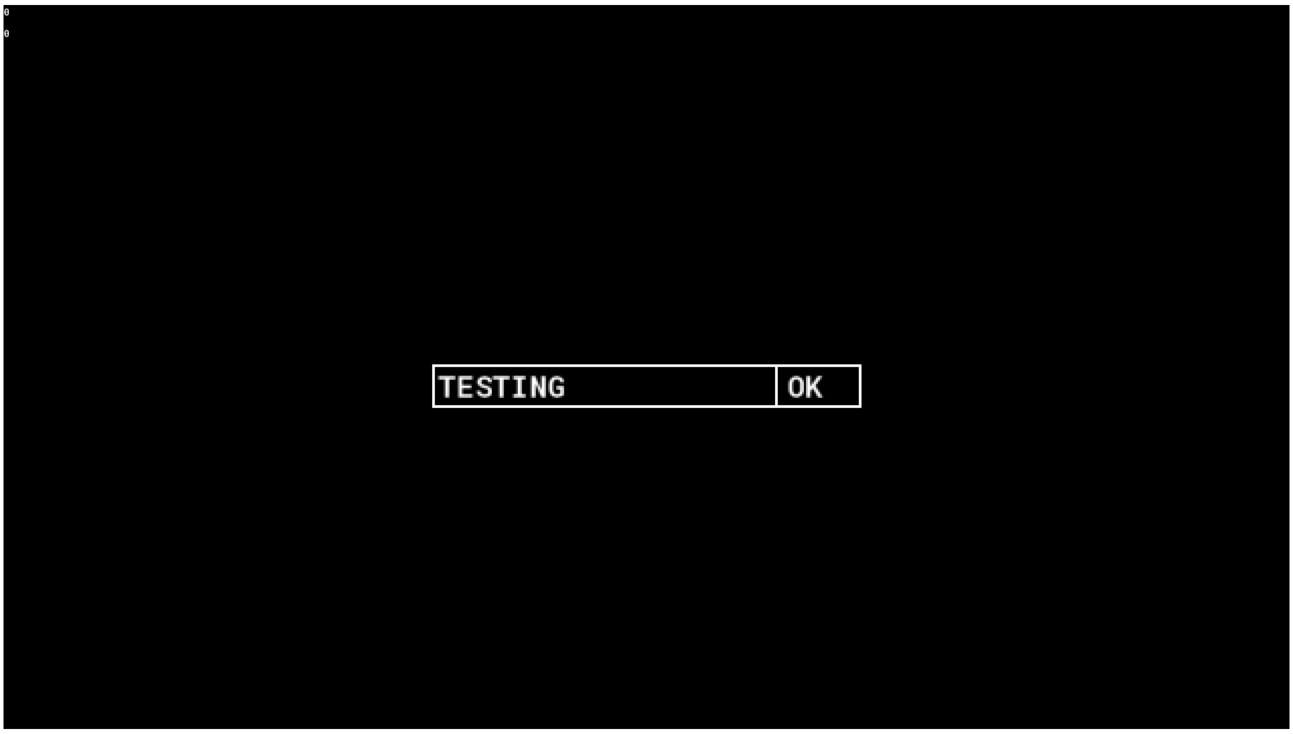


Illustration 11: String Input

For this project I added a text box upon launching the project for the user to input a seed. The string is capped to 8 characters. This lets me choose between both the first and second method outlined above.

Galaxy System

For the galaxy system I needed two separate algorithms. The first algorithm to place points randomly on a defined surface. The second will connect those points.

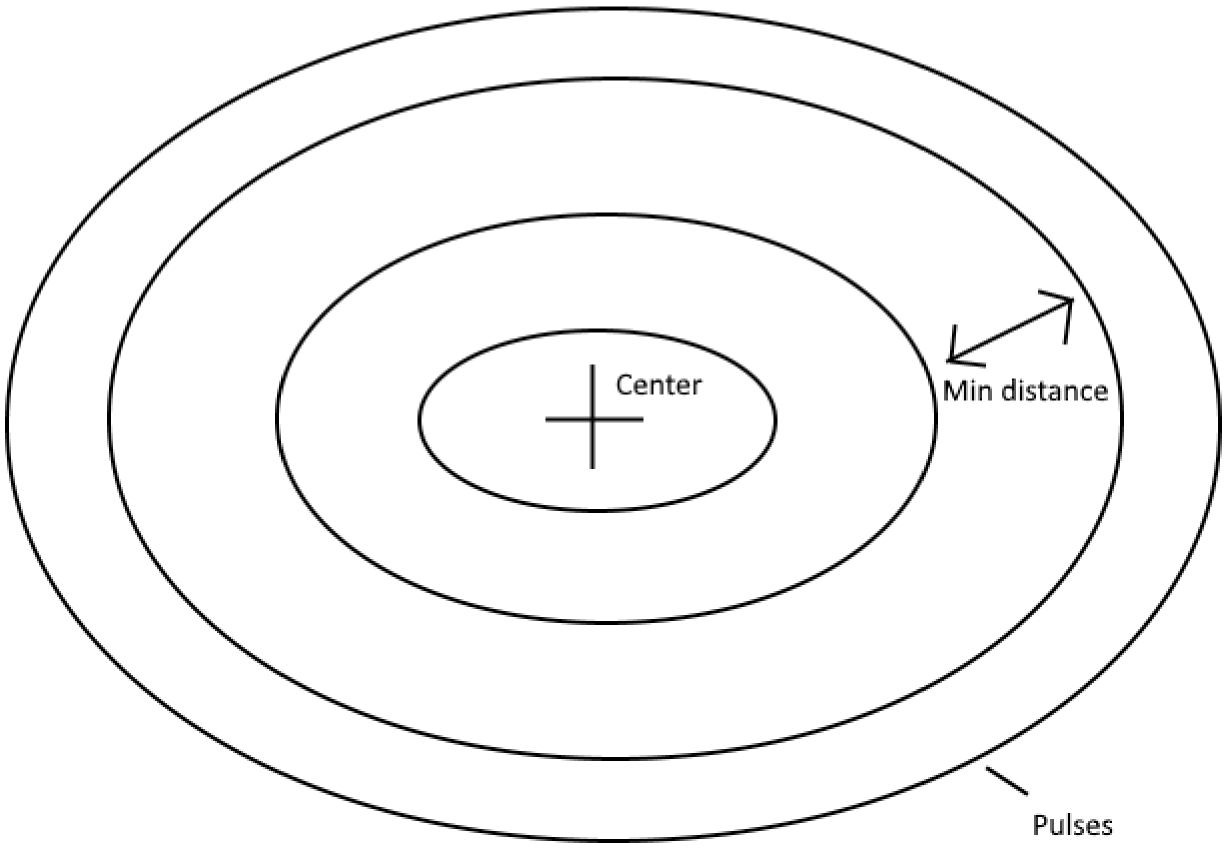


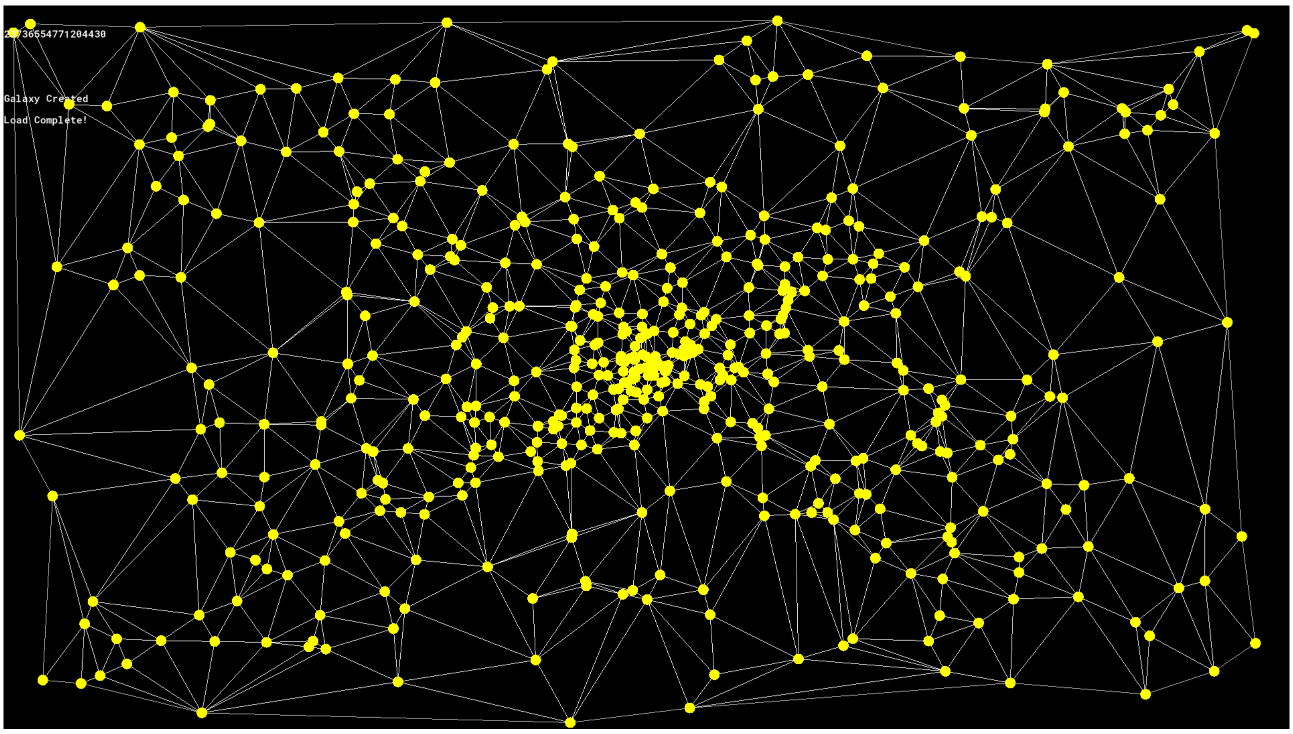
Illustration 12: Galaxy Creation

I implemented the galaxy populate algorithm by pulsing out multiple ellipses from a center point. On creation of an ellipse a random point on it is created. There is then a check to make sure the point is not within the radius of another.

This code is scalable (as can be seen in Illustration 13) as the minimum distance is using the following formula:

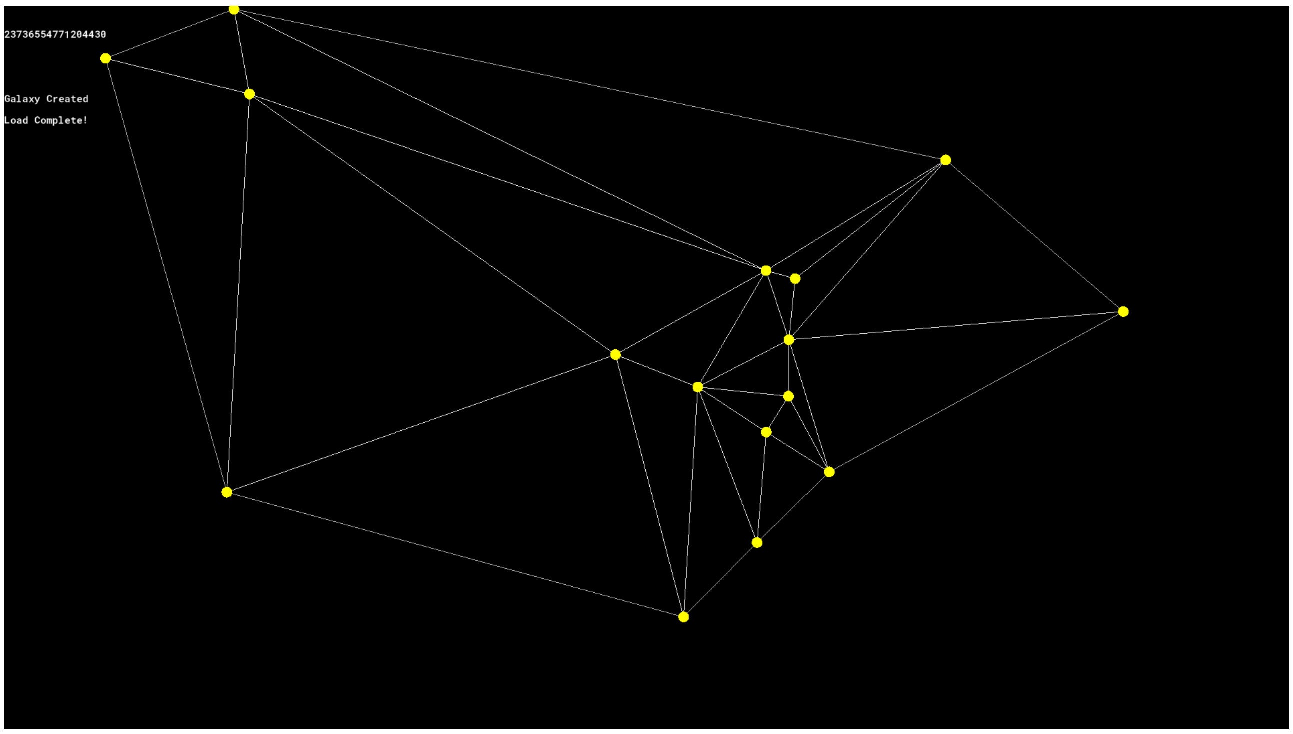
$$\frac{capacity}{2^{\frac{capacity}{32}-1}}$$

This formula will scale depending on the capacity passed into the script.
This resulting map has a distribution similar to a Gaussian distribution. The majority of the points are centered.



```
23736554771204430

Galaxy Created
Load Complete!
```

Illustration 13: 512 points in the galaxy



```
23736554771204430

Galaxy Created
Load Complete!
```

Illustration 14: 16 points in the galaxy

Finally I used a method of connecting the points called Delaunay triangulation[13]. The idea behind this algorithm is that given a set of points it should be possible to connect this points into triangles such that no circumcircle of any triangle contains any other points other than the ones in its own triangle (Illustration 15). This algorithm is usually used to create efficient meshes in generated terrain. But for my purposes it works particularly well at connecting my star systems.

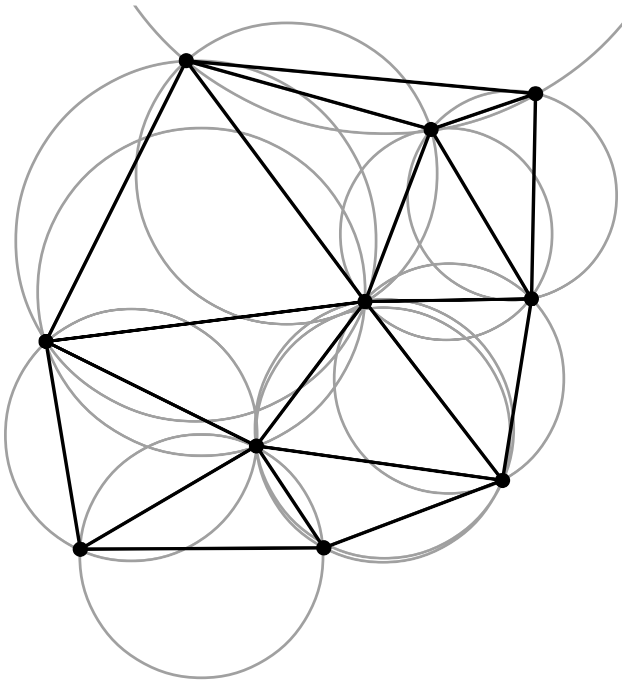


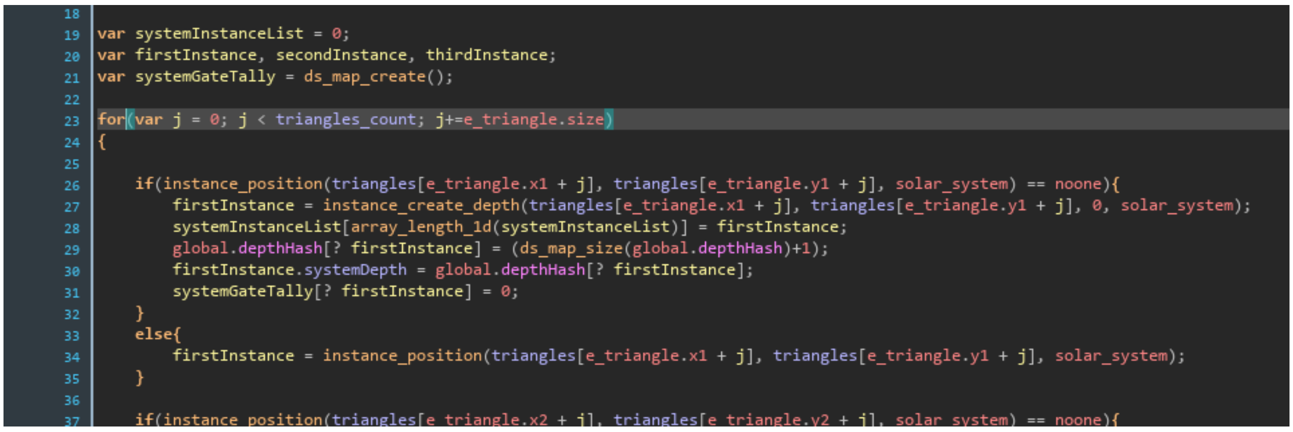
Illustration 15: Delauney Triangulation

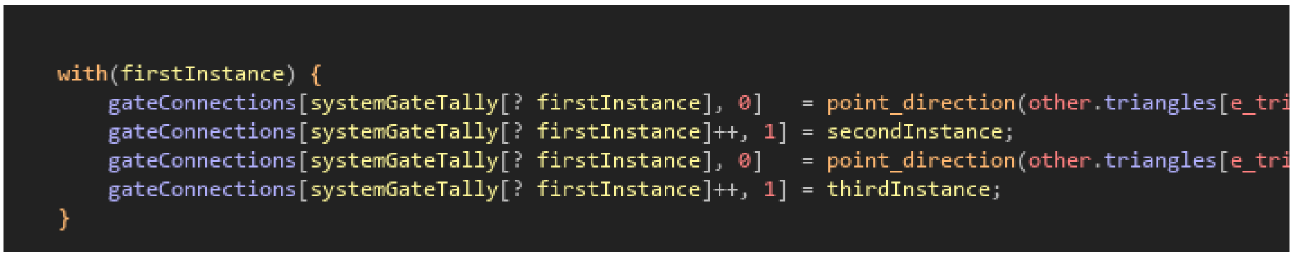
For this project I used an implementation created by user jujuadams on github created specifically for GMS.

https://github.com/GameMakerDiscord/delaunay

This library is distributed under the permissive MIT license.

This library only needs to be passed a list of points (from the first algorithm in this section) and returns a list of triangles based on those points. These triangles are then converted into useful data for my solar system and its connections(Illustration 16 and 17).


```

var systemInstanceList = 0;
var firstInstance, secondInstance, thirdInstance;
var systemGateTally = ds_map_create();

for(var j = 0; j < triangles_count; j+=e_triangle.size)
{

    if(instance_position(triangles[e_triangle.x1 + j], triangles[e_triangle.y1 + j], solar_system) == noone){
        firstInstance = instance_create_depth(triangles[e_triangle.x1 + j], triangles[e_triangle.y1 + j], 0, solar_system);
        systemInstanceList[array_length_1d(systemInstanceList)] = firstInstance;
        global.depthHash[? firstInstance] = (ds_map_size(global.depthHash)+1);
        firstInstance.systemDepth = global.depthHash[? firstInstance];
        systemGateTally[? firstInstance] = 0;
    }
    else{
        firstInstance = instance_position(triangles[e_triangle.x1 + j], triangles[e_triangle.y1 + j], solar_system);
    }

    if(instance_position(triangles[e_triangle.x2 + j], triangles[e_triangle.y2 + j], solar_system) == noone){
```
Illustration 16: Creating an instance of a solar system based on the triangle point



```
with(firstInstance) {
    gateConnections[systemGateTally[? firstInstance], 0]   = point_direction(other.triangles[e_tri
    gateConnections[systemGateTally[? firstInstance]++, 1] = secondInstance;
    gateConnections[systemGateTally[? firstInstance], 0]   = point_direction(other.triangles[e_tri
    gateConnections[systemGateTally[? firstInstance]++, 1] = thirdInstance;
}
```

Illustration 17: Passing the direction and system of the connection to the instance

A hash map is used as a tally to prevent duplicate solar systems.

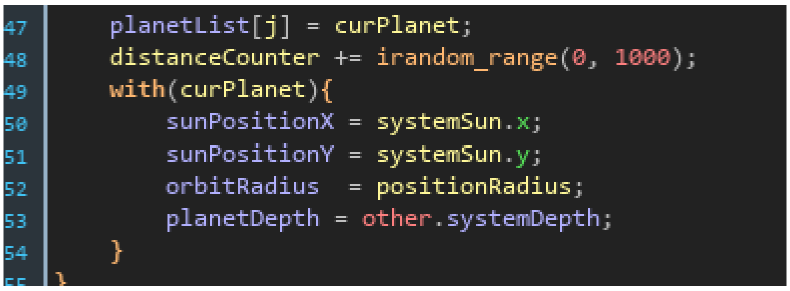
The solar system requires a the following data:

Any connections to other systems in the galaxy and direction(in degrees).
Assigned its own depth so that it will only be drawn if the player is currently in the current system.

Now that the galaxy has been created and each solar system has been instanced and given the information about the galaxy it can now begin populating its system with planets.

Solar System

The solar system defines the center of the game room as the sun position. In my case the room is 12000*12000px Next the solar system will randomly choose the number of planets orbiting the sun.



```
    planetList[j] = curPlanet;
    distanceCounter += irandom_range(0, 1000);
    with(curPlanet){
        sunPositionX = systemSun.x;
        sunPositionY = systemSun.y;
        orbitRadius  = positionRadius;
        planetDepth = other.systemDepth;
    }
```

Illustration 18: Passing properties to the planet

To do this I again used to algorithm that was used to populate the galaxy map. Instead of ellipses it pulses out in perfect spheres and checks the radius. Since the planet orbits I made sure that planets are placed on one axis to get a proper radius check.

Finally passing the properties of the solar system to the planet. The planet needs to know the position of the sun, it's orbiting radius and the depth of the solar system(for drawing purposes).

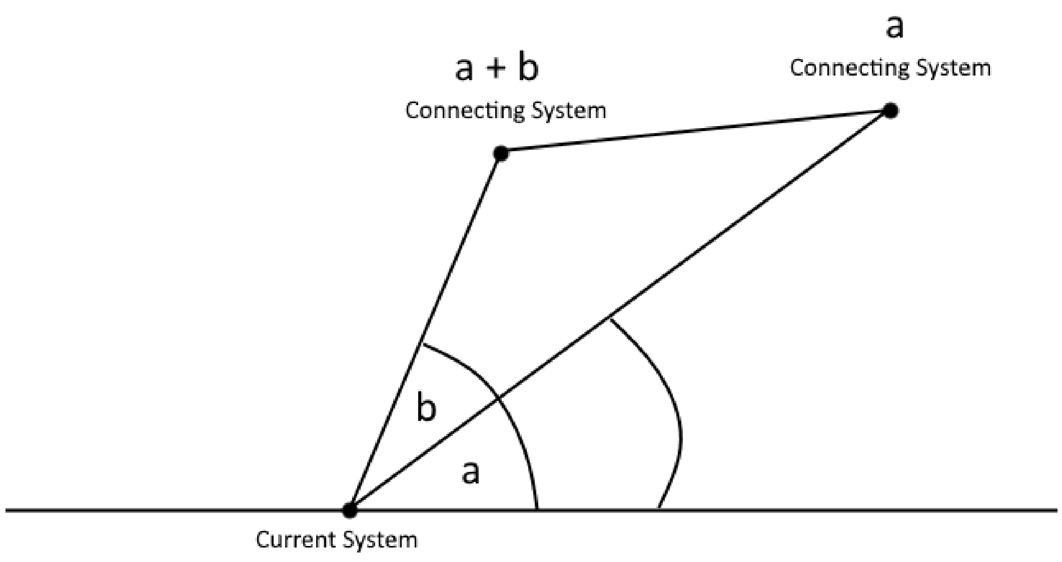


Illustration 19: The angle the line creates on the galaxy map

Using GMS point_direction() method(returning the degrees) that was passed into the solar system as a list of connections I was able to place the gate connections on the outer edges of the solar system outside of all the planets. Again we use a circle and place the gate on the circle based on its degrees. Once a player interacts with this gate they will be transported to the corresponding solar system. This will simply change the current global depth variable and thus a different solar system will be drawn.

The reason I’m using a depth variable to draw my objects is because of how rooms work in GMS. In GMS rooms are the area that objects can interact in. Each room is a completely separate instance and it is not possible to have two rooms running concurrently. Therefore my solution to keeping everything running (as this is a real-time strategy game) was to simply keep processing in the background. The foreground would only consist of the current solar system the player is present in.

Planets

Planets are the most important part of this project. They are will be the biggest interaction between the player and the game map. Given this I need to make sure the planets are visually distinct. To do so I used a gradient noise generator.

Initially I did some research on various noise generators.

Classic Perlin Noise

Developed in 1983 by Ken Perlin. Created to increase the realism of textures in computer graphics. This is the first known gradient noise function. Can be used for any N-dimension.

Not patented.

Simplex Noise

Developed in 2001 by Ken Perlin. Was created to correct some of the limitations of Classic Perlin Noise. In particular reduced computational complexity and reduction in the directional artifacts.

Patented. Details on the patent can be found:

https://patents.google.com/patent/US6867776

Can be used for 2-dimensions and below without infringement.

OpenSimplex

Developed by Kurt Spencer to get around the Simplex Noise patent. Performance is said to be worse than Simplex Noise.

Information on how it avoids patent infringement can be found summed up on the Wikipedia page: https://en.wikipedia.org/wiki/OpenSimplex_noise

Cubic Noise

An interesting alternative gradient noise function developed by Job Talle.

His algorithm can be found at the following github repository:

https://github.com/jobtalle/CubicNoise

His algorithm is released under public domain.

I wanted to avoid simplex noise altogether as I did not want to infringe on the patent it is under. I tried OpenSimplex and Cubic Noise within GMS but ran into a technical issue. The performance within GMS in its native scripting language is not compatible with these noise generators. Upon executing the generations there would be an immediate freeze of 5-10 seconds depending on the size of the noise texture.
Instead I decided to try to run these gradient noise algorithms on a shader. GMS has support for OpenGL ES 2.0. Which allows the execution of shader programs to apply effects to GMS graphical capabilities. These shaders programs run on the GPU which can speed up processing by thousands of times due to the sheer difference in cores between the GPU and CPU.

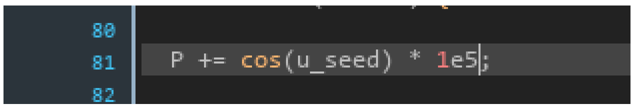
But I ran into another problem, OpenSimplex and Cubic Noise are not suitable for porting to GLSL. This will be discussed within the next chapter. So I decided to settle to using Perlin Noise.

Probably one of the best sources of information for Perlin Noise is Stefan Gustavson paper entitled “Simplex noise demystified”[14]. In it Stefan explains both Simplex and Perlin noise and gives a detailed code example of both. Stefan also has a github containing the code under the MIT License.

https://github.com/stegu/webgl-noise

Fortunately this code is compatible with the OpenGL standard that GMS uses. Implementing my own version would take a considerable amount of time. Since the classic noise portion of this code is distributed under the permissive MIT license I used it in this project with some modifications.

I added seed support to the code by offsetting the coordinates passed into the noise function.

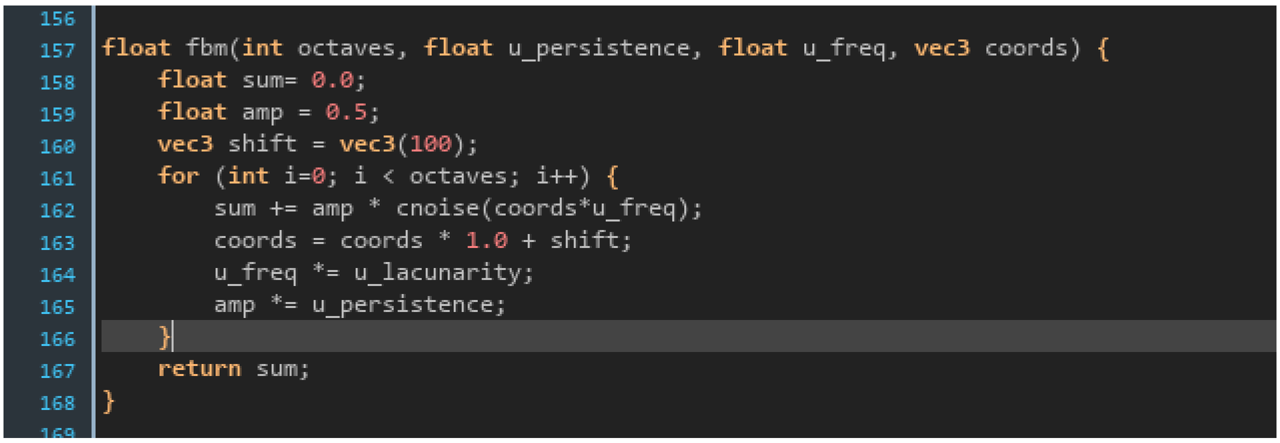


```
P += cos(u_seed) * 1e5;
```

Illustration 20: Seed support for Perlin noise using offsetting

Since Perlin noise is pseudo-random it is deterministic. All of it’s points are defined thus by offsetting the coordinates passed into it we should get a different but reproducible result.

Another change is the addition of Fractal Brownian Motion[15]. In addition in the shader I have added Fractal Brownian Motion. This where multiple steps(octaves) of a noise function with different frequencies(modified each step by the lacunarity) and amplitude(modified each step by persistence) are summed together to produce finer detail.



```
float fbm(int octaves, float u_persistence, float u_freq, vec3 coords) {
    float sum= 0.0;
    float amp = 0.5;
    vec3 shift = vec3(100);
    for (int i=0; i < octaves; i++) {
        sum += amp * cnoise(coords*u_freq);
        coords = coords * 1.0 + shift;
        u_freq *= u_lacunarity;
        amp *= u_persistence;
    }
    return sum;
}
```

Illustration 21: Fractal Brownian Motion

Sampling 2D Perlin noise to retrieve a circle return some undesirable results. The circle looked completely flat and lacked the sphere shape where the texture skews around the edges. Rotating the texture was not really possible in a realistic manner.

The alternative was to use a 3D sphere and texture it. GMS has limited 3D functionality compared to other engines but it is completely usable for problem such as this. Building a sphere in GMS is quite simple. GMS provides functions to build primitives to be used in shaders. Once the sphere is made it is only a matter of texturing it.

I need to sample 3D perlin noise such that the resulting texture seamlessly wraps around the created sphere(as in Illustration 22).

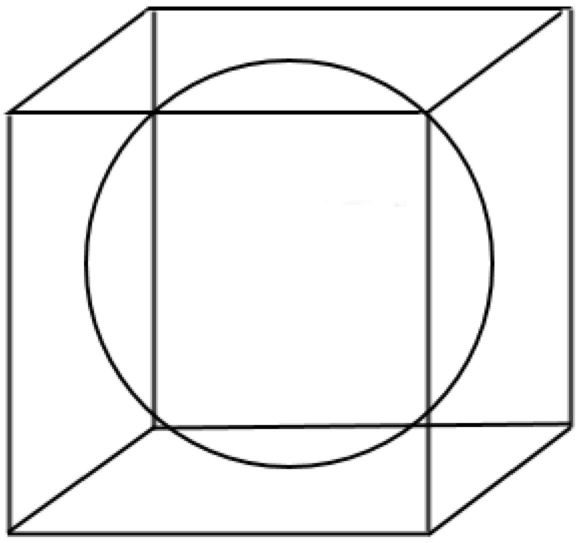
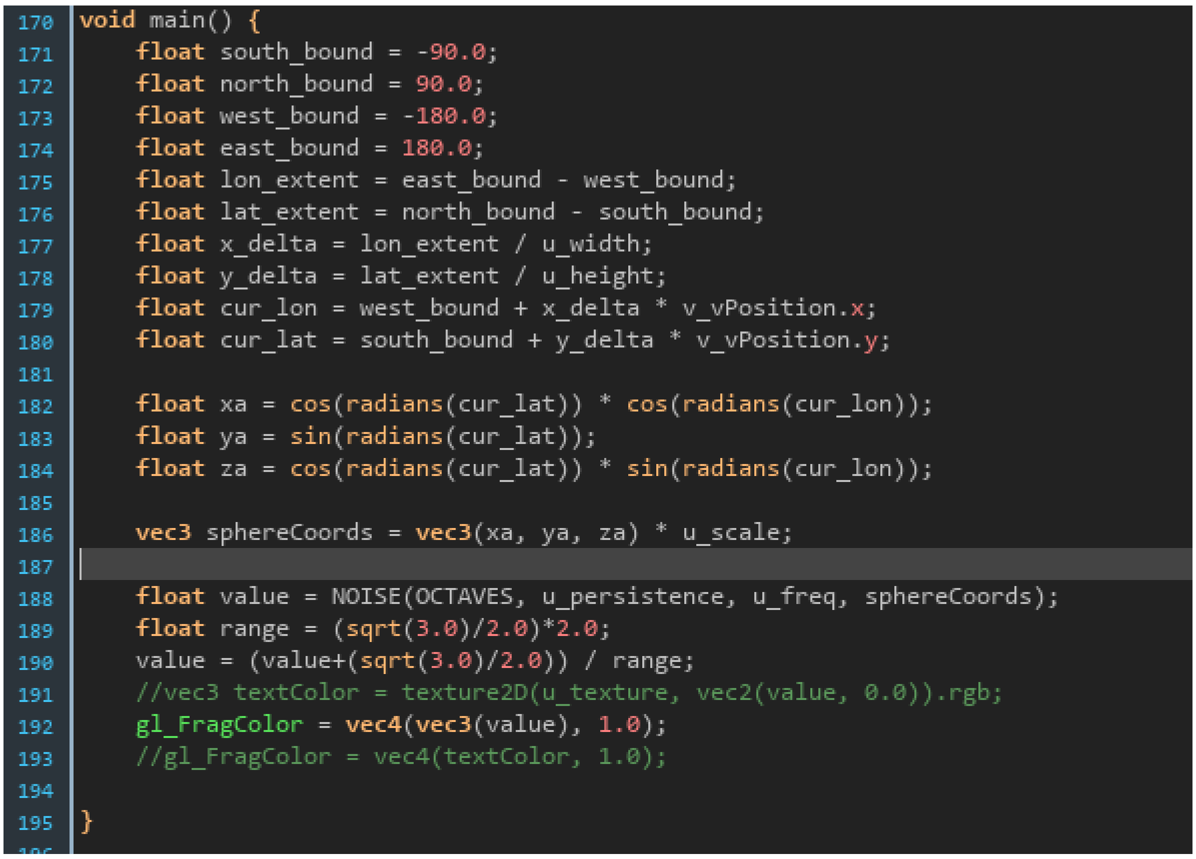


Illustration 22: Sphere in 3D space

To sample this I used the system used in GPS devices to display the earth. The code as follows:



```
void main() {
    float south_bound = -90.0;
    float north_bound = 90.0;
    float west_bound = -180.0;
    float east_bound = 180.0;
    float lon_extent = east_bound - west_bound;
    float lat_extent = north_bound - south_bound;
    float x_delta = lon_extent / u_width;
    float y_delta = lat_extent / u_height;
    float cur_lon = west_bound + x_delta * v_vPosition.x;
    float cur_lat = south_bound + y_delta * v_vPosition.y;

    float xa = cos(radians(cur_lat)) * cos(radians(cur_lon));
    float ya = sin(radians(cur_lat));
    float za = cos(radians(cur_lat)) * sin(radians(cur_lon));

    vec3 sphereCoords = vec3(xa, ya, za) * u_scale;

    float value = NOISE(OCTAVES, u_persistence, u_freq, sphereCoords);
    float range = (sqrt(3.0)/2.0)*2.0;
    value = (value+(sqrt(3.0)/2.0)) / range;
    //vec3 textColor = texture2D(u_texture, vec2(value, 0.0)).rgb;
    gl_FragColor = vec4(vec3(value), 1.0);
    //gl_FragColor = vec4(textColor, 1.0);

}
```

Illustration 23: Sampling a sphere in 3D Noise

First we get the latitudinal and longitudinal ranges(as in Illustration 24).

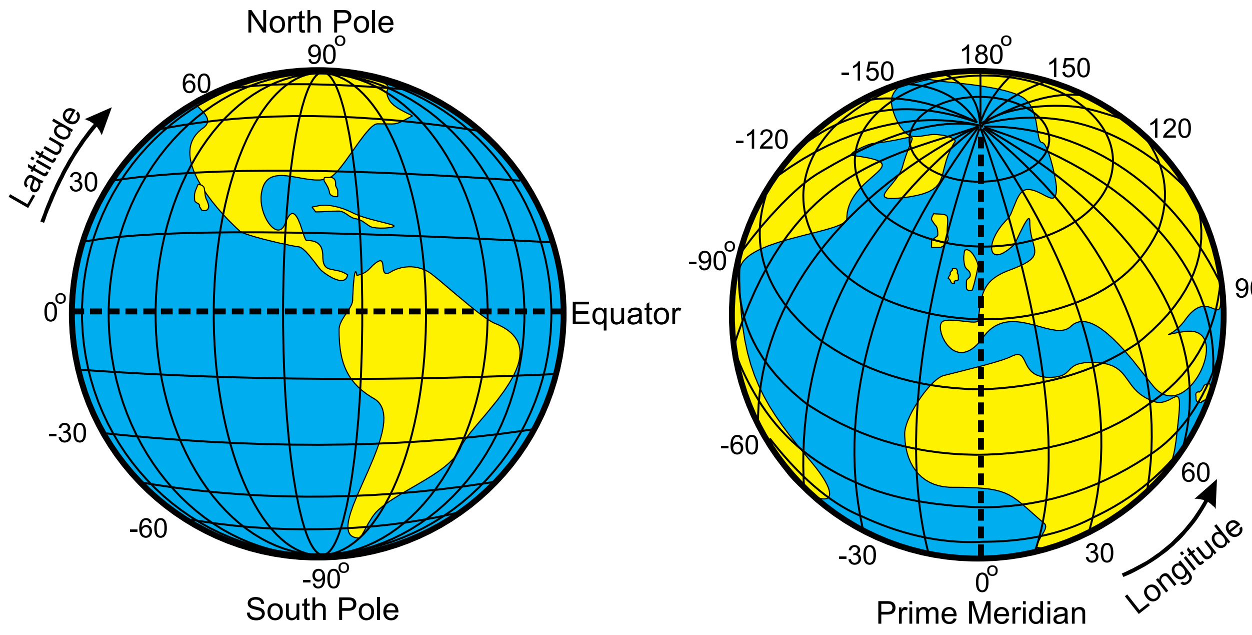


Illustration 24: Latitude and Longitude of a sphere

Divide them by the width and height of a defined resolution size (the texture size) to get our x and y -delta. This is our increment across the surface of the sphere at every pixel.

Finally to get our (x, y, z) coordinates we must convert from this geodetic coordinate system to ECEF coordinate system. The formulae[16] for this is defined as:

$$x = r * \cos(lat)\cos(lon)$$

$$y = r * \cos(lat)\sin(lon)$$

$$z = r * \sin(lat)$$

In the case GMS and several other engines in the above formulae the y-axis and z-axis are swapped. In the code in Illustration 23 r is equal to the u_scale variable. Once sampled the noise is normalized to the range of [0, 1]. This is due to how colors are defined in shaders. The RGBA format the shader uses is defined in the range of [0, 1].

The resulting texture:

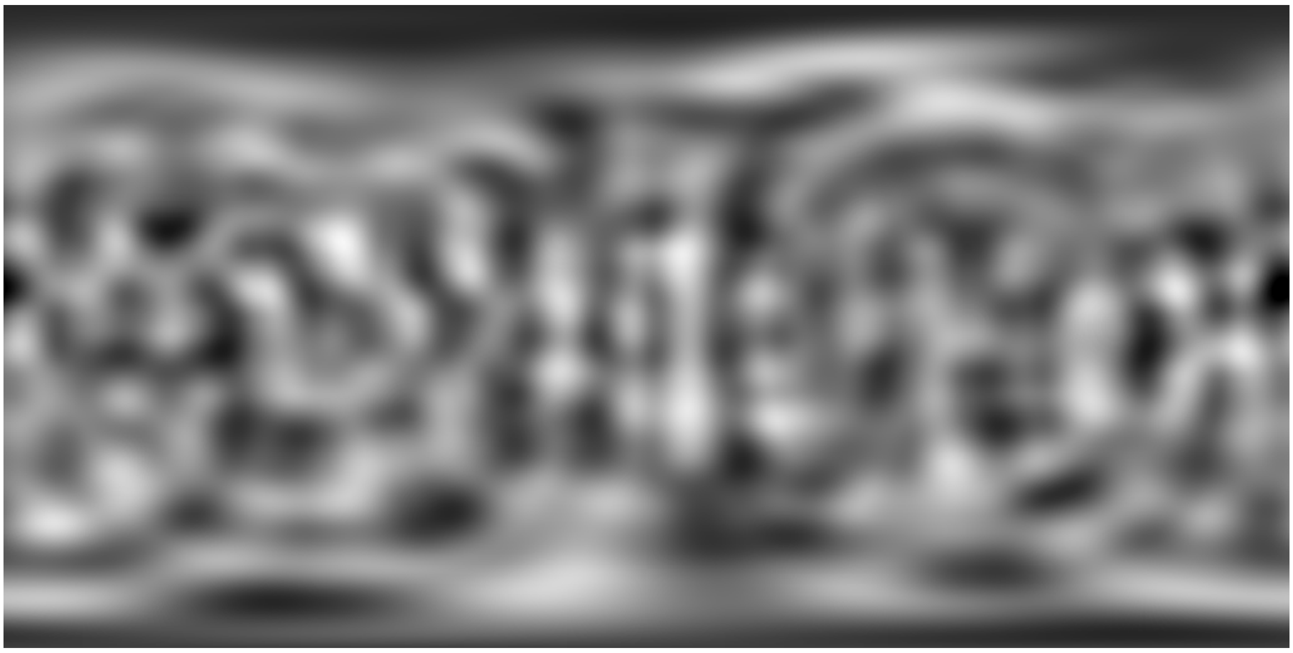


Illustration 25: Perlin Noise 3D sphere texture

The resulting texture has noticeable distortion around its poles.

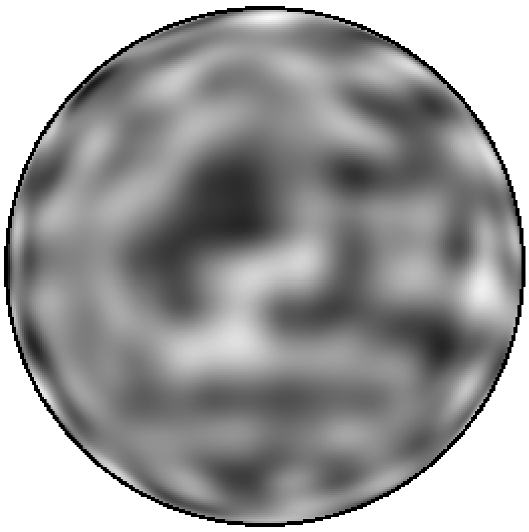


Illustration 26: Texture wrapped around a sphere

But it wraps perfectly around the sphere. There’s an interesting property of Perlin noise that is visible here. There are a few straight well defined directional patterns within the texture (Illustration 27). This is considered an artefact and was subsequently part of the motivation for Ken Perlins creation of Simplex Noise.

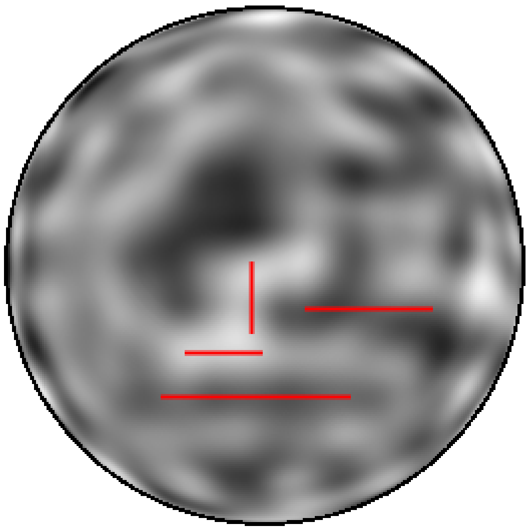
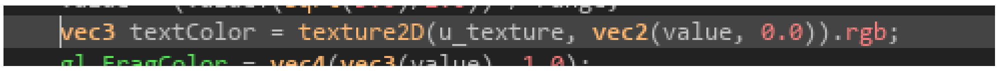


Illustration 27: Directional Artefacts

To color the sphere there are two options. The first is to define ranges for specific colors. For example any value within the range of [0, 0.2] might be the color blue. The another alternative is to pass a gradient texture into the shader.


```
vec3 textColor = texture2D(u_texture, vec2(value, 0.0)).rgb;
```
Illustration 28: Choosing a color from a gradient texture

Given that the range of the noise is [0, 1] we can define each point on our gradient texture as a value within that range. Simply sampling the textures at each x and y value passed into the noise texture.

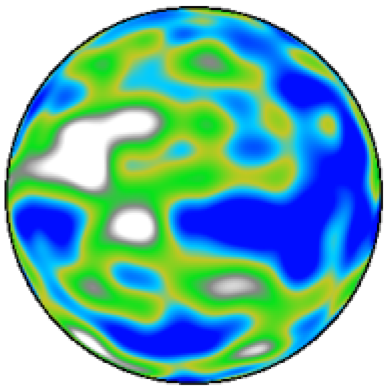


Illustration 29: Colored planet

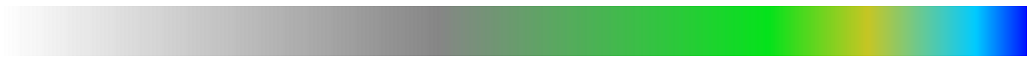


Illustration 30: Gradient for Illustration 29

Using different colours with varying strengths on the gradient can return colorful unique planets.

Finally the last thing to do was to get the planets to rotate. There are two methods that allow this. First is to use a rotation matrix and the second is using quaternions. Matrix support in GML is quite recent addition. Before any matrix calculation would have needed to be manually coded. Even with the new matrix functions the rotation matrix method is quite verbose. Thus I stuck with the latter for this project.

Quaternions is a really interesting topic. They were described in 1843 by Irish mathematician William Rowan Hamilton. Quaternions allow for a rotation is 3D space. A quaternion (w, x, y, z) defines a rotation in 3D space.

The code for which is a port of:

http://www.geeks3d.com/20141201/how-to-rotate-a-vertex-by-a-quaternion-in-glsl/

to GML.

Even though quaternions define the rotation there is a bit of an addition needed to this. I had to separate GMS movement system from the rotation of the sphere with quaternions. Since the planet is orbiting around a point in space we can't use the movement system defined by GMS.

This just requires the use of Pythagoras theorem.

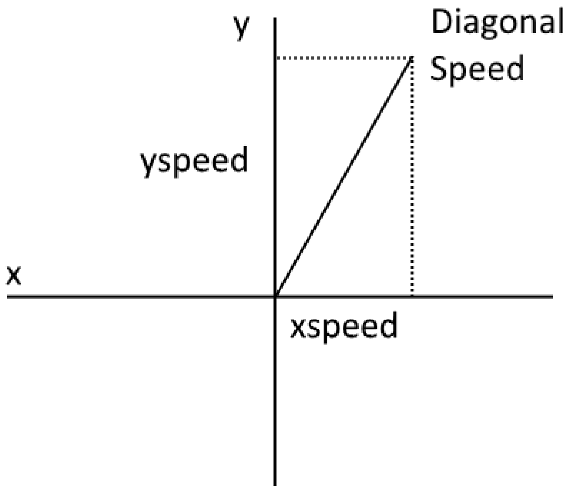


Illustration 31: Diagonal Speed calculation

From Illustration 31 we can clearly see the triangle created when an object is moving a certain speed along the x-axis and y-axis. Thus the diagonal speed is equal to:

$$\sqrt{xspeed^2 + yspeed^2}$$

Once disconnected from its orbit the planet can now rotate about its own axis.

Bringing the project together

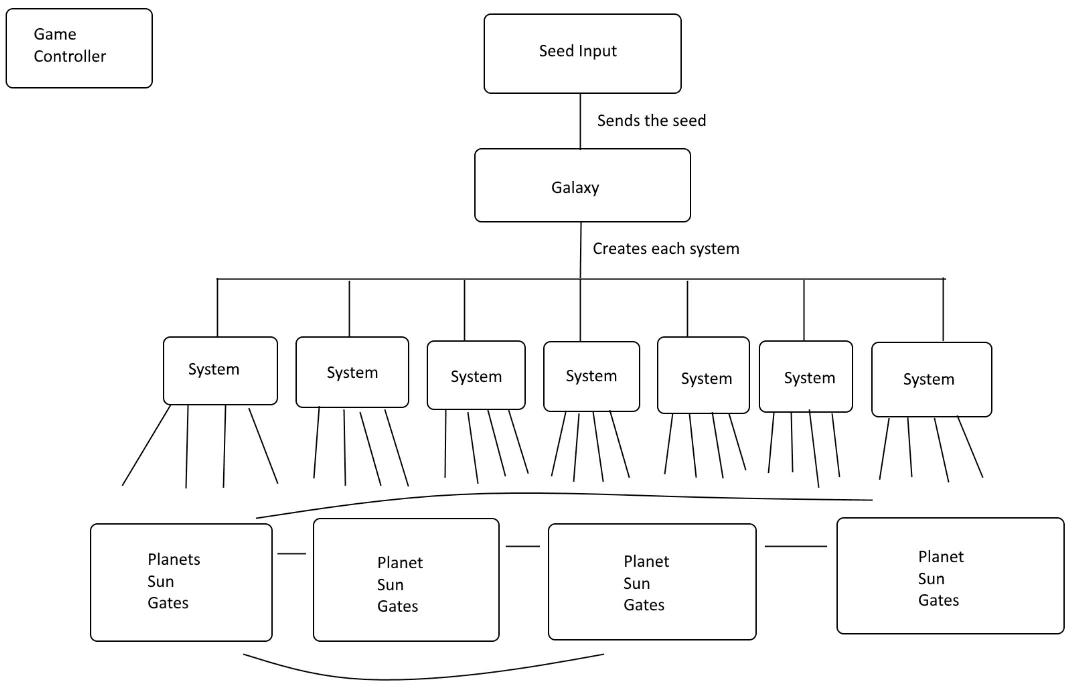


Illustration 32: Quick Sketch of the layout of the map creation

One of the big things I enforced was that any child object within the tree (Illustration 32) that is changed or edited will not have any effect on the parent. This means that if in the future I try to add systems to the planets, the solar systems will be unaffected by changes.

Because of the strict order of events and the PRNG in GMS; given the same input seed the output map is always the same at every level. Using the debugger I could view the same data being created every time on launch.

I think this project worked out quite well. I managed to accomplish a map creator based on what I defined. Given more time I would've liked to have worked with some of the noise functions. In particular applying noise to the sun with the dimension of time to get a flowing effect.

Chapter 3 (Technical Issues and Analysis)

Technical Issues

There biggest issue in this project was definitely the terrible performance of the GMS engine when running a gradient noise function in GML. The resulting performance was abysmal. Although using shaders solved this problem, it led to another issue. OpenGL ES 2.0 is an older version of the standard. As such, this led to issues porting the noise algorithms over. For example in Cubic Noise the algorithm makes use of the XOR operator. This operator is not supported in OpenGL ES 2.0[17]. However it is supported in the latest version of OpenGL ES. It is possible that the version may be updated in future in the GMS engine.

Given the above if I was continuing on with the project I would probably port it to a different engine. GMS is very good for what it can do considering the speed at which you can get parts of a game up and running.

Performance is a key topic with this game. On testing the display of multiple planets on screen at once there was a noticeable drop in FPS. With 1 planet on screen the game runs at a solid 120 FPS. With 2 planets there is ~15 FPS drop. At 3 planets or more the FPS bottlenecks at 50 FPS. A solution to this is to simply not calculate the noise at every draw frame within GMS. Because of this I implemented a surface and buffer system. The noise is first drawn to a surface then saved to a buffer. Once the planet is outside the player view the surface is destroyed and no longer being render. If the player gets into range of the player again the surface is reloaded from the buffer and drawn again.

Another roadblock was the issue of how GMS handles sprites. Sprites are saved to 'texture pages'. When a sprite is referenced to be drawn the texture page is called into memory and the sprite is pulled from it. If another sprite is called that has a separate texture page then a texture page swap occurs. Too many of these swaps can cause slowdown. When you pass a texture reference into a shader in GMS it does not just pass the sprite, it passes the whole texture page. It took me a while to figure this out as the documentation on such is lacking. The solution to this problem is to either have a multiple texture pages for each gradient or find the normalized coordinates of the gradient in the texture page and pass this to the shader.

Analysis

There are a few parts in the map design that I discovered upon completion of this project. Notably the scale of things is probably too big. The map is currently set to 12000x12000px. This could probably be scaled down to about half.

Initially the planets orbited too fast around the sun, this could possibly get frustrating if the player is trying to find their planet or move units to it. I think a solution is to slow this speed down drastically. Moving at 1 pixel every step is too fast. I think a change to about a tenth of that with keep the dynamic movement of the planets without creating a frustrating experience.

The planets rotate on a single axis which looks quite jarring when orbiting the sun (Illustration 33).

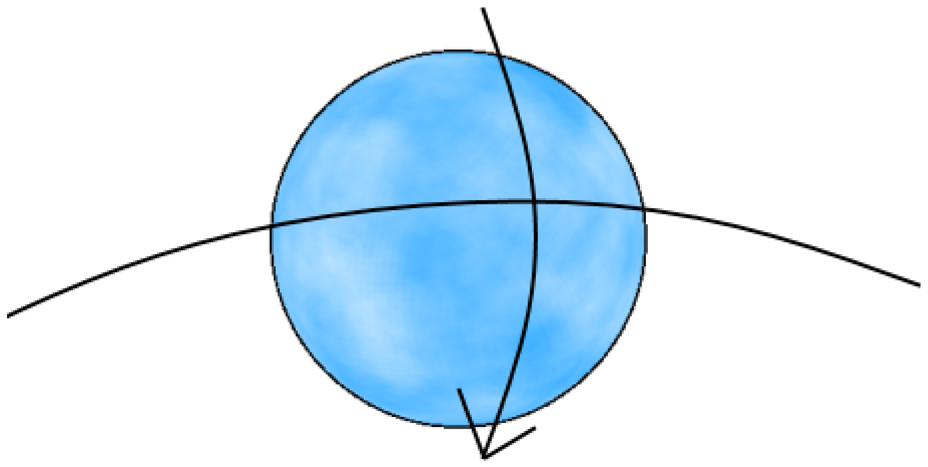


Illustration 33: Planet rotation

I think a solution to this would be have it spin around like a top (Illustration 35).

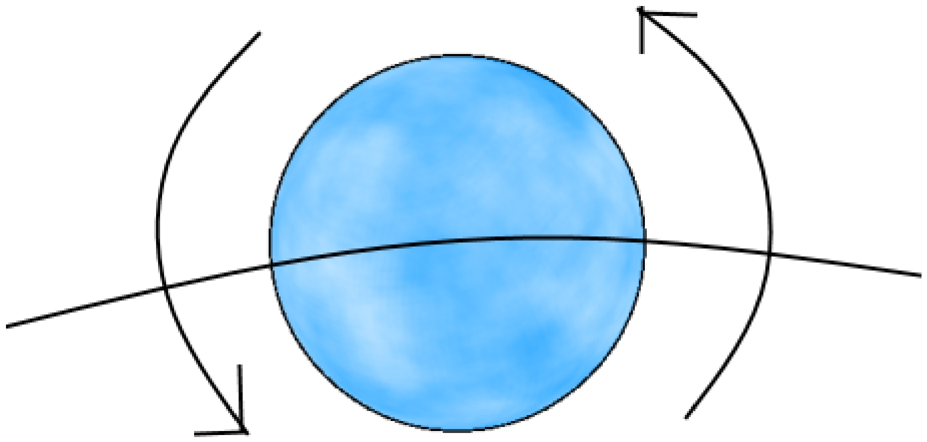


Illustration 34: Alternative rotation

This could remove that sliding feel, like the planet is on ice. This would require alternate increments in the xspeed and yspeed of the rotation.

The current gate system is also quite jarring. There’s no transition between each zone.

Chapter 4 (Future Work and Conclusion)

Future Work

The next logical step for this project is to add gameplay mechanics. Starting small by adding one or two different units, being able to move them from system to system, control them by grouping them and possibly an early combat system.

There's also another procedural part to the game and that's the planet properties. Each planet could have a different resource or could produce special units specific to that planet. I believe every step of the way should be tested thoroughly to find out what works and what doesn't from a gameplay perspective.

Development would be an iterative process adding piece by piece until the game is feature complete. I would like to keep the Mediator pattern within this game. Making sure any objects that communicate have a mediator (as is the Game Controller). This decoupling also will make sure any changes made do not affect the game as a whole.

Graphically the game could do with a lot of polish. In particular the background of space on the map needs some sort of image to give a sense of movement. The addition of stars or using noise and an appropriate gradient could produce a nice cosmic appearing result. To create a seamlessly tiling 2D perlin noise texture would require the use of 4D noise. Since we sampled the planets in 3D space and this produced a horizontally tiling sphere, then we can assume that a horizontally tiling texture can be sampled from a cylinder in 3D space.

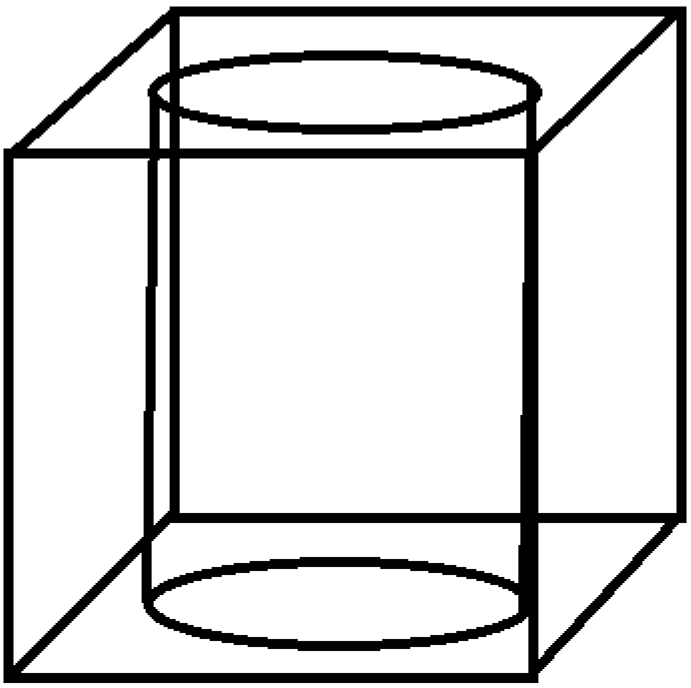


Illustration 35: Cylinder Sampled in 3D Space

To get it to tile vertically we would need to use an extra dimension. To best visualize it would be to image two cylinders creating an x and y axis. This should provide a perfectly tiling image.

I think two of the most difficult parts in any future work would be AI support and multiplayer support. I think before tackling these I would want to be using a different engine. There is not a lot of control in GMS for quite a number of things that other engines provide. Other than using extensions to add more support I would think there were be several issues getting multiplayer to work. For AI I don’t necessarily think it wouldn’t work in GMS but considering that my project has slowdowns already it could be a lot worse with a complex AI doing multiple things at once.

Conclusion

This project set out to show that its possible to create game worlds cheaply without having extensive resources that are usually found in big budget games. The use of procedural generation techniques such as perlin noise helps to provide uniqueness to the maps.

I think this project could definitely be developed further into a full game. For an independent developer there are a lot of options out there today to publish a game. The Steam digital market platform has become one of the biggest and easiest platforms to publish your game on. There’s also the option of kickstarter to crowdfund your project. With the decreasing costs and vast information it is entirely possible to make a game completely solo and release it. For the GameMaker: Studio engine on of the popular games to have been made in it in recent years is Undertale created by solo developer Toby Fox. This game went on to huge commercial success.

The learning outcomes of this project have been substantial for me. In particular the topic of quaternions and gradient noise is fascinating. If I continue with any sort of game development in future this knowledge will certainly be useful. Perlin Noise has many wonderful applications. Not only can it be used for texture generation, it can also be used for terrain height maps.

Chapter 5 (References)

All the following text, code and image references were accessible on the 15th April 2018.

Text References

1. https://stackoverflow.com/

2. https://en.wikipedia.org/

3. http://pcg.wikidot.com/

4. http://progamedev.net/wp-content/uploads/2017/08/Games_and_Interactive_Media_Report_2017_SuperData_Research.pdf

5. https://www.forbes.com/2008/11/20/games-indie-developers-tech-ebiz-cx_mji_1120indiegames.html#7aa5c68f73a6

6. http://9thimpact.com/

7. http://9thimpact.com/9th-impact-games/danger-mouse-the-danger-games/

8. https://www.unrealengine.com/en-US/release

9. https://unity3d.com/legal/terms-of-service/software

10. https://github.com/godotengine/godot

11. https://www.yoyogames.com/get

12. https://www.theguardian.com/books/2003/oct/18/features.weekend

13. Wikipedia article on Delaunay triangulation:
https://en.wikipedia.org/wiki/Delaunay_triangulation

14. “Simplex Noise Demystified” by Stefan Gustavson
http://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

15. https://en.wikipedia.org/wiki/Fractional_Brownian_motion

16. http://clynchg3c.com/Technote/geodesy/coordcvt.pdf

Code References

Delaunay Triangulation for GMS code available at the following repository:
https://github.com/GameMakerDiscord/delaunay

Perlin Noise WebGL Implementation by Stefan Gustavson available at the following repository: https://github.com/stegu/webgl-noise

Quaternion rotation in GLSL code available in the following article:
http://www.geeks3d.com/20141201/how-to-rotate-a-vertex-by-a-quaternion-in-glsl/

Illustration References

Illustration 2: Age of Empires II Image accessed at the following location:
https://www.giantbomb.com/images/1300-2170685

Illustration 15: Picture of Delaunay Triangulation accessed at the following location:
https://en.wikipedia.org/wiki/File:Delaunay_circumcircles_vectorial.svg

Illustration 24: Picture showing the lattitude and longitudal system applied to a sphere accessed at the following:
https://commons.wikimedia.org/wiki/File:Latitude_and_Longitude_of_the_Earth.svg

Illustration 6: Was taken by me on my licensed copy of Minecraft

All other Illustrations were either drawn in paint.net by me or screenshots from the game at runtime.