Procedural Content Generation Assessment Item 1 Report

**Development Log**

The first part of the assignment I decided to tackle was the landscape elements, starting with creating the galaxies to fill the background of the diorama, which was simply black in colour. Following the workshop 8 tasks, I created a single galaxy for the background of the scene, I then added randomness to the constant variables of the equation to add variance to each distant galaxy generated. These galaxies are placed by randomly spawning them onto a large sphere collider that is situated in the middle of the game world. I then added an emissive material and applied random colour values to the galaxies, I also applied a rule that makes stars, closer to the centre of the galaxy more brighter (See Figure 1). This turned out to not be as effective as predicated due to bloom’s reliance on distance and the angle of the object, but can sometimes be seen (See Figure 1).

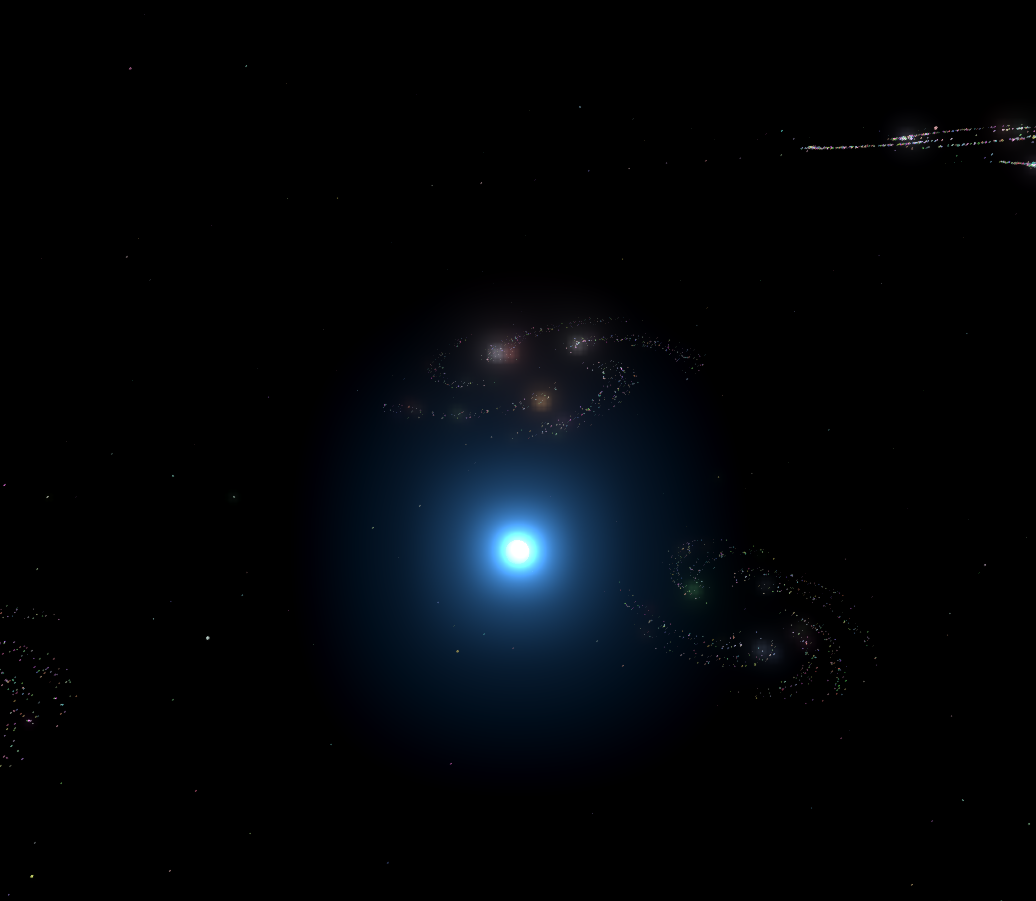
**Narrative**

The narrative of the diorama being created is that of a newly created colony fleet that has been created and assembled above the humanoid beings’ planet. The fleet’s mission is to colonise a nearby temperate planet in their solar system for their goal of humanoid expansion beyond their world. This large fleet contains many colony ships full of hopeful future residents of the nearby temperate planet, dubbed “Terra”.



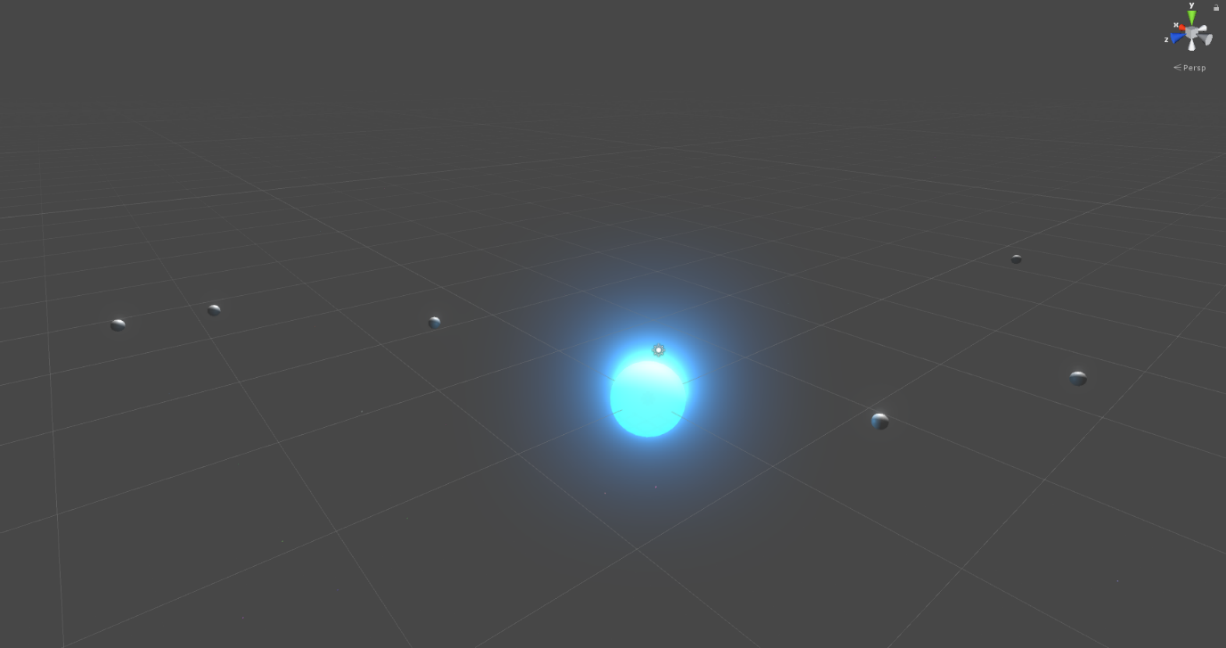
*Figure 1: Procedural distant galaxies being spawned around the diorama.*

With the diorama now having a functional background, I moved onto the spawning of the solar system and its planets. The star was simply generated at the centre of the game world, which is given a random size attribute, the scale of the star determines what colour it will result in being. Past the scale threshold the star has a much higher chance to spawn with a warmer colour set, similar to red giant stars and being below the threshold results in the star using cooler colours (See Figure 2).



*Figure 2: A small star (with cooler colours) generated in the diorama.*

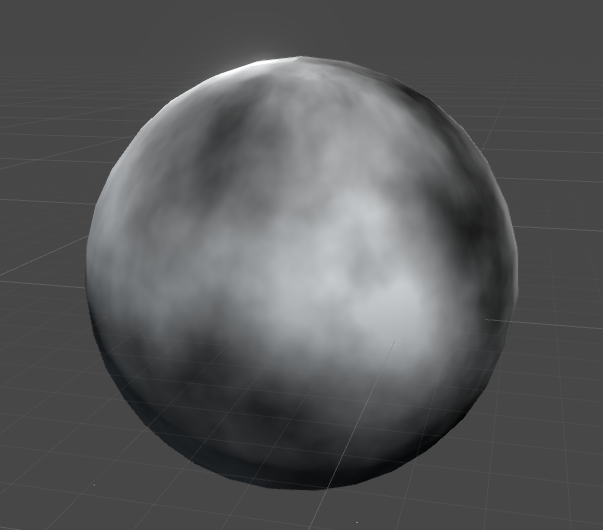
For the placement of planets in the solar system I used the polar coordinates system described in the lecture 8 slide materials. This system works by placing one planet after the other with differing minimum and maximum spawn values based on the previous planet, this ensures that the planets will never intersect each other when orbiting around the star (See figure 3). While this system is simple, it creates vastly different solar systems each time the diorama is started



*Figure 3: Spawned planets orbiting around the star using the placement system.*

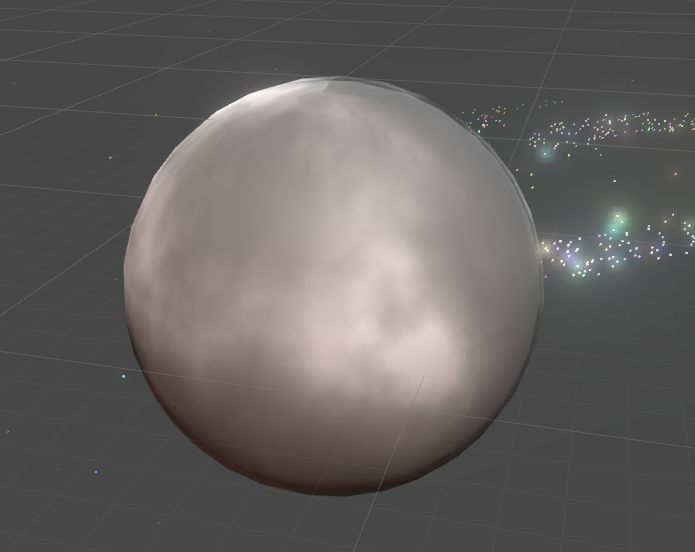
But the planets themselves were just boring grey spheres with zero variation between them, so the next problem to target was to provide variance to the planets.

Perlin noise from the LibNoise library (Bevins, 2007; Mendez, 2010) was used with the goal of creating procedural atmospheres for the planets to help give them variance. The values and intensity of planetary atmospheres are also dictated by probability, which is augmented by the distance of the planets from the star. This effect was achieved by placing another sphere with a transparent material over the planet (With a slightly larger overall scale size) and adding Perlin noise to the material’s main texture. The immediate issue was that the texture created by the Perlin noise generator has no alpha channel and is opaque, meaning that we can no longer see the planet underneath (See figure 4).



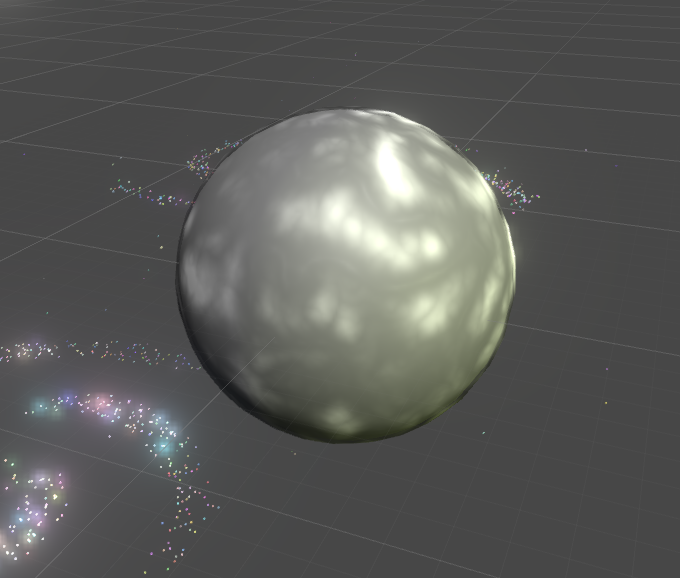
*Figure 4: A planet with the raw Perlin noise texture applied to it.*

This is solved by looping through all the pixels in the Perlin 2D noise texture and taking their colour and grayscale values, I then reapply the colour values back as normal but then apply the grayscale value to the alpha channel instead. Doing this means a value of 0 on the grayscale is black, which in the alpha channel means complete transparency leading to the desired effect (See figure 5).



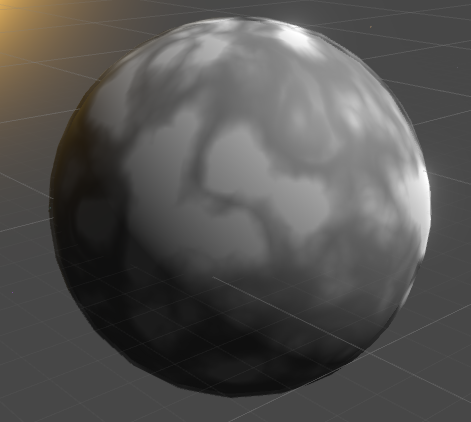
*Figure 5: A planet with the Perlin noise texture with the dark colour pixels phased out.*

While this is already looks like a planetary atmosphere, I wanted to push LibNoise further to see if it can make a more realistic looking cloud effect. I then realised that LibNoise had different forms of noise such as Billow, which had a wispier effect than Perlin Noise which looked more like clouds (See Figure 6).



*Figure 6: Planet with Billow noise generated atmosphere.*

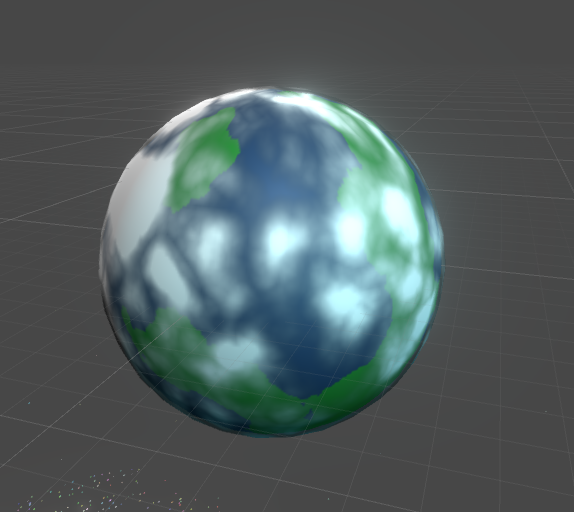
While this now looks acceptable for a planetary atmosphere, I was curious to combine the two noise generation methods together to see if it produced an even more realistic effect (See Figure 7).



*Figure 7: Planetary atmosphere created with a combination of Billow and Perlin noises.*

This method of atmospheric creation results in a thicker, denser looking atmosphere which is effective on types of planets such as gas giants. Both techniques can be used to create different types of planets, based on the probability system which is augmented by a planet’s distance from the star of the solar system.

For the planet’s terrain, my original plan was to use noise to augment the vertices of the planets to give them terrain. I then realised that this approach is likely unnecessary as the atmosphere of most planets will cover the terrain generated anyway meaning it would not be as impactful to the scene and could be more computationally expensive to apply. I settled on generating a 2D texture using Perlin noise and then changing their colour values based on their grayscale value, like the atmosphere system. Darker values would result in the planet’s oceans and the lighter values would result in land (See Figure 8).

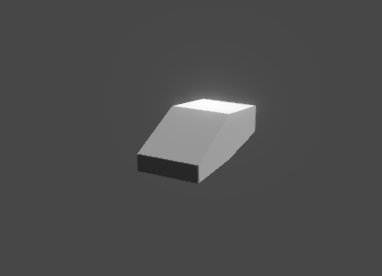


*Figure 8: A complete temperate planet*

By creating these planets procedurally with noise, it allows for a large number of possible combinations meaning that each planet will be different from each other. The noise also produced a realistic looking atmosphere that would have taken a large amount of time to produce one atmospheric texture manually. Whereas procedural generation allows for much faster creation of the same effect with a potentially infinite number of possible variances.

With this system now complete it was now time to progress onto the ship generation. This began with modelling various ship parts to be used in a modular system that the system would then use to build the ships.

For the guards/fighters a base shape was created for the modular parts to be placed upon (See Figure 9).

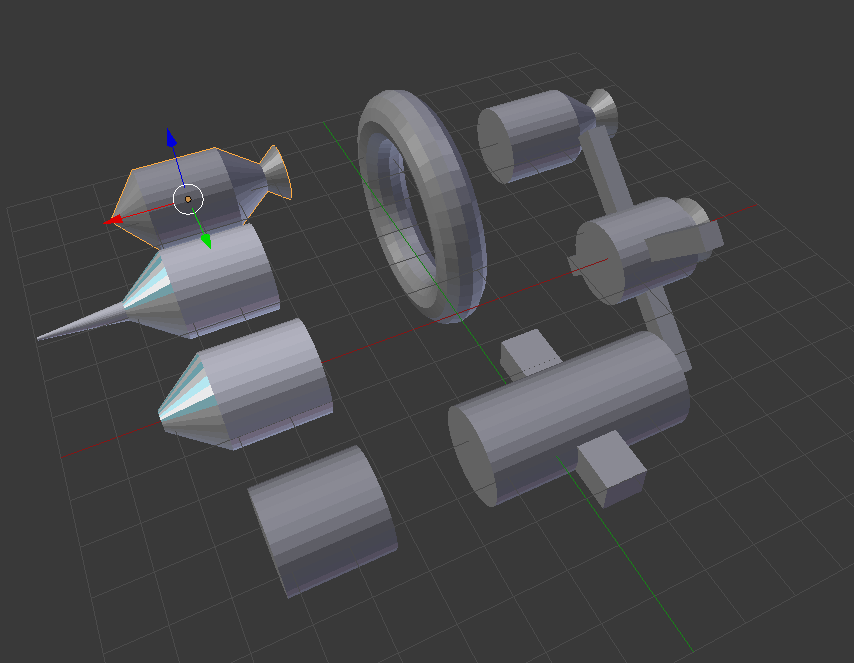


*Figure 9: Base model for the colony fighters/guards.*

The fighter base is then created as a prefab with multiple empty game object “nodes” attached to it in predefined positions that I positioned manually. These nodes make up all possible combinations of components that can be attached to the fighter base to make up the completed fighter. This system was created to eliminate any amount of erroneous results (See Figure 10).

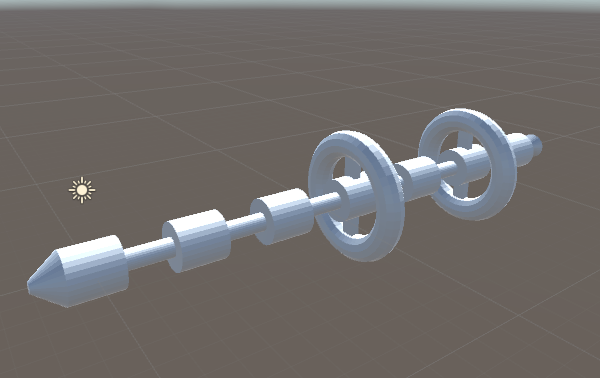
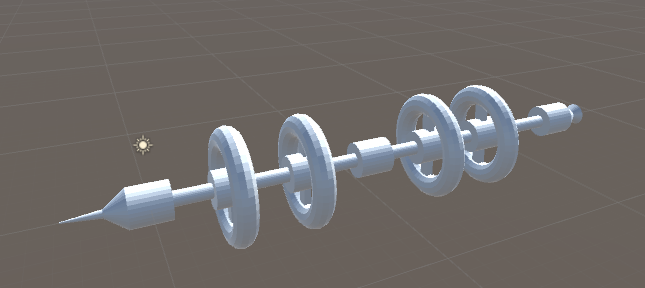
[MULTIPLE FIGHTERS CREATED HERE!]

The colony ships are created differently from their fighter/guard counterparts, I used an odd/even number system that was previously intended to be used on a moonbase system for this assignment that failed to work correctly. In this instance the odd/even system worked to create effect, creating the different parts needed to create the moonbase in a sequential manner, but the placement of the buildings did not work as intended and such this system was scrapped. I then realised that this system could be implemented to generate the colony ships from parts I had modelled in blender (See Figure 10).



*Figure 10: Ship parts to be used in the colony ship generation system.*

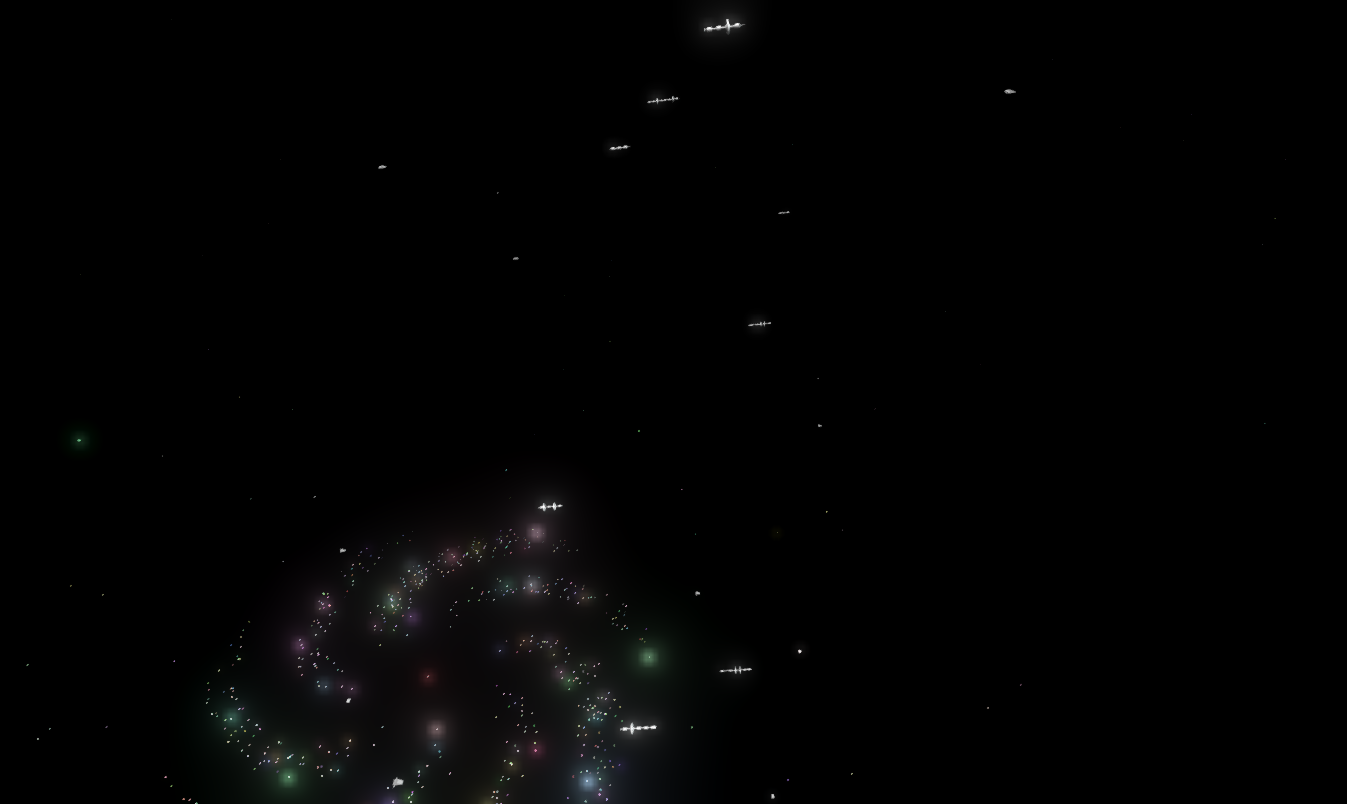
The odd/even system means that for every odd number a connection must be placed and every even number means a module must be placed. The module type depends on a simple probability variable that is randomly assigned each loop of the colony ship build. The first number of the loop will always be one of two cockpit models and the final number will always be an engine module, combined this creates the colony ships with a good level of variance (See Figure 11).



*Figure 11: Two colony ships made with the colony ship generator.*

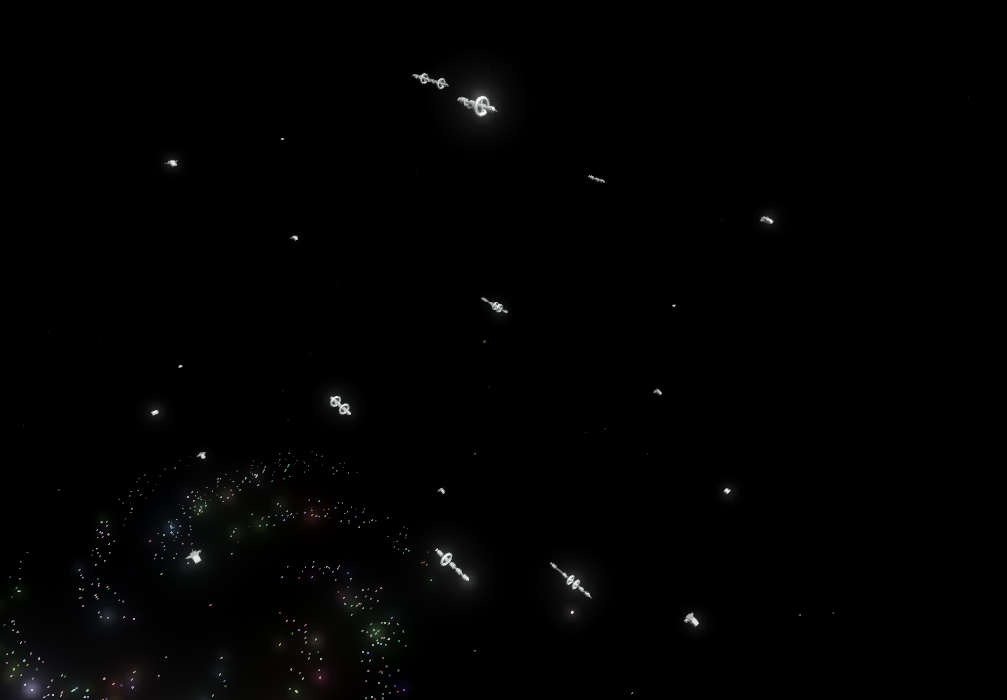
This system is now able to produce a potentially infinite number of colony ships which all vary in visual appearance, and many can be produced within seconds of the game loading. Doing this manually with the same result would have been a huge time consuming task and would have still have been limited in variance whereas with this system, the possible combinations of colony ships is extremely large.

With both colony ships and fighter/guards being created, it was now time to move onto placement of these ships as a fleet in the diorama. The solution was to simply instantiate a fleet controller parent in the world and have all ships randomly spawn as children within its sphere collider trigger using *Random.inUnitSphere* (See figure 12). The odd/even system was used again, but for this system it was used to spawn the fleet between guards and colony ships to ensure the fleet was evenly distributed.



*Figure 12: The ships being spawned into the scene as a fleet.*

To get the fleet to move as a coherent unit, the first action taken was to move the parent object towards the target, moving the individual ships along with it. While this implementation did work, it made the fleet feel static as there was no real individual movement among members of the fleet. To solve this, the boids algorithm was implemented to add more life to the fleet by making their animation procedural (See figure 13).



*Figure 13: The fleet now with boids algorithm implemented, they now move as a more coherent unit.*

Bevins, J. (2003) *LibNoise* [coherent noise-generating library]. Available from <http://libnoise.sourceforge.net/> [accessed 29th April 2018].

Mendez, J.R. (2010) *LibNoise.Unity* [UnityPackage]. Available from <https://github.com/ricardojmendez/LibNoise.Unity> [accessed 29th April 2018].

Perlin cloud atmosphere based on distance, do multiple layers for thicker atmospheres, distance from star affects probability of size, terrain jaggedness, colour, moons. Galaxy stars brightness affected by how close they are from the centre of the galaxy.

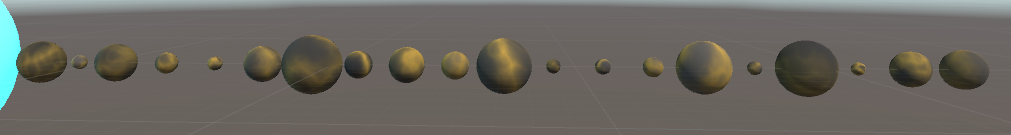
Ships move along empty gameobject waypoints that transition them between planets and other items

Planets that are a certain distance away from the star will have a higher chance of getting asteroid rings

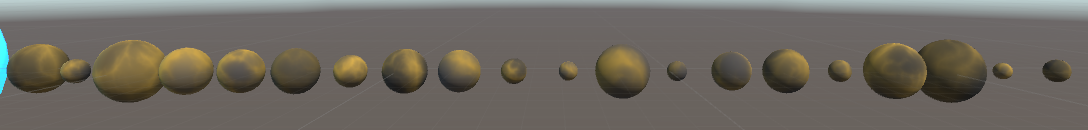
**Testing**

The approach for testing involves breaking down the procedural systems into their subsystems, such as the planet’s system’s texture generation via noise for the atmosphere and the surface of the planet. To thoroughly test these subsystems, the independent variables of the noise parameters will be constrained to thresholds and tested in quarters. A sample size of 20 planets will be created by the system in its own isolated scene, to ensure this system is working as expected.

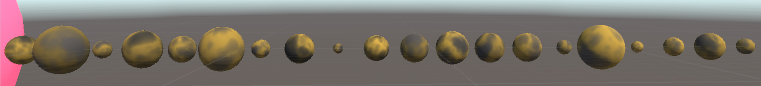
Splitting the variables into quarters will help provide a broader coverage of the entire range by constraining the median value of the range into 4 places. Using 20 planets as a sample provides ample distribution across each quarter, allowing for the program to replicate the majority of outcomes. This process can also be repeated by running the program multiple times to force it to produce different results.



*Figure 14: The full/default range of variables being used to create Barren surfaces.*

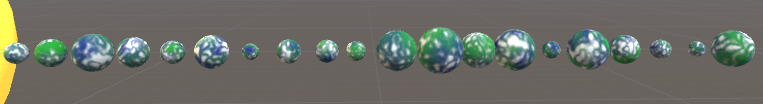


*Figure 15: Barren surfaces created with frequency range (0.4, 0.8 (default)) being constrained to the lower quartile (0.4, 0.5).*

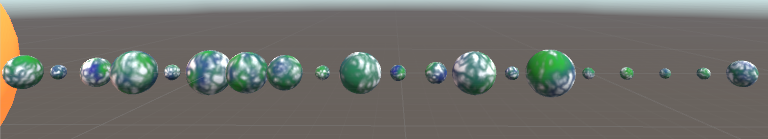


*Figure 16: Barren surfaces created with frequency range (0.4, 0.8 (default)) being constrained to the upper quartile (0.7, 0.8).*

This methodology was repeated for each noise parameter of the barren surface subsystem to ensure each value was producing expected results, and was also repeated for each surface and atmospheric subsystem (See figure 17 & 18).



*Figure 17: Temperate surface and atmospheres being generated at their default/full variable ranges.*



*Figure 18: Atmosphere frequency variable being constrained to the upper quartile.*

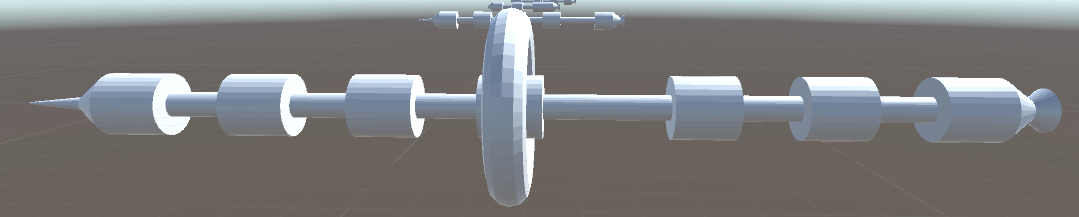
After all planetary texture subsystems were thoroughly tested for erroneous results, the placement of the planets was tested to ensure planets would not collide with each other.

Finally, the colony and fighter/guard ship building algorithms were tested to ensure they produced expected outputs, by using the same methodology as before (See figure 19).



*Figure 19: Colony ships being created (top-down view) with default parameters.*

Some ships were created in ways I did not expect, some ships used two connectors after a module instead of one which was the expected behaviour (See figure 20).



*Figure 20: A colony ship that used two connectors after the ring module.*

However, while this result was unexpected it added even more variance to the ships by making them look less uniform and breaking up their overall structure.

**Reflection**

Overall, I am satisfied with the implementation and final outcome of my procedural diorama. My decision to create all content within the scene procedurally has led to drastically different results that make the scene feel different each time. I am especially satisfied with the odd/even number system I created to produce the colony ships and my use of procedural noise to create planetary textures. Creating the odd/even system taught me about the importance of abstracting a problem down into its simplest form to then create an effective yet simple solution to the problem, the odd/even system. This module has enlightened me about the power and potential applications of using procedural/coherent noise to solve problems, in my case I used noise to create atmospheres and surface textures for planets, to great effect.

This module has also highlighted my ability to problem solve is lacking to some degree, it has taught me the importance of understanding the different tools that are available to me and the ways in which tools can be used creatively and sometimes used outside of their designed use case, to solve problems in an inventive way.