UNIVERSITY "BABEȘ-BOLYAI" OF CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

COMPUTER SCIENCE ENGLISH

DIPLOMA THESIS

Creating a large-scale virtual universe with procedural elements

Graduate

Radu Teofil Bocăniciu

Scientific coordinator

Dr. Molnar Arthur, Lecturer Professor

2019

UNIVERSITATEA "BABEȘ-BOLYAI" DIN CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ INFORMATICĂ

INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENȚĂ

Crearea unui univers virtual de scală mare cu elemente procedurale

Absolvent

Radu Teofil Bocăniciu

Coordonator Științific

Dr. Molnar Arthur, Lector Universitar

2019

Table of Contents

1. Introduction

1.1 Overview

The following paper will consist of a detailed discussion and exposition of various conceptual frameworks that would contribute to the collection of aspects that describe a realistic, large-scale universe. These ideas will be translated into functionalities that are then implemented in a 3D, single-player, exploration, and simulation video game in Unity [14], on a desktop (PC) platform, set in outer-space. What does this mean? Large, celestial bodies as well as smaller items like stars, planets, and spaceships respectively are represented in the same frames to produce an immersive space environment. The application will simulate massive sized procedurally generated bodies together with smaller sized objects and large distances in a seamlessly natural, diverse, and multifarious way.

The purpose of this project is to document relevant, applicative concepts that take part in the creation of a large scale game world set in space. In order to evaluate these frameworks on a tangible level it will be used to develop a functionality-focused video game application that simulates the generation of a realistic virtual universe with various unique objects at large and small magnitudes. The priority is to explore and implement various concepts and functionalities that achieve this purpose, while graphics and the aesthetic aspects like 3D models and textures are not as thoroughly emphasized.

Several issues arise from attempting to reach this goal:

- Will scale be a problem? Is it possible to achieve very large size distances and create big objects?
- Does everything have to be handled manually? How much can we generate automatically?
- How close to reality can we get?
- How can we create truly unique objects without a manual approach?

Because the application in which those concepts are applied will be created in Unity, the focus of the subjects related to the game engine will be corresponding to the Unity 3D game engine.

1.2 Background context and motivation

In order to perceive the size and scope of game worlds today, we must realize how small they once were. During the times of Space Invaders, Pong, and Pac-Man, it seemed enough for everything to be done in only one screen. Back when gaming first started to take off in 1970, everything was so incredibly limited that, in that sense, today's games aren't even comparable to the first titles. Since then, everyone in this particular industry has benefited from the massive increase of resources. Access to significantly more and more powerful hardware has turned restricted and narrow game worlds from games that didn't even exceed the 10 kilobye mark into worlds of vast expanses of terrain and space from titles that nearly take up to one quarter of a terabyte. This incredible progress has been a big attraction to people who play video games because in gives everyone a sense of lifelike experience (even though the game may be very different from real life), and unlimited possibility and exploration. [1]

I identify with this interest and fascination towards not just gaming and programming, but also the increasing possibility of experiencing "infinitely" (pseudo-infinite) large physical worlds with truly unique elements. It's one of the reasons I've chosen to push my abilities and research in this direction, thus resulting in my decision to adopt this subject as my bachelor's degree thesis.

2. Research and conceptual frameworks

2.1 Space simulations

2.1.1 Overview

A space simulator game is software that allows the user to experience space flight in an environment closely similar to outer space, with the added elements of gameplay. It consists of a system that attempts to replicate or simulate outer space as realistically as possible. There are many different types of simulators. In principle, space simulations range from pure, realistic simulation to added elements of game play and entertainment. Space flight takes place beyond the Earth's atmosphere into empty, zero gravity vacuum. Because of this, space flight simulators feature the basic flight phsyics mechanics of airplanes and space crafts. Namely, the ability to roll, pitch, and yaw in a zero gravity vacuum context. The player is free to fly under those circumstances in an open environment with the ability to move within the three-dimensional coordinate system or the x, y, and z axis.

When attempting to create a simulation of space, certain characteristics must be achieved in order to accurately preserve the simulation of the space environment. Things like zero gravity in vacuum space, representation of huge celestial bodies to a somewhat (pseudo) realistic scale, a night sky with distant stars / galaxies / nebulae, and basic simulation of outer space physics e.g. orbits.

2.1.2 Player movement

An important mechanic to consider in relation to space simulation games is the player movement. Common approaches include: point and click navigation, where only a mouse click is needed in order for the user to be brought to the desired location; free form camera movement, where the user can freely navigate with a camera towards places in space, with little to no restraints; spacecraft flight, where the user steps in the shoes of a character-like object inside a spaceship, being limited by some realistic constraints on the movement, due to the attempt to imitate real world spacecraft flight mechanics. In most space games, there is a hybrid combination of elements from all approaches.

In the case of simulating flight in space, there is a spaceship that can accelerate,

decelerate, turn (or yaw), roll, and pitch, in a zero gravity space. Usually, a camera that follows the spacecraft is implemented, and the user is able to control it in various ways: zoom in / zoom out, rotate around a pivot etc.

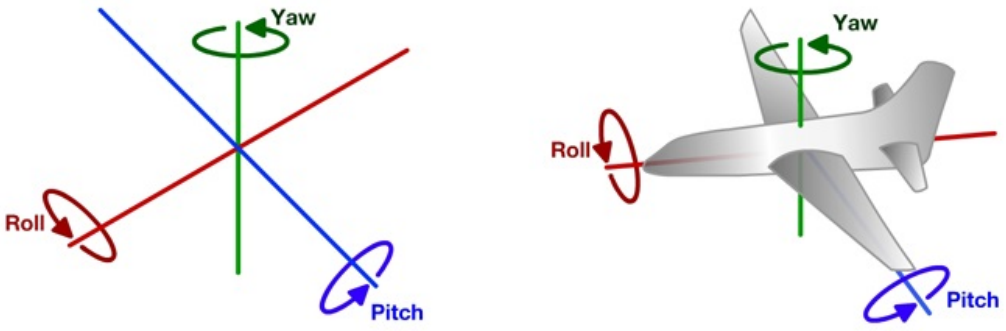

Figure 2.1: pitch, roll, yaw axes representation [12]

2.2 Large-scale open game worlds

2.2.1 World dimensions

Open-world video games are a type of video game where a player is free to roam through a virtual space, with significantly reduced constraints on how or when to operate in the game.[2] “Free-roaming”, “Open world”, “Sandbox” are terms that are often used to suggest nonlinear gameplay with the absence of artificial barriers.[3]

Video game worlds are always implemented as some sort of simulated physical space where the player controls his camera or character in and around this space and handles, manipulates, and interacts with other elements in it. The various physical properties of this space, such as spatial dimensionality, scale, and boundaries characterize the physical dimension of a game and also determine a great deal about the gameplay.[4]

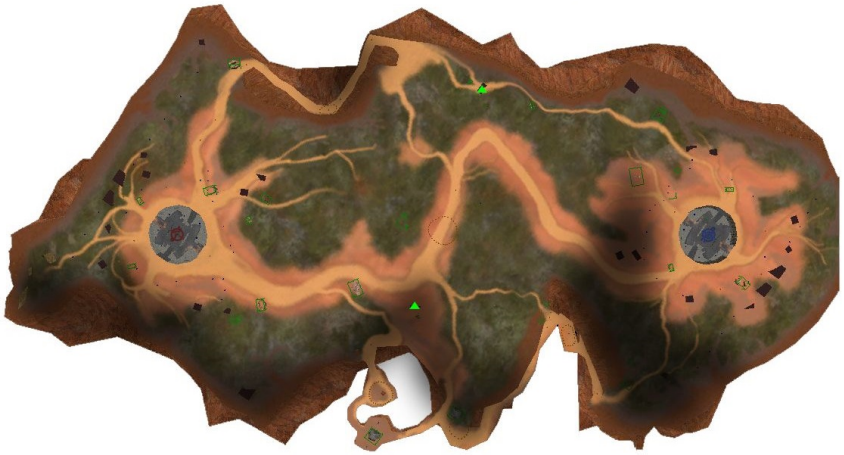

Figure 2.2: Halo Blood Gulch map top-down view[13]

One of the first important decisions to be taken is how many spatial dimensions will our physical space have - 2D or 3D. If we're talking about a realistic space game, 3D is the best option here. Thanks to 3D hardware accelerators and modelling tools, 3D spaces are now easier to implement and work with than ever. They give the player a much greater sense of being inside a world, rather than 2D representation, where the player only has the feeling of looking at it. Because of this the exploration limit also increases exponentially.[4]

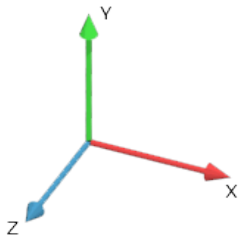

Figure 2.3: 3D axes representation

Scale refers to both the absolute size of the physical space, measured in units according to the game world (meters, miles, light-years) and the relative sizes of objects in the game. In a game that represents the real world (partially at least), the design must address the question of how big everything should be to feel real and play well. Some distortion is often desired or even necessary for the sake of gameplay, due to various limitations. We have to be careful to distort the scale without harming the player's suspension of disbelief too much. Because of the nature of this project, and the fact that it is set in space, it was

necessary to distort the scale to a certain degree, because of the huge distances between celestial objects like stars and planets, in order to improve the gameplay and exploration aspect of the game, even though it can be possible to represent real world dimensions with an almost precise accuracy.[4]

The physical dimension of a video game must have an edge, because computers have finite resources. However, computer games can maintain the player's sense of immersion, due to often trying to hide away the fact that the world is limited. So there's also some amount of illusion that comes into play. Our game being similar to flight simulators, setting the boundaries of the world becomes a problematic feat. There is no real natural way to limit the world in which the player is allowed to move. So most of the time, the solution unfortunately is to clamp the movement into a certain space and simply make the player unable to move past those limits. Alternatively, we can constantly 'create' objects in front of the player as it moves through the space while simultaneously 'destroying' the objects behind the player. This way, there's no real limit to the game, which seems like much more ideal option, but it's not as easy as it seems, as many other problems can arise from that challenge.[4]

2.2.2 Pros and Cons

Worlds in modern video games are continuously increasing in detail and scale, requiring greater and greater resources to achieve the creation of this content. In the present, these kinds of games are considered the norm. With series like Fallout, Metal Gear, and The Witcher, everyone now fully expects triple-A franchises to contain huge worlds to discover and explore. But with great size, comes great responsibility. Spending all the time on designing and developing the world, and forgetting about the fundamentals that make gaming great is far from ideal. However, this primary focus of this project is only on the game world, which allows negligence of other processes of game development. The point is that there are some disadvantages on insisting too much on this alone. An example of this was the title No Man's Sky, at least in the early stages of its release.[5]

There are also advantages as well as disadvantages from an isolated programming perspective:

- The game feels more immersive and realistic; (pro)
- Significant increase in possibility and exploration; (pro)

– Gives the player a greater sensation of control; (pro)

– Things like boundaries, distances, and size are much more difficult to define and represent, small to big ratio rendering harder to implement; (con)

– Loss of detail and unicity, everything must be generated in a complex way to preserve detail and unicity (procedural generation), because there is too much to design manually; (con)

– Issues dealing with movement due to lack of floating-point precision at greater distances; (con)

2.3 Game development

Video game development is the process of creating a video game. Games are generally developed as a creative outlet and/or to generate profit.[6]

A game is software with art. Because of that, like most computer applications, games are produced through the software development process, and everything starts from an initial idea or concept. Often times the idea is based on a modification or adaptation of an existing game concept. The game idea may fall within one or several genres. A game designer starts with writing a description of the basic concept, gameplay, features, setting and story, target audience, requirements and schedule etc. Following the process of game design is the implementation process, where the game programmer becomes involved. [7]

Game programming is the software implementation part of the game design of a video game. It's the subset of work that requires considerable skill in software development and computer programming.[8] Often times in the pre-production stage, it can be useful to develop prototypes in order to try out and examine different ideas and features.

A game engine is a software-development environment which provides creators with the necessary tools and features to build video games quickly and efficiently. Game engines are used by software programmers to develop games for consoles, mobile devices, and personal computers. Many games can be developed to work on more than one of those platforms. The core functionality typically provided by a game engine includes rendering 2D or 3D graphics, a physics system or collision detection, sound, memory management, scripting, animation, artificial intelligence, networking etc. The most important components in a game engine are: The main game scripting program which contains the game logic including base classes and predefined keywords; a rendering engine used to generate

animated graphics; an audio engine consisting of algorithms which are related to sound production; a physics engine that implements physical simulation and logic in the system.[9]

It is necessary to operate on a game engine, whether it is one that is created itself by the developers or chosen from a multitude of already existing game engines.

Most popular game engines include: Unity [14], Unreal Engine 4 [15], GameMaker Studio [16], Godot [17] etc.

A simulation game attempts to replicate various activities from real life in the form of a game. Usually there are no strictly defined goals in the game, with the player instead allowed to navigate and interact with the environment unrestrained. [10]

2.4 Multi-scale representation

2.4.1 The floating point problem

To create large-scale realistic worlds, one would think it's important to be able to render big dimensions and great distances. Or is it? Unity, as well as nearly every game engine, stores the position values of GameObjects in a single-precision floating-point format as opposed to a double-precision format. That's because every computer software and hardware, until a certain point, was designed and optimized for this format, which makes it very fast.

Single-precision floating-point format is a computer format which occupies 32 bits in computer memory. The first bit of this format is considered the Sign bit, which determines the sign of the number. The exponent is determined by the next 8 bits (from -128 to 127 if signed, from 0 to 255 if unsigned). And the last 23 bits are the fraction bits which represent the true significant of the float. [11]

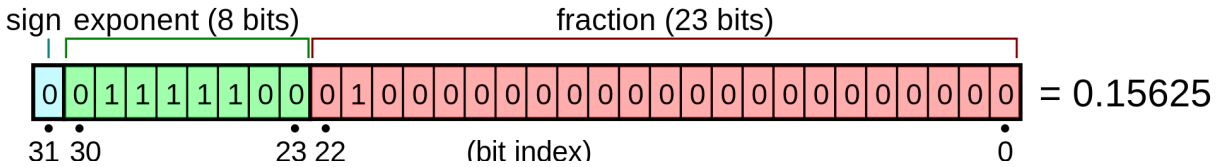

Figure 2.4: Single Precision Floating-Point Standard representation [11]

This works well for small values, where entities can be represented precisely; but precision decreases as values increase because of floating-point rounding errors. As a result, game objects' positions that are rendered far from the origin aren't represented accurately (they're slightly misplaced). The effect becomes more pronounced the further the gameplay

moves from the origin, and it's most noticeable during active events, like movements and animations. So for example, instead of the player moving smoothly through the scene, their movement appears jittery. [18]

Everything inside the 3D game scene is represented by a vector of three floats (x-axis, y-axis, z-axis). Each of these floats represent a point on a separate axis, and those are in a single-precision floating point format. The 23 fraction bits give us around reliable 7 digit precision to work with. Which means our float point on an axis will be represented most accurately with 7 significant digits, with a decimal place anywhere within those 7 digits. The origin is a point with floats in origin on each axis (0, 0, 0). The closer our objects are to the origin in the scene, the more accurate our objects and their movements will be represented. Let's take an example p = (1.234567, 1234.567, 123456.7). If one unit is a meter, the float point on the x axis will have a precision of a micrometer, the float point on the y axis will have a precision of a millimeter, and the float point on the z axis will have a precision of a decimeter, which isn't very helpful if we want to move smoothly. With this system, the further away from the origin (0, 0, 0), the more precision is going to be lost. The degradation of accuracy becomes an obvious problem when we want to work at a small scale. Things become strange at a far distance like 123456.7 because objects become more and more difficult to be represented.[19] There's flashing, and stuttering, shape and appearance of objects can degenerate. Because of this, we can't just scale everything up by simply assigning huge numbers. That's a lazy option, but only if we could work with immense objects. If we wanted to render a small object like a ship, as well as a large object like a planet that would be an even trickier challenge.

There are also camera issues on a larger scale. A camera has a Near and Far clipping plane, which defines the View Frustum. Everything which falls within the View Frustum will be rendered, and anything else outside this range will not. To render something really small, we would have to be able to bring the camera really close to the object before clipping gets in the way. If one wants to render something really big like a planet, they would set the far clipping plane enough to encompass the object. If we try to set a really close Near clipping plane and a too large Far clipping plane, multiple issues like polygon intersecting and flickering, which is a similar problem to the floating-point accuracy on the coordinate system, but applied to a camera.[19]

Nevertheless, games with large-scale worlds exist. How do they represent big and small objects in the same frames? Like many things, this is possible, but not the way normally anticipated. We have to work around this floating-point somehow and create an

illusion.

2.4.2 Multiple cameras - solution

When created, a Unity scene contains just a single camera and in most situations, this is perfectly appropriate. However, we can have as many cameras in a scene as needed and their views can be combined in different ways. By default, a camera renders its view to cover the whole screen and so only one camera view can be seen at a time. By disabling one camera and enabling another from a script, we can “cut” from one camera to another to give different views of a scene.[20] This is perhaps desirable, for example, when we want to switch between an overhead map view and a first-person view. But there’s more, multiple cameras can be active at the same time.

The solution to the problem presented earlier, is to take advantage of the ability to have more than one camera render to the player’s viewport, with each camera layered on top of each other. A first, probably evident course of action, would be to set the view frustum of two cameras to certain values in order to combine into one large view frustum. But the complexity doesn’t stop here. We can significantly scale everything in the game world down a much smaller, manageable space. In addition, each camera can represent objects of different scales as follows: one camera renders large far away objects, and another one renders small near objects. The cameras are part of a level of detail object, each having their own scale factors. With scale factors we can represent one unit inside the game engine as any amount of distance. Naturally speaking, one unity can be equivalent to one meter, so the ratio would be 1:1, which can be changed. In the same way it can be said that one unit in Unity is equivalent to one hundred meters or one kilometer (1000 meters) and so on. And that can be different for each level of detail which contains a camera. The camera that renders distant, huge objects are part of a level of detail where one unit is equivalent to 1 kilometer (1:1000 ratio), and small objects are part of a level of detail where one unit is equivalent to 1 meter (1:1 ratio). This way, the cameras’ positions are multiplied by their parent objects’ scale factors in order to move proportional to the scale, thus making everything in their scope look much bigger, including the distance in between objects.

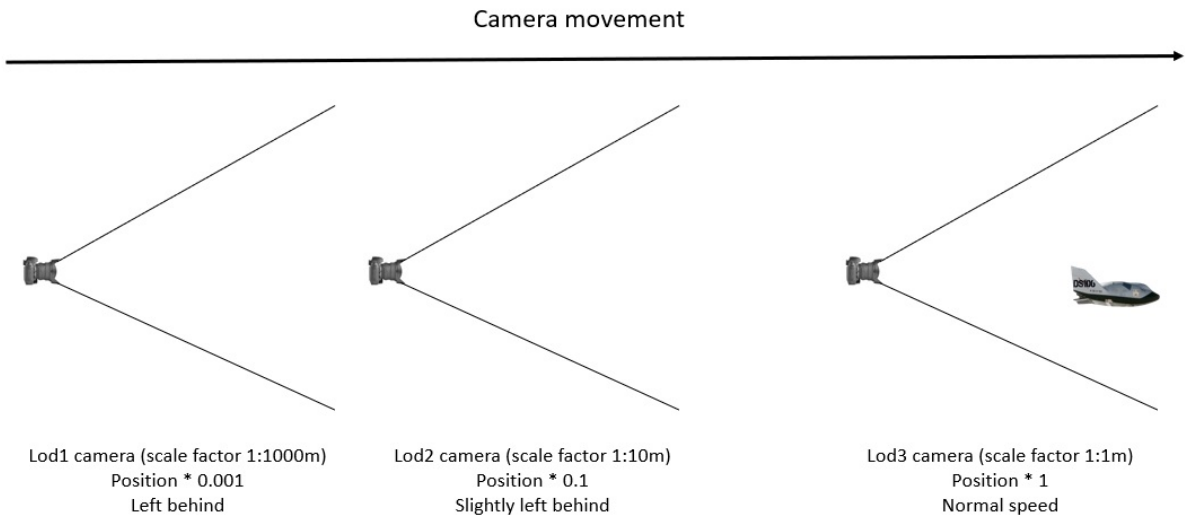

Figure 2.5: multiple layered cameras movement in relation to each other

Various adjustments may be made, depending on the specifications of the application, such as having the multiple cameras follow one main camera that follows the user's viewpoint. With this trick, the bigger objects in the scene can be scaled down to much smaller items, having everything close to the origin and solving the floating-point precision issue.

2.4.3 Floating Origin approach - solution

With multiple cameras, we can solve the representation issue. But we still have to travel immense distances in order to reach other objects (stars, planets etc.). That would mean the character and camera that we are controlling will inevitably get further and further away from the origin, which furthermore entails jittering. What can we do to fix this?

The concept of "Floating Origin" refers to the origin of the world being reset. What does this mean? We want to be as close to the origin as possible with our player camera. We can achieve this by translating all objects in the world to keep the camera near the origin. It's possible to detect when the camera is further than a fixed 'threshold' units from the origin, at which point all objects in the scene can be moved in such a way that the camera is back at the origin.[21]

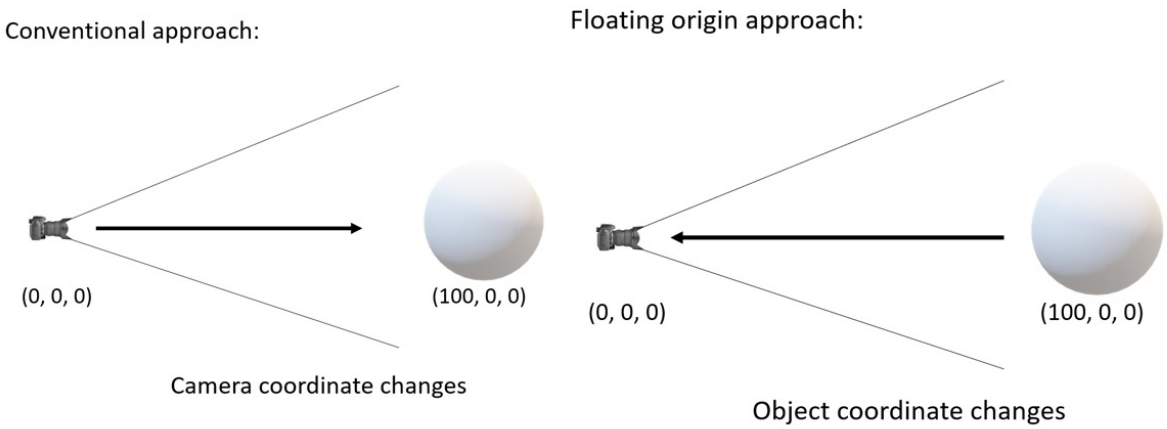

Figure 2.6: conventional movement approach (left) vs floating origin approach (right)

The Floating Origin approach is a divergence from conventional navigation thought process. Instead of allowing the viewpoint to move in the world, the world is reverse transformed in order to position the desired point at the origin. The origin is not fixed relative to the space we are working in but floats with the viewpoint. This way, the viewpoint is consistently centered to the highest fidelity place, even when freely navigating. [22] These are the main benefits of using the floating origin approach:

- Elimination of observable stutter, jitter, degradation of accuracy etc.;
- Lowering design complexity;
- Provision of constant, optimum fidelity of motion and rendering compared to the continually variable fidelity of conventional approaches [22];
- Not discussed here, but also lowers possible processing overhead [22];

Floating point suits the centric-view nature of computer graphics: the local region around the camera viewpoint is what is visible to the user and that is where most of the accuracy of rendering and motion needs to be. A floating origin allows this natural benefit of floating point to be applied to the rendering of every frame. When combined with level of detail streaming and dynamic placement a floating origin optimizes the quality of motion, interaction and rendering throughout large, continuous virtual worlds by minimizing the various problems existent otherwise without this approach. [22]

2.4.4 Optimization

Another issue that presents itself in most video games, especially those with long distance and large-scale representations is optimization. We can't simply render all objects without noticing significant drop in FPS (frames per second) and increasing requirement of resources.

This has a straight forward solution: disable objects (altogether or just their rendering) that are far away enough from the viewpoint so that the visual experience isn't affected excessively. Unnecessary entities' performance-consuming display and/or functionality will be stopped. This way, performance will be enhanced.

2.5 Procedural content generation

2.5.1 Overview and context

Procedural generation is a method of creating data algorithmically rather than manually.[23] Procedural content generation (PCG) studies the algorithmic creation of content (e.g. maps, textures, meshes etc.), often for video games. In computer graphics, it is also called random generation and is commonly used to create textures, 3D models, and populate worlds in various ways.[24] Procedural content generation (PCG) is the programmatic generation of game content using random or pseudo-random processes that result in an unpredictable range of possible gameplay spaces.[25] Advantages of procedural content generation include smaller file sizes, larger amounts of content, and randomness for more exploration and less predictable gameplay. There are also disadvantages to this approach, which include exponentially higher algorithm complexity as the demand for detailed worlds and gameplay increases.[24]

The term "procedural" refers to the process that computes a particular function, which means that the generation is handled automatically by an algorithm, by a procedure, instead of manually.[26]

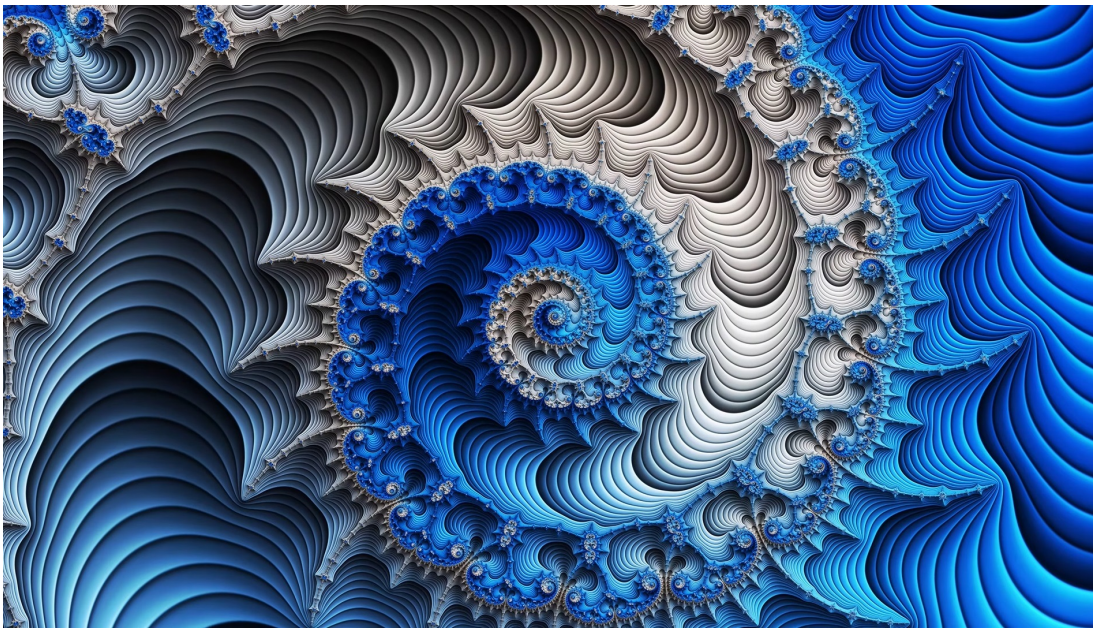

Figure 2.7: Manderlbrot set zoomed-in visualization[51]

Fractals, for example, are geometric patterns which can be generated procedurally. Commonplace procedural content includes textures and meshes.[24] But in theory, almost anything can be procedurally generated. While software developers have applied this technique for years, few products have employed this approach extensively. Procedurally generated elements have appeared in earlier video games: Spore features procedural content generation as a central mechanic. All entities in the game will be generated by other players using the in-game editors. Avalanche Studios[27] used procedural generation to create a large and varied groups of detailed tropical islands for Just Cause. No Man's Sky, a game developed by games studio Hello Games[28], is all based upon procedurally generated elements.[24]

The ability to automatically and to a degree, randomly, create of objects of different sizes, of different shapes and colors; constructing objects through different combinations of elements; populating scenes with varied amounts of entities; and many other scenarios are all included in the procedural content generation concept. It provides an immense advantage to the arguably tedious manual alternative approach.

Spore – character editor

No Man's Sky – environment and creatures

Figure 2.8: Spore procedural character editor (left)[52] and No Man's Sky procedural environment (right)[53]

2.5.2 Random Seed

A random seed is a number (or a vector) used in computer programming to initialize a pseudorandom number generator or a deterministic random bit generator. The number specifies the start point when a computer generates a random number sequence. [29]

A pseudorandom number generator is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers, which means everything is already predicted, but it resembles randomness, hence "pseudo".

A pseudorandom number generator's number sequence is completely determined by the seed. This means computers don't generate truly random numbers. Random generator algorithms always operate by a set of reules, mimicking but unable to completely replicate randomness. Thus, if a pseudorandom number generator is reinitialized with the same seed, it will produce the same exact sequence of numbers. [30]

Random seeds are often generated from the state of the computer system (such as the time), a cryptographically secure pseudorandom number generator or from a hardware random number generator.[29]

2.5.3 Procedural meshes

Procedural terrain generation is a widely explored field, especially in movies and video games. It is more and more essential nowadays for such applications to generate visually

appealing and detailed, rich landscapes. The terrain can be manually produced or entirely loaded by an input data. However, these approaches are not scalable for extremely large terrains, or for large numbers of unique representations of those terrains on multiple objects.

What is a mesh?

A polygon mesh (synonymous with "grid") is a collection of vertices, edges and faces that defines the shape of a 3D polyhedral solid represented in computer graphics and modeling. The faces usually consist of triangles (triangle mesh), quadrilaterals, or other simple convex polygons, since this simplifies rendering, but may also be composed of more general concave polygons, or polygons with holes.[31]

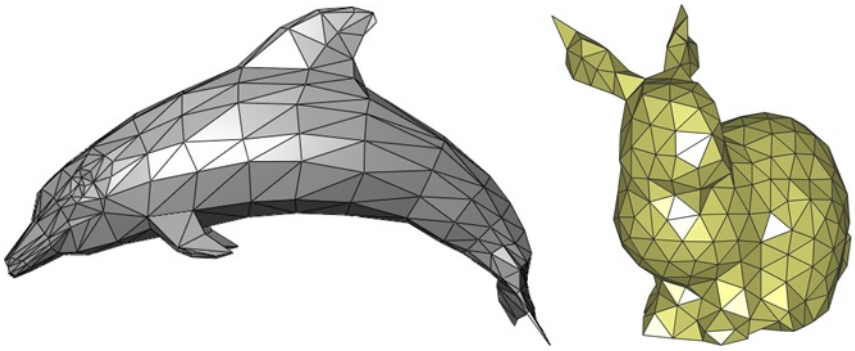

Figure 2.9: 3D object meshes[54][55]

The study of polygon meshes is a large sub-field of computer graphics and geometric modeling. Different representations of polygon meshes are used for various applications and goals. There is a wide variety of operations performed on meshes.[31]

Mesh generation

Mesh generation is the practice of creating a mesh. This process consists of algorithmical calculations which result in the subdivision of a continuous geometric space into discrete geometric and topological cells. Often these cells form a set composed of points, line segments, triangles, and their n-dimensional counterparts (simplicial complex). Mesh cells are used as discrete local approximations of a larger domain. Meshes are created by computer algorithms, often with human guidance, depending on the complexity of the domain and the type of mesh desired. The goal is to create a mesh that accurately captures the input domain geometry, with high-quality (well-shaped) cells. The mesh should

also be fine (have small elements) in areas that are important for the subsequent calculations.[32][60]

Meshes are used for rendering to a computer screen and for physical simulation. Meshes are composed of simple cells like triangles, because we know how to perform operations such as finite element calculations (engineering) or ray tracing (computer graphics) on triangles, but we do not know how to perform these operations directly on complicated spaces and shapes. We can simulate more complex spaces and shapes, or draw it on a computer screen, by performing calculations on each triangle and calculating the interactions between triangles.[32][60]

Sphere generation

Most of the geometric primitives like points, lines, triangles, planes, pyramids, cubes are perfectly representable in modern graphics hardware. When graphics programmers face the problem of creating a mesh for a sphere, trade-offs must be made between quality and construction, memory and rendering costs. There are four different methods simple and appropriate enough for this project, and their characteristics can be analyzed and compared to allow programmers to make an informed decision on which method suits their needs.[33]

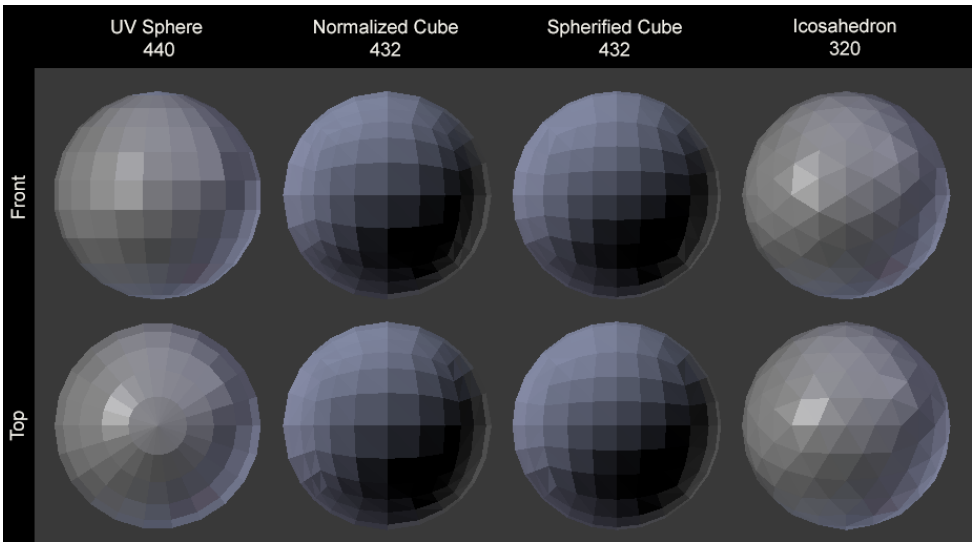

Figure 2.10: 3D sphere meshes[33]

Standard Sphere. This is the most common implementation of a sphere mesh and can be found in almost any 3d toolset. We can find it under the name of “UV sphere” in blender or just “sphere” in 3d max. This method performs a division on the sphere using meridians (lines from pole to pole) and parallels (lines parallel to the equator). It produces a

mesh with bigger polygons near the equator and smaller ones close to the poles. The faces are of triangles, at the poles, and quads.[33]

Normalized Cube. This method uses a uniformly subdivided cube where each vertex position is normalized and multiplied by the sphere radius. This creates a non uniformly subdivided sphere where the triangles closer to the center of a cube face are bigger than the ones closer to the edges of the cube.[33]

Spherified Cube. This method is based on a subdivided cube as well but it tries to create more uniform divisions in the sphere. The area of the faces and the length of the edges suffer less variation, but the sphere still has some obvious deformation as points get closer to the corners of the original cube.[33]

Icosahedron. An icosahedron is a polyhedron composed of 20 identical equilateral triangles and possesses some interesting properties: Each triangle has the same area and each vertex is at the same distance from all its neighbours. [33]

The normalized cube method proved to be best suited for this project, because it's not excessively complicated to achieve and reasonably optimal. The basic idea is to take a simple 3D object, such as a cube, and then divide the surfaces - into triangles. After division, we "push" all the vertices in the direction of their normals (out of the center of the object) so that they're all the same distance from the center.[34]

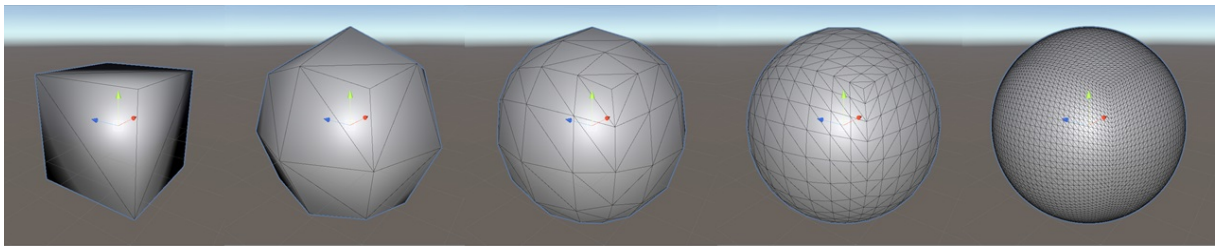

Figure 2.11: 3D spherified cube with different number of triangles (resolution)

Repeating this process a few times turns any of those simple objects (tetrahedron, cube, octahedron etc.) into a sphere.[34]

To get a higher number of triangles (or a higher resolution) we need to subdivide each triangle into four triangles by creating a new vertex at the middle point of each edge which is then normalized, to make it lie in the sphere surface. Sadly this breaks the initial properties of the icosahedron, the triangles are not equilateral anymore and neither the area nor the distance between adjacent vertices is the same across the mesh. An added problem with this method is that we can only increase the number of faces by four each time. But it is still

a better approximation by almost any measure excluding its number of triangles.[33]

Using layered noise to generate procedural meshes

Gradient noise is a type of noise used in the creation of procedural texture primitives in computer graphics.[36] The first form of implementation of a gradient noise algorithm was Perlin noise, credited to Ken Perlin, who introduced the concept and published its documentation in 1985. Later developments were Simplex noise and other variations.

According to Ken Perlin, the purpose was to develop a technique used to produce natural appearing textures on computer generated surfaces for motion picture visual effects.[35] The development of Perlin Noise has allowed computer graphics artists to better represent the complexity of natural phenomena in visual effects for the motion picture industry, enabling automatic generation of detailed surfaces.[37]

Simplex noise is a method for constructing an n-dimensional noise function very similar to Perlin noise but with fewer directional artifacts and, in higher dimensions, a lower computational overhead. Ken Perlin designed the algorithm in 2001 to address the limitations of his classic noise function, especially in higher dimensions.[38]

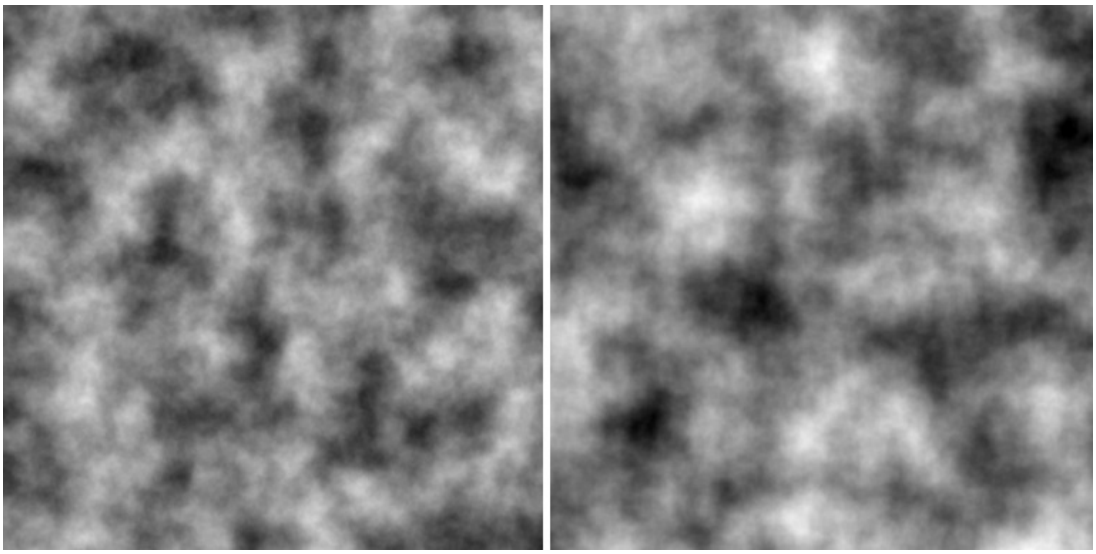

Figure 2.12: simplex noise (left) vs classic Perlin noise (right)[56]

Noise by itself however, is just a bunch of numbers. There's is no meaning assigned to it. The noise can be used as a map to generate various configurations of triangles with which different looking meshes can be obtained. The first thing to think about is to make the noise correspond to the elevation of a mesh (also called a "height map").

Secondly, noise can be generated at any frequency. Lower frequencies make wider portions (or "hills") and higher frequencies make narrower portions ("hills"). Frequency describes the horizontal size of the features. And by combining noise functions together at different frequencies we get octaves. These coherent-noise functions are called octaves because each octave has, by default, double the frequency of the previous octave.[39]

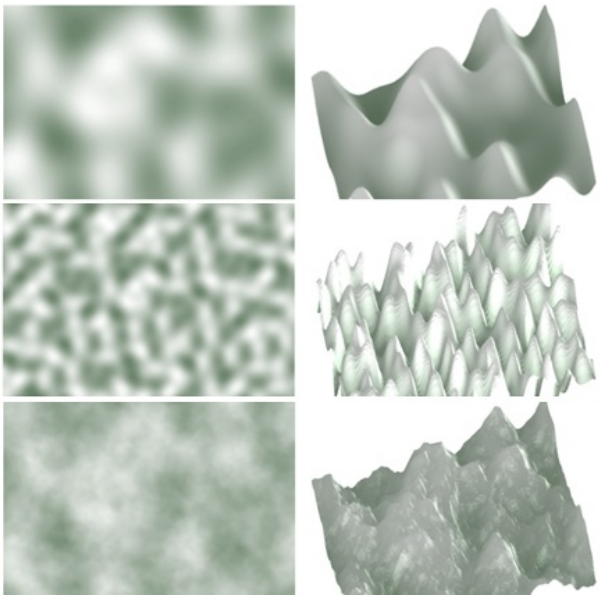

Figure 2.13: terrain maps with different noise values[39]

But it doesn't stop here. There are endless possible ways to manipulate a mesh using noise functions, and there can be layers upon layers of noise. This process allows for a random generation of terrain which has a convincingly natural look and feel. And this can be applied in such a way that the creation of something resembling a planet can be achieved.

3. Technologies

3.1 The Unity game engine

Unity is, arguably, the world's most popular game engine. It's free and it comes with a ton of features that make everything easier, which is why it's considered the best option for starting developers. The C# scripting and built-in Visual Studio integration attracts many programmers, although it supports JavaScript and MonoDevelop as an alternative. Unity offers artists powerful animation tools that make it simple to create and build 3D scenes from scratch.[40]

Figure 3.1: Unity logo[14]

Because of the beginner-friendly, easily accessible features Unity brings to the table, it proved itself ideal for this project.

3.2 Structure

Unity provides a complete integrated development environment (IDE) with an integrated editor, asset workflow, scene builder, scripting, networking and more. It also has a vast community and forum where anyone who wishes to know and learn to use Unity can go and have all their questions answered. The five main views used in the Unity editor to get all the work done are the project view, scene view, game view, hierarchy view and inspector view.

The scene view is one of the most used views since this is where all the game objects are located and where scenes for the game are built. The game view is everything the user will see when the game is started. Unity provides an extensive structure on which simple and complex applications can be built. The hierarchy view reveals the architecture and placement of every object in the game. Each object can be created, accessed, grouped and manipulated to make the game. The project view is where all the scripts and scenes can be accessed from. This view is exactly like a file explorer on Windows or Mac that allows creating files and folders in order to help organize the projects assets. Finally, the inspector view is where all the physics and properties of the objects are stored and accessed from. Every game object has a transform; this is what holds attributes of the object such as position, rotation, and scale. Other properties are the physics affecting the object, textures to load on the object and sound etc.[41]

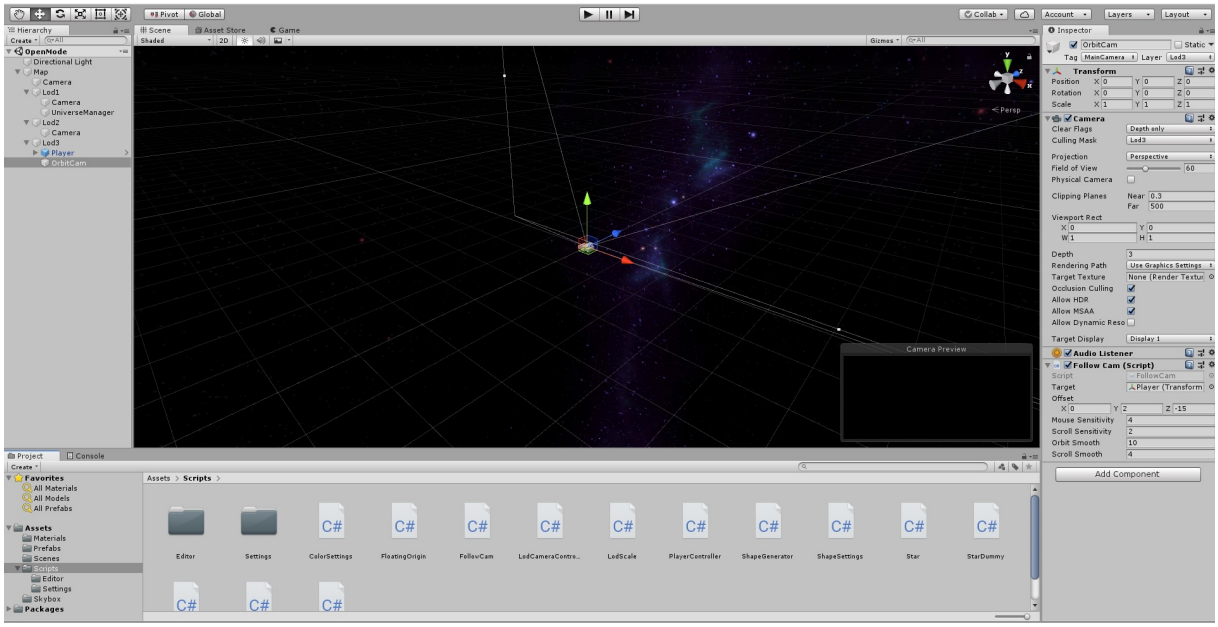

Figure 3.2: Unity's User Interface

Unity contains the powerful NVIDIA PhysX Physics Engine. With this phsycics engine built into Unity various powerful assets can be added to the game. Most video games played today are composed of common game mechanics like path finding, collision detection and input. These mechanics have been around now for decades and have been improved on throughout the years.

Much of the power of Unity resides in its rich scripting language, C#. It can be used to handle user input, manipulate objects in the scene, detect collisions, create and instantiate

new GameObjects and cast directional rays around the scene to help with the game logic. Unity is a great beginner tool (as well as for advanced users) because it has revealed well-documented APIs that explain and ease the developer's understanding of its assets, tasks, and libraries.[42]

Gameobject is the base class for all entities in Unity Scenes. It comes with its set of properties, constructor, a multitude of methods. Among other useful attributes, it hold the notable transform property, which give developers access to the objects, position coordinates, and rotation and scale values, and many other useful functions. Some of the most commonly used functions are[43]:

- GetComponent (and its related methods) - Returns the component of Type type if the game object has one attached, null if it doesn't;
- Find (and its related methods) - Finds a GameObject by name and returns it;
- Instantiate - Clones the object original and returns the clone;
- CreatePrimitive - Creates a game object with a primitive mesh renderer and appropriate collider;

MonoBehaviour is the base class from which every Unity script derives, and it must be explicitly derived when inteded. It offers some life cycle functions like Start(), Update(), FixedUpdate(), LateUpdate(), OnGUI(), OnDisable(), OnEnable(); that are make developing easeier for the user in various ways. In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes.[45]

3.3 Particle Systems

There are various entities in games, other than solid objects, lights, and generally object that have meshes. These are fluid and non-physical in nature and thus difficult to portray using meshes or sprites (2D graphic objects). For effects like flowing liquid, smoke, fire, lasers, and magic spells, a different approach known as particle systems are more appropriately used.[44]

A particle system emits particles in random positions within a predefined space, which can have a shape like a sphere or a cone. The system determines the lifetime of the particle itself, and when that lifetime expires, the system destroys the particle.[46]

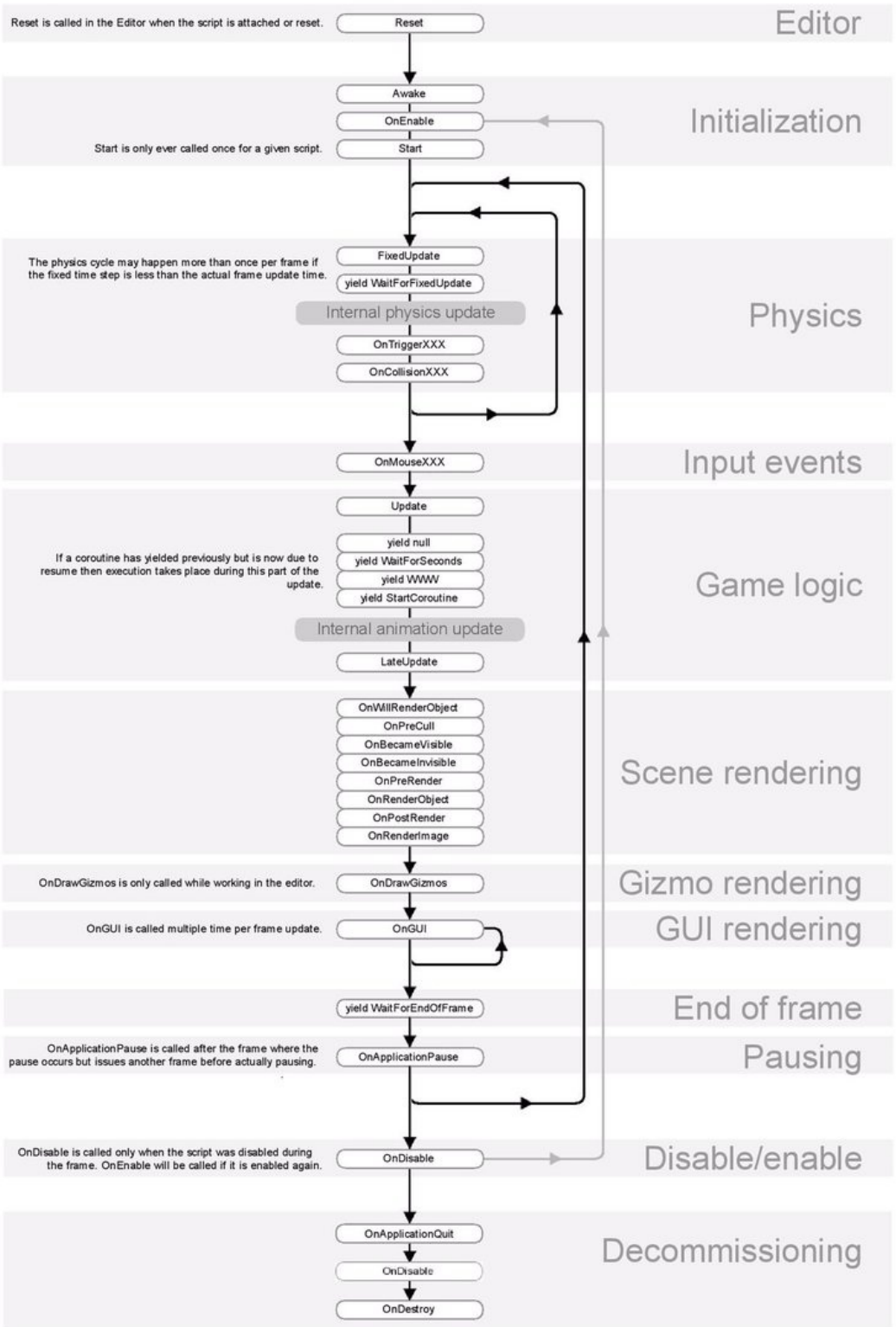

Figure 3.3: flowchart with order of Execution for Event Functions in Unity[57]

3.4 Component-based architecture

While its programming design is fundamentally object-oriented, the Unity workflow strongly builds around the structure of components, which requires a component-based thinking.[47]

A component can be conceptualized as a smaller piece of a larger machine, small piece in a puzzle. Each component has its own task, and can generally accomplish its job or purpose without the help of outside third party sources. Components rarely belong to a single structure, and can be joined with various systems to accomplish their specific task, but achieve different results when used in different contexts. This is because components are completely unaware about the bigger machine they are part of, they don't even know it exists.[47]

Unity was built with components in mind. One of the most valuable and distinctive aspects of Unity is that it's a very visual program.[47]

Unity provides a real time interface where the user can see everything they are working on. Testing a project, running a project, editing the code or game objects all have their separate panels that are visible to the user to freely operate on and witness the changes live. This system gives the user a substantial amount of power, which can be considered as an essential aspect of modern game development. All of this is accessible and made possible by Unity's component-based architecture.[47]

An important element of the Unity's interface is the Inspector. This is a panel that showcases all of the properties of a game object. Any time the user click on an object in the scene (either at run-time, or before) they will be able to see everything about that object. If that object has five components on it, each will be listed in a separate tab, and every public variable will be available for the user to see, and modify. The inspector changes and updates even as the game is running according to what happens in the game: tabs can be updated, removed, or added.

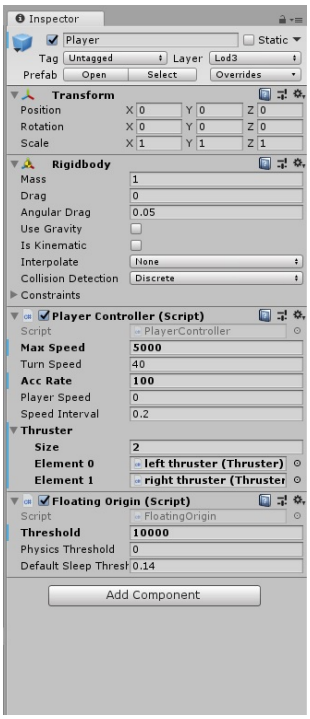

Figure 3.4: Unity's Inspector Interface

3.5 Rendering

3D rendering is the process of producing an image based on three-dimensional data stored on the computer. It's also considered to be a creative process, much like photography or cinematography. With 3D rendering, computer graphics converts 3D wireframe models into 2D images with 3D photorealistic, or as close to reality, effects. Rendering can take from miliseconds to even days for a single image or frame. There are two major types of rendering in 3D and the main difference between them is the speed at which the images are calculated and processed: real-time and offline or pre-rendering. In real-time rendering, most common in video games or interactive graphics, the 3D images are calculated at a very high speed so that it looks like the scenes, which consist of multitudes of images, occur in real time when users interact with the game. Interactivity and speed play important roles in the real-time rendering process, because everything happens so fast. Unity uses real-time rendering as its 3D rendering process.[48]

In 2018.1, Unity introduced a new system called the Scriptable Render Pipeline (SRP), allowing the user to create their own rendering pipeline, based on the needs of the project. SRP includes two ready-made pipelines, called Lightweight (LWRP) and High Definition

(HDRP). HDRP aims for high visual fidelity and is suitable for PC or console platforms. [49]

The High Definition Render Pipeline (HDRP) is a high-fidelity Scriptable Render Pipeline built by Unity to target modern (Compute Shader compatible) platforms. [50]

Volume Settings allow the user to visually alter environment preferences, adjusting elements such as the Visual Environment, Procedural Sky and HD shadow settings. This also enables the user to create custom volume profiles and switch between them. [49]

Lighting in HDRP uses a system called Physical Light Units (PLU). PLU means that these units are based on real-life measurable values, like what can be seen when browsing for light bulbs at the store or measuring light with a photographic light meter. [49]

Unity provides an extensive structure on which simple and complex applications can be built. The hierarchy menu reveals the architecture and placement of objects inside the scene. Scripts can be attached to objects in the hierarchy view.

4. A virtual universe simulation application

This chapter contains the presentation of the game application. Details about the application's structure and architecture, specifications for the creation of the virtual large-scale world, and description functionalities.

4.1 Overview

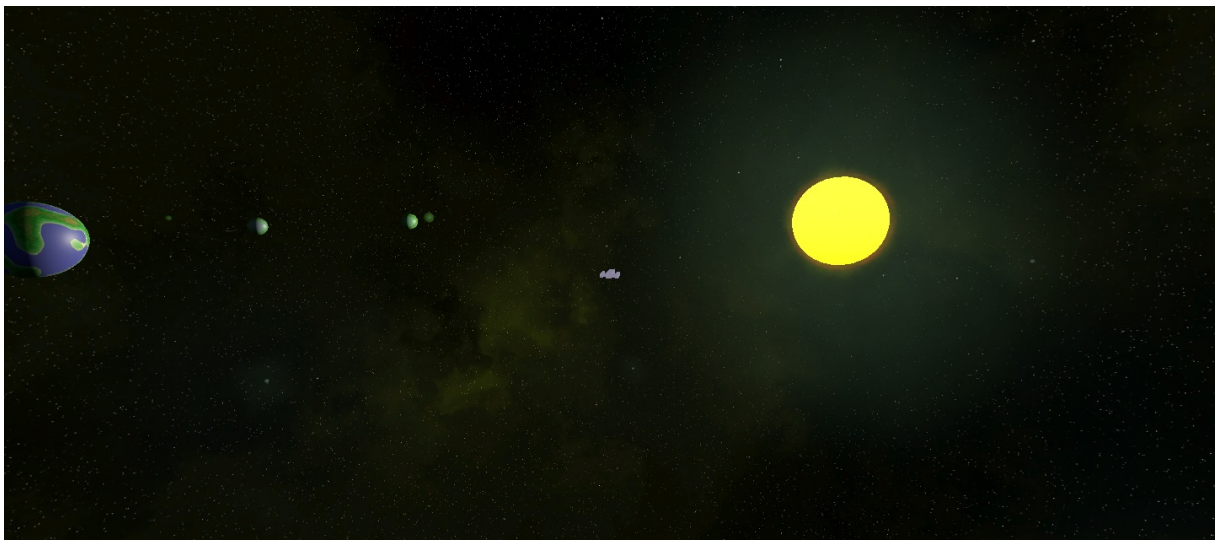

Figure 4.1: Beyond Light cover

Beyond Light is a space simulation game, providing the user a realistic visual experience of space exploration in a virtual universe, where the concepts like large-object scaling, floating origin, and procedural content generation discussed earlier in theory can be put in practice and implemented. The user will be able to navigate through space with a spaceship and discover and experience realistic representations of celestial objects, such as stars, planets, asteroids etc.

Although it misses some essential "game-like" elements, the application is called a game, since it is developed in a very similar manner to that of a game, and it is intended to work as a video game in the future. For most simulators developed in game engines like Unity, it is universally accepted to be called simulation games.

A large-scale virtual universe is procedurally created once an instance of the game is initiated. The space is of a limited size, and it contains stars of different sizes, colors, of

a set minimum distance between them, on random positions. Before each instance of the game, values like the number of stars, the radius of space, the minimum and maximum size of each star, the minimum and maximum amount of planets that orbit a star, and a seed number.

Each star has a random number of planets in its orbit. Each of the planets performs an orbital rotation around the star. A planet is procedurally generated so that each planet can vary in size, colors, and terrain. The terrain of a planet is procedurally generated so that every planet is unique regarding its terrain, with varied elevation on the surface.

The user is presented with a view of their spacecraft, and is free to navigate through the virtual universe to explore with the help of a controller.

4.2 Architecture

4.2.1 Scene structure and assets

The Hierarchy consists of all the objects which represent entities in the game. One scene is used: the spaceship, stars, planets and asteroids, all contained inside levels of detail (LODs) with multiple cameras layered on top of each other in order to give the user a free and open space universe to discover and navigate through.

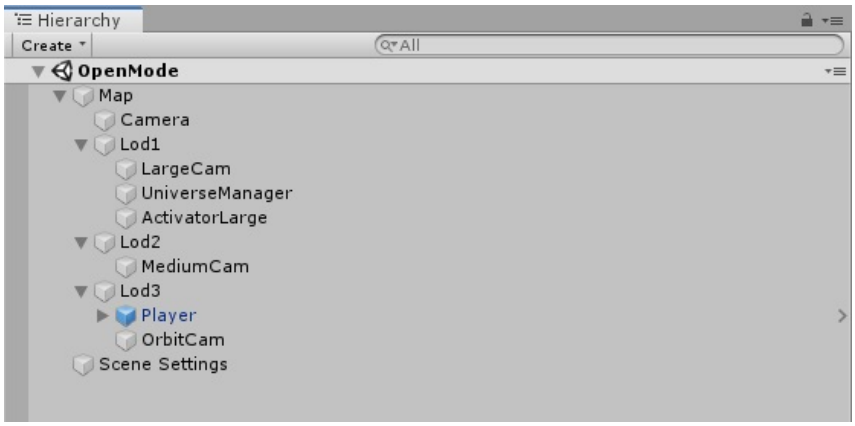

Figure 4.2: Scene object hierarchy view

There are 3 LODs contained in a main object called "Map", each with its own camera. The main object's camera renders the Skybox of the scene (the night sky). Each LOD is assigned a layer and a scale factor, and their cameras' viewports are rendered in a specific depth order, such that the view of the player-following camera (OrbitCam) consists

of multiple layers of rendering, having the result of expressing the illusion of a cosmic space with large and small objects.

The first LOD (Lod1) represents the large scale (1:100), containing the UniverseManager, which has the purpose of generating the stars and the planets, it's corresponding camera, and an Activator object which disables objects that are further than a threshold away, in order to optimize performance. The second LOD (Lod2) represents the medium scale (1:10), with its corresponding camera. The last LOD (Lod3) represents the small scale (1:1), where we can see the player's ship and the camera that follows the ship. Note that these ratio values can be changed if the scale needs to be adjusted to represent larger or smaller spaces.

Some game objects aren't visible in the scene hierarchy, since they are created automatically at runtime, which is when the program is running.

The entirety of the assets consists of Materials, Prefabs, C# scripts, and other packages that are used in this project.

Almost every object has one or more scripts attached to it, with the purpose of implementing a certain functionality.

GameObject	Script
Map	LodCameraController.cs
Lod1, Lod2, Lod3	LodScale.cs
UniverseManager	Universe.cs
Player prefab	PlayerController.cs, FloatingOrigin.cs
OrbitCam	FollowCam.cs
StarClose in Star prefab	Star.cs
Star prefab	DisableIfFar.cs
Planet prefab	Planet.cs
ActivatorLarge	Activator.cs

Figure 4.3: Object - Script relations

4.2.2 Creating the world

The Universe Manager

The UniverseManager object located in the first LOD is responsible for generating the space and the objects inside. The Universe.cs script is attached to this object. It is inherited from MonoBehaviour and it handles the whole creation process in its Start() function. It receives as properties a star and a planet of type GameObject, which are actually the Star and Planet prefabs. Other serializable fields are the minimum size of stars, maximum size of stars, the maximum radius in which everything is created, the seed number, the minimum and maximum distance between the created stars, and the minimum and maximum number of planets each star has.

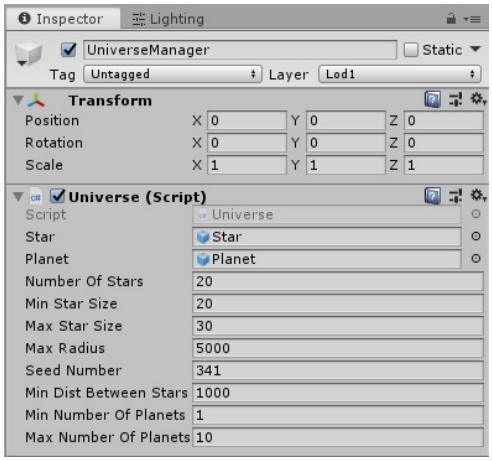

Figure 4.4: UniverseManager Inspector interface

Firstly, the Random.InitState() function is called to initialize the random number generator state with a seed nubmer. This way, any fixed configuration of randomly placed star objects in the virtual universe can be accessed.

The next step is to enter a loop for generating each star and its planets. Each iteration produces a new random position within the maximum radius (Max Radius) assigned in the Inspector by converting random spherical coordinates into cartesian coordinates. After calculating a random size for the star and the number of planets it has, we create an array of colliders that overlap with the current hypothetical star "range". This range is defined by the sum of its radius and the minimum distance between stars property (Min Dist Between Stars) assigned in the inspector. This is necessary so that multiple stars aren't created too close or on top of each other. After making sure that nothing collides with the current star

range, we can proceed to actually create it. A fail count is needed to check if the minimum distance between stars is too big, so that the program won't enter an infinite loop and crash. When the function to create the star is called, the StarSystem prefab is instantiated inside the first LOD (Lod1), at the random position produced earlier, with the respective size. Similarly, this process is repeated for generating the planets for every star. But the scale problem still isn't solved.

The star and planets are created inside the StarClose component of the StarSystem prefab.

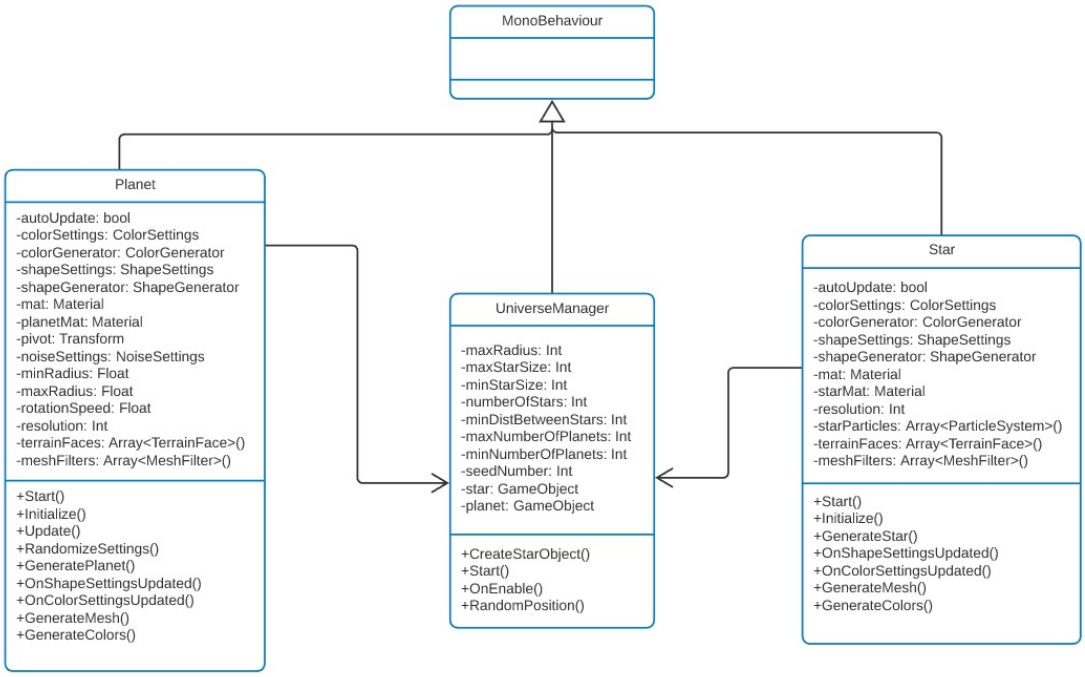

Figure 4.5: UniverseManager class diagram

Level of detail camera controller

Each of the LODs inside Map have a Lod scale script, which simply sets the scale factor of one unit in that LOD. Lod1 has a scale ratio of 1:0,01, which means 1 unit is equal to 100 'meters'. Lod2 has a scale ratio of 1:0,1, which means 1 unit is equal to 10 'meters'. Lod3 has a scale ratio of 1:1, which means 1 unit is equal to 1 'meter'. The object that contains everything (Map) has a Lod Camera Controller Script attached to it. This script handles all the cameras inside each LOD like so: It gathers all the cameras and their respective LODs' scale factors into separate lists, and in a LateUpdate() function it scales down the position of the cameras by dividing the position of the camera to the scale factor

of its LOD, so that a camera in a "bigger" LOD moves slower. Thus, objects generated by UniverseManager in a larger LOD appear much bigger. For this it's necessary to set for each camera a specific culling mask and depth, so that unintended entities appear in the game view. All of this occurs while every camera simultaneously follows the OrbitCam, which follows the player around.

```
// Update is called once per frame
void LateUpdate()
{
    Position = follow.transform.position;
    Goal = follow.transform.position / GameObject.Find("Lod3").GetComponent<LodScale>().ScaleFactor;


    for (int i = 0; i < cameras.Count; i++)
    {
        Vector3 scaledPos = Position * scales[i];
        Vector3 scaledGoal = Goal * scales[i];

        if((scaledPos - scaledGoal).magnitude > 200)
        {
            cameras[i].transform.parent.gameObject.SetActive(false);
        }
        else
        {
            cameras[i].transform.parent.gameObject.SetActive(true);

            cameras[i].gameObject.transform.position = scaledPos;
            cameras[i].gameObject.transform.rotation = follow.transform.rotation;
        }
    }
}
```

Figure 4.6: scaling cameras with their respective scale factors code

The LodCameraController script also checks for a distance relative to a goal. If the distance is below a set number, the objects in the respective LOD will be disabled. If the distance goes back above the set number, the objects in the respective LOD will be enabled.

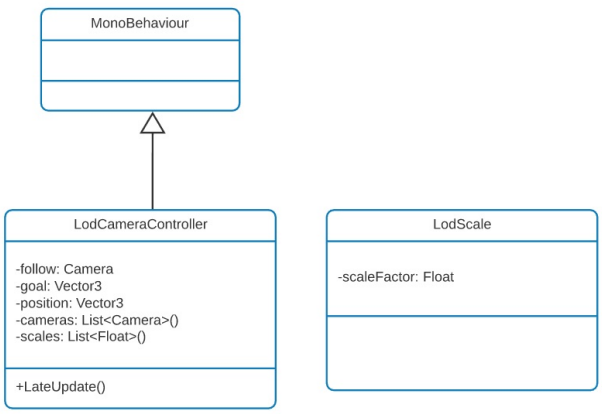

Figure 4.7: LodCameraController class diagram

Floating origin

While the scale problem is solved with the LodCameraController script, a new issue appears. The user controlling the ship and main camera (OrbitCam) can move the spacecraft up to tens and hundreds of thousands of units, while still seeing the objects contained in a much smaller radius that are scaled up to appear bigger. The problem is that only the camera in the largest LOD doesn't go beyond the maximum universe radius, because it moves very slow across the maximum radius. Everything else in other LODs can move way past that. This makes is so as the player moves further away from the origin (0, 0, 0) the less accurate things get represented, because of the single-precision floating-point format Unity uses. The spaceship controlled by the user will appear jittery after reaching distances past 10000 - 50000 units.

```
if (camPosition.magnitude > threshold)
{
    GameObject[] gos = GameObject.FindObjectsOfType(typeof(GameObject)) as GameObject[];
    foreach (GameObject go in gos)
    {
        if (go.transform.parent && go.transform.parent.CompareTag("Level3"))
        {
            go.transform.position -= camPosition;
        }

        if (go.transform.parent && go.transform.parent.CompareTag("Level2"))
        {
            go.transform.position -= camPosition * GameObject.Find("Lod2").GetComponent<LodScale>().ScaleFactor;
        }

        if (go.transform.parent && go.transform.parent.CompareTag("Level1"))
        {
            go.transform.position -= camPosition * GameObject.Find("Lod1").GetComponent<LodScale>().ScaleFactor;
        }
    }
```

Figure 4.8: Floating Origin code

The Floating Origin script attached to the Player prefab fixes this by resets the position of the player and everything else in the world such that the player (space ship and camera) are located in the origin again. For this to be possible, it's necessary to synchronize everything around the player (the stars and planets). Thus, every object in the game must be reset according to the player's position. This takes place, every time the player passes a certain distance threshold from the origin. This is achieved by iterating through every object in the game and subtracting the distance the player currently is from the origin from its position. Because objects are in different LODs, the distance between the player and the origin must be multiplied with that object's LOD scale factor, in order to achieve the desired effect with accuracy. This resets everything back to origin, and no loss of accuracy due to

floating point representation is suffered. Degradation of floating point representation and jitter is fixed. Similarly, this process is repeated for particles as well.

4.2.3 Player movement and camera control

The user controls the Player object, which is actually a spaceship prefab, with the help of the PlayerController script attached to the object. This script allows the user to accelerate, decelerate until stop, turn the ship, roll, and pitch. The spaceship accelerates, with an accelerating rate, until a maximum speed is reached. Once the user hits the accelerating button, the ship will start moving and adds to the speed a certain accelerating rate at a short interval until the user stops pressing the accelerating button. Once the accelerating button is not pressed, the acceleration will stop and the ship will move at the current speed. If the user wants to stop the ship, they will need to repeat the same process by pressing the decelerating button.

The OrbitCam camera object follows the spaceship controlled by the user with its attached Follow Cam script. It has two mandatory public properties called Target, and Offset. The target is the spaceship, and the offset is where the camera is located in the first frame in the game, preferably with an offset relative to the spaceship so that it is visible from behind for example.

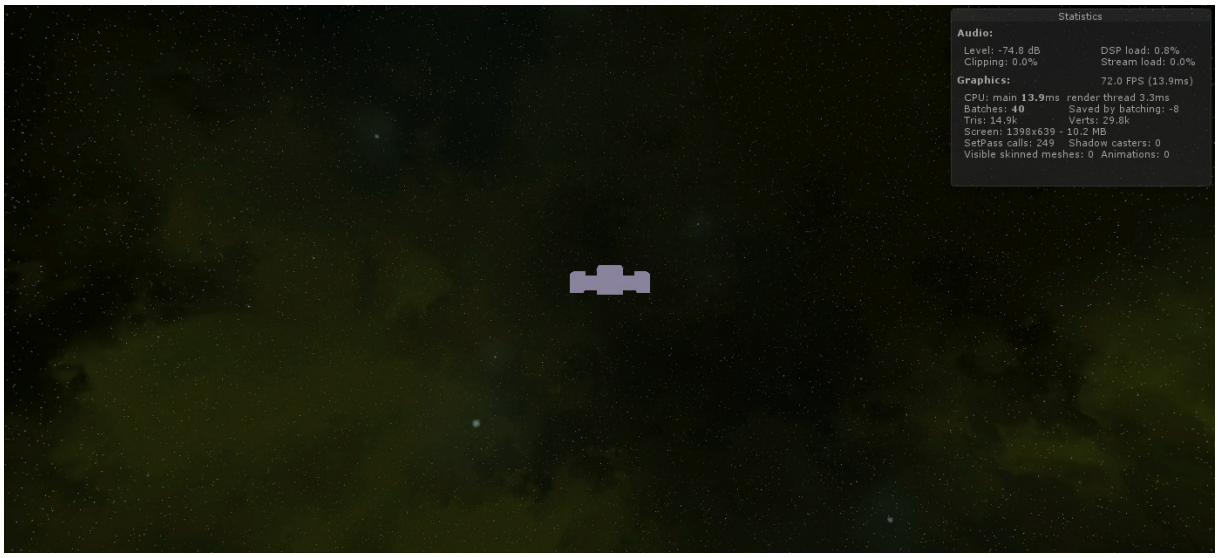

Figure 4.9: Player camera view

The Follow Cam script also allows the user to pan the camera around the spaceship. This rotation is interpolated so that the orbital rotation of the camera feels smooth. It is also

possible to zoom in and out with the mouse scroll wheel withing set limits.

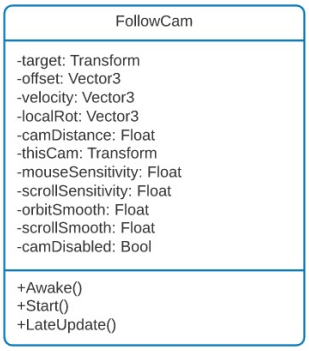

Figure 4.10: FollowCam class

4.2.4 Optimization

Each object rendered in the game takes up resources, and the more objects generated in our scene, the more it will affect the performance. This issue is addressed by the ActivatorLarge object which functions on the Activator script. The Activator script uses a serializable property called Distance and checks for each element in a list of ActivatorItems if it is farther away from the camera than that Distance. Each Star object's StarClose component will be disabled if the Star is too far away, and the StarFar component will be enabled, and vice versa. Star objects are added to this ActivatorItems list by a DisableIfFar script when they are created. This process significantly improves performance as only objects that are close enough will be displayed, instead of everything.

```
if (activatorItems.Count > 0)
{
    foreach (ActivatorItem item in activatorItems)
    {
        if (Vector3.Distance(layerCam.transform.position, item.objPos) > distance)
        {
            if (item.obj == null)
            {
                removeList.Add(item);
            }
            else
            {
                item.obj.transform.Find("StarClose").gameObject.SetActive(false);
                item.obj.transform.Find("StarFar").gameObject.SetActive(true);
            }
        }
        else
        {
            if (item.obj == null)
            {
                removeList.Add(item);
            }
            else
            {
                item.obj.transform.Find("StarClose").gameObject.SetActive(true);
                item.obj.transform.Find("StarFar").gameObject.SetActive(false);
            }
        }
    }
}
```

Figure 4.11: Activator code

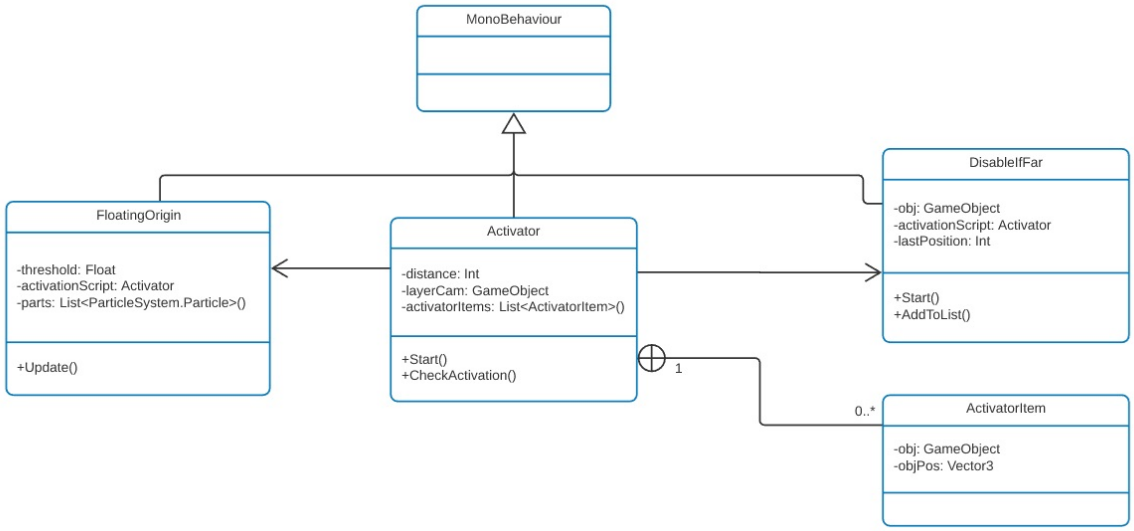

Figure 4.12: Activator and FloatingOrigin class diagram

4.2.5 Generating Star Systems

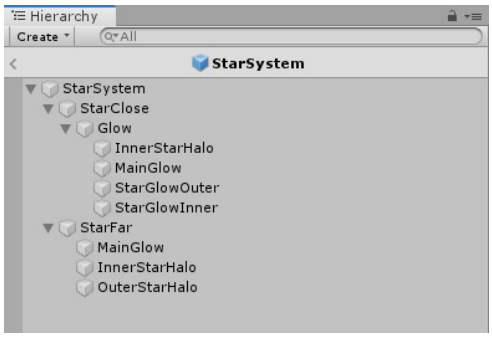

Figure 4.13: StarSystem object hierarchy view

The StarSystem prefab has 2 main components: StarClose and StarFar. One of these components will be active and inactive depending on the object's position relative to the position of the player. StarFar is composed of a few particle system components that simulate a natural visual aspect for a star that is far away, which is much less resource consuming that StarClose. StarClose also has a Glow component, also made up of particle system components to replicate a realistic visual appearance for an up close star. This is where planets and the star is created.

The star is a simple sphere created on top of the particles. It's size and color vary from star to star. Its corona aura effect is simulated by a system of particles. For the solid surface it uses some of the same functions that are used to generate a planet, but without elevation.

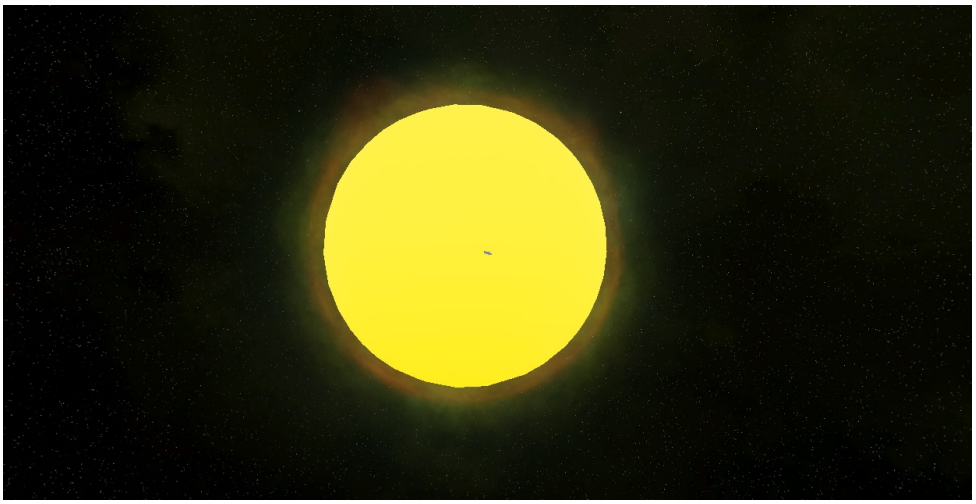

Figure 4.14: A star in Beyond Light

The planet consists of a spherical procedural mesh. In addition to the size varying from planet to planet, the terrain is entirely uniquely generated for each planet, with varied elevation mapped by a Noise function.

The mesh is generated with a TerrainFace script which computes and draws a mesh from triangles. First, it creates a cube composed out of 6 faces (meshes) from a certain number of triangles (resolution) according to 3 the directions (axes). This is done by calculating each point (vertex) and triangle individually and assigning them all to a new Mesh type object. To obtain a sphere from this cube, the vertices need to be normalized, meaning that every point must be equally far away from the origin of the cube.

```
{
    Vector3[] normals = new Vector3[resolution * resolution];
    Vector3[] vertices = new Vector3[resolution * resolution];
    int[] triangles = new int[(resolution - 1) * (resolution - 1) * 6];
    int triIndex = 0;

    for (int y = 0; y < resolution; y++)
    {
        for (int x = 0; x < resolution; x++)
        {
            int i = x + y * resolution;
            Vector2 percent = new Vector2(x, y) / (resolution - 1);
            Vector3 pointOnUnitCube = localUp + (percent.x - .5f) * 2 * axisA + (percent.y - .5f) * 2 * axisB;
            Vector3 pointOnUnitSphere = pointOnUnitCube.normalized;
            vertices[i] = shapeGenerator.CalculatePointOnPlanet(pointOnUnitSphere, radius, planet);
            normals[i] = vertices[i].normalized;

            if (x != resolution - 1 && y != resolution - 1)
            {
                triangles[triIndex] = i;
                triangles[triIndex + 1] = i + resolution + 1;
                triangles[triIndex + 2] = i + resolution;

                triangles[triIndex + 3] = i;
                triangles[triIndex + 4] = i + 1;
                triangles[triIndex + 5] = i + resolution + 1;

                triIndex += 6;
            }
        }
    }
    mesh.Clear();
    mesh.vertices = vertices;
    mesh.triangles = triangles;
    mesh.RecalculateNormals();
    mesh.normals = normals;
}
```

Figure 4.15: Mesh generation code

The CalculatePointOnPlanet() function from the ShapeGenerator class is called to compute the vertex. In order to explain how, it's necessary to shed light on the Noise, NoiseSettings, and NoiseFilter classes.

A Noise script was imported into this project in order to be able to successfully bring about the process of procedural terrain generation.[58] The Noise script is a module that outputs 3-dimensional Simplex Perlin noise. This noise module outputs values that range from -1.0 to +1.0. The NoiseSettings class is a class with just a constructor for all its properties be initialized. These properties are generated by making different calculations using the Noise script in the NoiseFilter class. The NoiseFilter class has an Evaluate() function that performs these operations, and returns a noise value that predicts the position

of the vertex on the planet mesh. This function is then used in the ShapeGenerator class to pass on the value based on every layer of noise that results in the creating the point the procedural sphere.

These values are randomized within certain bounds for every planet in the Planet script when it is created, in order to generate a unique terrain for it. These properties can also be freely adjusted to showcase the procedural aspect of the application.

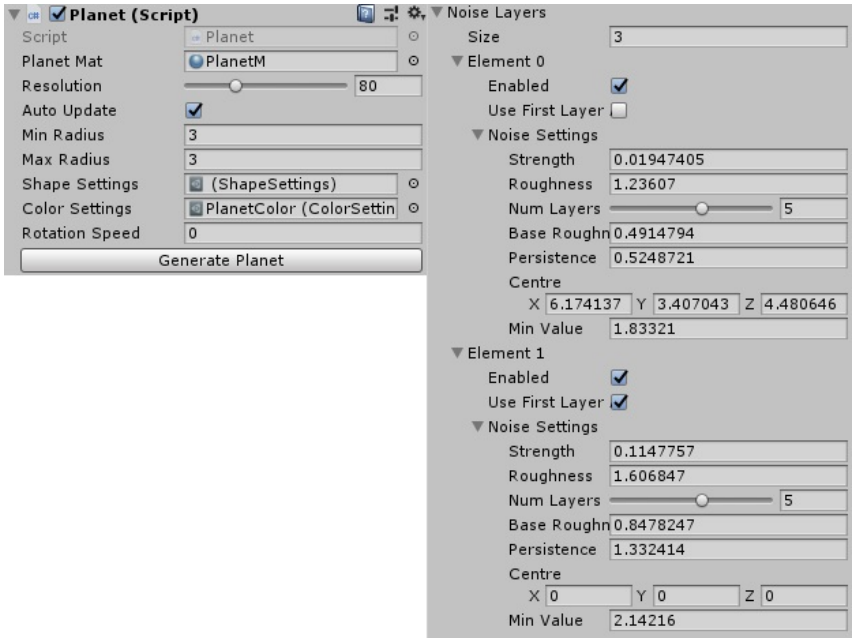

Figure 4.16: Planet script inspector view

Terrain adjustment values:

- Strength - the raw strength/intensity of the elevation or the height of the "hills" on the planet terrain;
- Roughness - the frequency of elevation or the number of "hills" on the planet terrain;
- Persistence - similar to strength, more subtle and natural;
- Centre (Vector3) - moves the elevation along the three axes;
- MinValue - Makes it possible to recede the terrain into the base sphere shape of the planet;

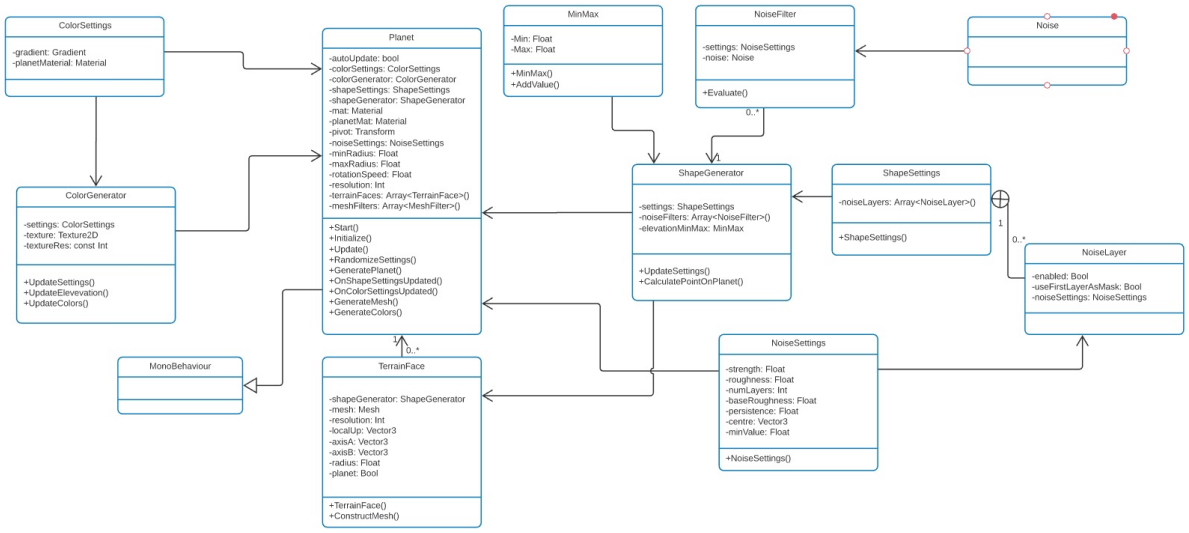

Figure 4.17: Planet class diagram (similar to Star class diagram)

In order to obtain more natural and more detailed terrain, multiple layers of noise can be added on top of each other. The first layer can act as a mask (base) for the rest. This means that all the other layers compute elevation only in the places where elevation is more prominent in the first layer.

A planet script also has a ColorSettings property. The ColorSettings contains a gradient and a material, which are applied as a texture to the planet material through a custom shader. The texture is created by the Planet Shader, and is based on the minimum and maximum elevation values of the mesh.

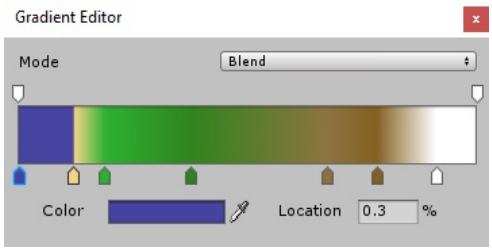

Figure 4.18: Planet gradient)

This way, a gradient with colors can be applied to the planet's material on different elevations. For example, when the elevation is lowest, a blue color can be set on the gradient property. When the elevation is slightly higher, a sandy color like yellow can be used. Higher, green for fields and vegetation, brown for mountains, and white for the highest elevation, simulating snowy areas.

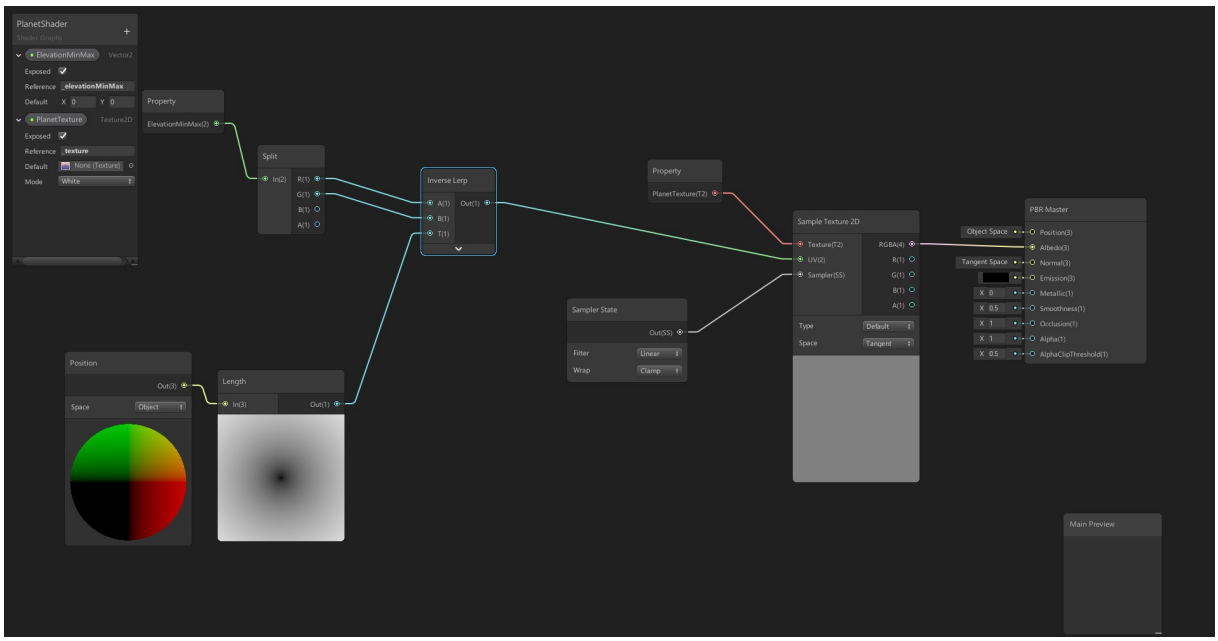

Figure 4.19: Planet shader functionality

Together with the mesh generation based on layered noise functionality, with proper constraints applied to the random terrain adjustment values, this entire process allows the universe manager to instantiate unique, realistic, procedural, earth-like planets that orbit around stars. These values can be modified in many ways in order to obtain many different planets of different colors. Some examples can be seen below.

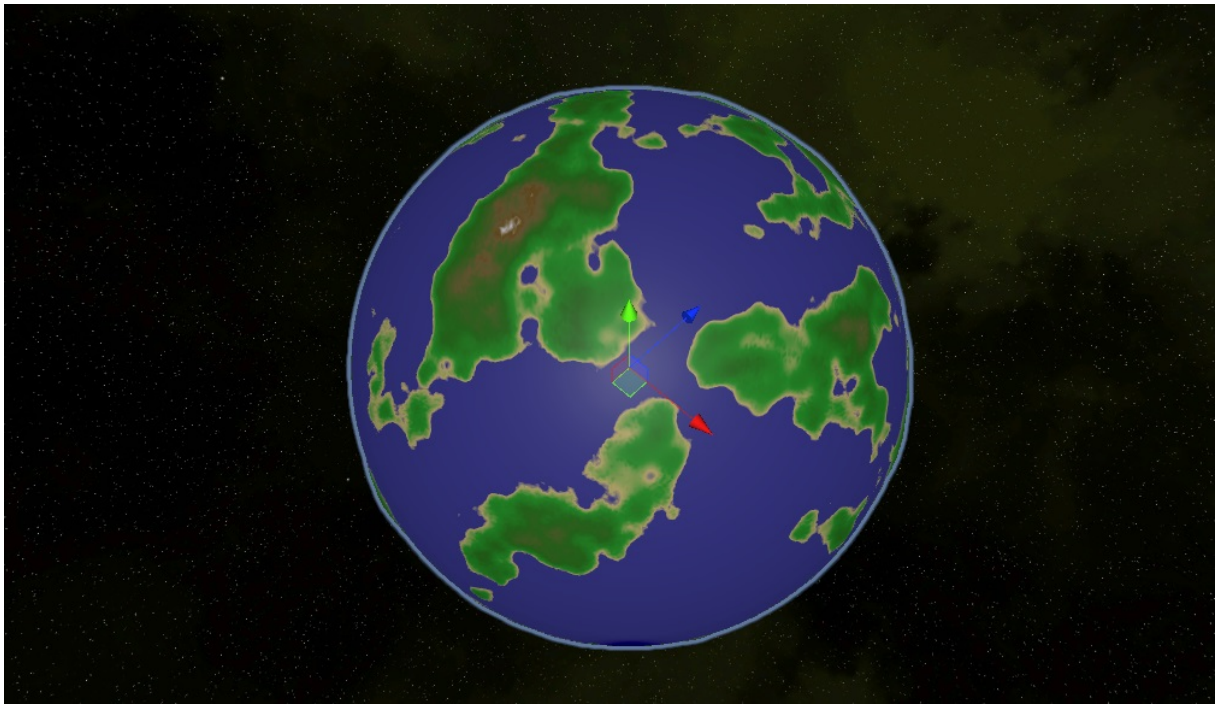

Figure 4.20: Planet generated in Beyond Light

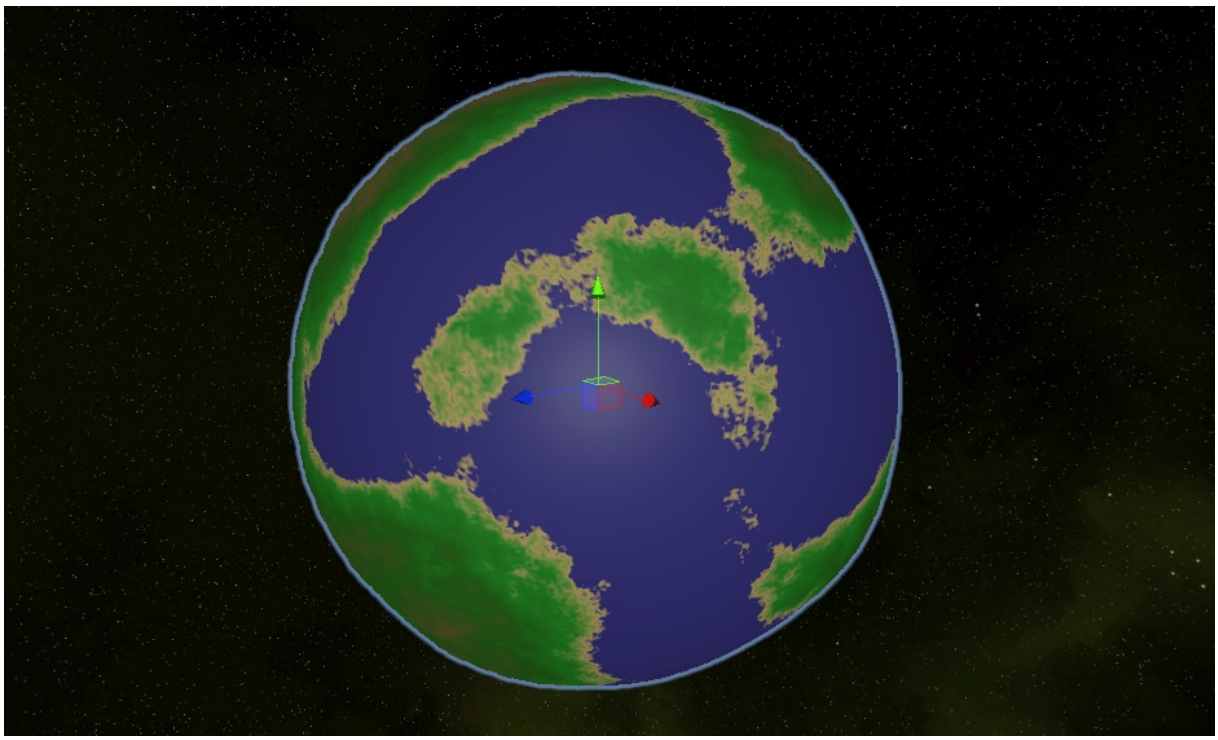

Figure 4.21: Planet generated in Beyond Light

Planets orbit around their star at different speeds depending on their distance from the star (further = slower orbit), and also rotate around a "vertical" axis.

A simple but crucial element that plays a big part in space immersion is the skybox. A procedurally generated image converted to a cubemap texture of a night sky was imported and used for the skybox.

5. Conclusions

5.1 Conclusions

The relevant concepts and frameworks discussed that are practiced for the creation of a large-scale realistic virtual universe with procedural elements were researched, explored, and documented in this project.

Automatically generating objects in a manager with various constraints provided the user to a unique configuration of stars and planets for the game world.

The application reproduced realistic depictions of outer space environment accurately enough for it to be an immersive visual experience for the user.

Using multiple cameras with different LODs and scale factors addressed the scale aspects of this simultation. Floating origin concept solved the navigation issue risen in the context of a large-scale environment.

Mesh generation and modification through usage of Noise functions, and custom shader texture creation produced unique configurations of natural looking terrain on planets.

The game application was developed to apply these concepts in order to produce a visual simulation of a procedurally generated space environment.

5.2 Further work and improvements

Since this application was designed as a proof of the concepts discussed in the paper rather than a finite product, there are many ways in which the features implemented can be extended or even collectively function as a basis to build a fully fledged video game on, with additional features and mechanics.

There is undoubtedly plenty of space where the modules explored and applied in Beyond Light can be improved in order to achieve even greater realistic qualities and detail. Optimization, performance enhancing updates as well as aesthetic aspects and graphics improvements are in place.

Beyond Light is intended to work, in the future, as a complete video game with goals, obstacles, enemies, resources etc. while preserving the open world exploration element as well. Other features that use and simulate AI, combat, strategy will be further researched and applied through hard work and ambition.

6. Bibliography

References added to the end of a paragraph imply that the idea as a whole of that paragraph is similar to what is found in the reference link.

[1] Joel Harvey, *Size Matters: Are Bigger Game Worlds Less Fun To Play?*, 2017, https://www.theversed.com/71053/original-size-matters-bigger-game-worlds-less-fun-play/#.e96UlEtcAY

[2] Jamie Sefton, *Size Matters: The roots of open-world games*, 2008, https://www.gamesradar.com/the-roots-of-open-world-games/

[3] https://gamicus.gamepedia.com/Open-world_video_games

[4] Ernest Adams, *Fundamentals of Game Design: Game Worlds*, 2009 http://www.peachpit.com/articles/article.aspx?p=1398008&seqNum=3

[5] Tanner Fox, *The 30 Biggest Open-Worlds In Video Games Ever (From Smallest To Largest)*, 2018 https://www.thegamer.com/video-game-worlds-biggest-smallest/

[6] Erik Bethke, *Game development and production*, Texas: Wordware Publishing, Inc. ISBN 1-55622-951-8, 2003

[7] Bob Bates, *Game Design (2nd ed.)*, Thomson Course Technology. ISBN 1-59200-493-8, 2004

[8] https://en.wikipedia.org/wiki/Game_programming

[9] Interesting Engineering, *How Do Game Engines Work?*, https://interestingengineering.com/how-game-engines-work, 2016

[10] Ken Jones, *Simulations: A Handbook for Teachers and Trainers* p. 21, 1995

[11]*Single-precision floating-point format*, https://en.wikipedia.org/wiki/Single-precision_floating-point_format

[12] *Touring Machine Company*, https://www.touringmachine.com/Articles/aircraft/6/

[13] https://i.pinimg.com/originals/da/8c/80/da8c806749a62df408942236c0686871.jpg

[14] https://unity3d.com/unity

[15] https://www.unrealengine.com/en-US/

[16] https://www.yoyogames.com/gamemaker

[17] https://godotengine.org/

[18] https://developers.google.com/maps/documentation/gaming/move_floating_origin

[19] Dave Newson, *Unity: coordinates and scales*, http://davenewson.com/posts/2013/unity-coordinates-and-scales.html, 2013

[20] *Unity - Manual: Using more than one camera*, https://docs.unity3d.com/Manual/MultipleCameras.html

[21] Tony Lovell, *Floating Origin*, http://wiki.unity3d.com/index.php?title=Floating_Origin, 2018

[22] Chris Thorne, *Using a Floating Origin to Improve Fidelity and Performance of Large, Distributed Virtual Worlds*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.7201&rep=rep1&type=pdf

[23] patriciaispale, *Procedural Generation*,
https://www.instructables.com/id/Procedural-Generation/

[24] https://en.wikipedia.org/wiki/Procedural_generation

[25] http://pcg.wikidot.com/

[26] Dale Green, *Procedural Content Generation for C++ Game Development*, 2016, https://www.academia.edu/38251876/Procedural_Content_Generation_for_C_Game_Development

[27]https://avalanchestudios.com/games/

[28]http://www.hellogames.org/

[29] *Random Seed and Pseudo Random Number Generators* https://en.wikipedia.org/wiki/Random_seed, https://en.wikipedia.org/wiki/Pseudorandom_number_generator

[59] Radu Trimbitas, *Random Number Generation* http://math.ubbcluj.ro/~tradu/randnumg2.pdf

[30] Stephanie Glen, Random Seed: Definition, 2017 https://www.statisticshowto.datasciencecentral.com/random-seed-definition/

[31] https://en.wikipedia.org/wiki/Polygon_mesh#cite_ref-Smith(2006)_1-0

[32] https://en.wikipedia.org/wiki/Mesh_generation

[60] Marshall Bern, Paul Plassmann, *Mesh Generation* , https://www.ics.uci.edu/~eppstein/280g/Bern-Plassman-meshgen.pdf

[33] Oscar Sebio Cajaraville, *Four Ways to Create a Mesh for a Sphere*, 2015, https://medium.com/game-dev-daily/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4

[34] Jari Komppa, *Sphere Mesh Creation*, http://sol.gfxile.net/sphere/

[35] Ken Perlin, *MAKING NOISE*, 1999, https://web.archive.org/web/20071013155720/http://noisemachine.com/talk1/index.html

[36]https://en.wikipedia.org/wiki/Gradient_noise

[37] Ken Perlin, *Noise and Turbulence*, 1999, https://mrl.nyu.edu/~perlin/doc/oscar.html

[38] https://en.wikipedia.org/wiki/Simplex_noise

[39] Red Blob Games, *Noise Functions and Map Generation*, 2013, https://www.redblobgames.com/articles/noise/introduction.html, *Making maps with noise functions*, https://www.redblobgames.com/maps/terrain-from-noise/

[40] Josh Petty, *What is Unity 3D & What is it Used For?*, https://conceptartempire.com/what-is-unity/

[41] *Unity Manual: The Inspector window*,
https://docs.unity3d.com/Manual/UsingTheInspector.html

[42] Georgi Ivanov, *Introduction to Unity Scripting*, 2016 https://www.raywenderlich.com/980-introduction-to-unity-scripting

[43] *Unity Manual: GameObject*, https://docs.unity3d.com/Manual/class-GameObject.html

[44] *Unity Manual: Particle Systems*, https://docs.unity3d.com/Manual/ParticleSystems.html

[45] *Unity Scripting API: MonoBehaviour*, https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[46] Anthony Uccello, Eric Van de Kerckhove, *Introduction To Unity: Particle Systems*, 2018 https://www.raywenderlich.com/138-introduction-to-unity-particle-systems

[47] Porter, *Unity: Now You're Thinking With Components*, 2013 https://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

[48] *Graphics rendering*, https://unity3d.com/unity/features/graphics-rendering

[49] Kieran Colenutt, *The High Definition Render Pipeline: Getting Started Guide for Artists*, 2018 https://blogs.unity3d.com/2018/09/24/the-high-definition-render-pipeline-getting-started-guide-for-artists/

[50] *Scriptable Render Pipeline*
https://github.com/Unity-Technologies/ScriptableRenderPipeline/wiki/High-Definition-Render-Pipeline-overview

[51] *Mandelbrot Set Zoom*, https://www.youtube.com/watch?v=8cgp2WNNKmQ&t=291s

[52] *Spore creature creator*, http://www.spore.com/creatureCreator

[53] Christopher Groux, *'No Man's Sky' New Experimental PC Update Released: Improves Framerate, Pop-In And More*, 2016, https://www.ibtimes.com/no-mans-sky-new-

experimental-pc-update-released-improves-framerate-pop-more-2416663

[54] *Dolphin mesh*, https://en.wikipedia.org/wiki/Triangle_mesh

[55] *Bunny mesh*, https://sites.google.com/site/dengwirda/jigsaw

[56] *Simplex vs Perlin noise*, https://radiumsoftware.tumblr.com/post/134168524229/amp

[57] *Order of Execution for Event Functions*, https://docs.unity3d.com/Manual/ExecutionOrder.html

[58] Stefan Gustavson, Karsten Schmidt, *Simplex Noise Script*, https://github.com/SebLague/Procedural-Planets/blob/master/Procedural%20Planet%20Noise/Noise.cs