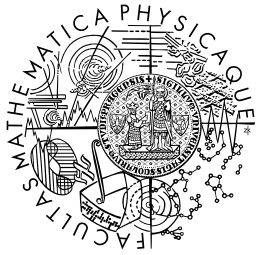


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Anna Vičíková

Procedural generation of game visuals in shaders and Unity DOTS

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Vojtěch Černý
Study programme: Computer Science

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Mgr. Vojtěch Černý for the guidance and support of this thesis. Thank you very much to my colleagues in the development team of the game "Game of Life" and most of all to the leader of this team and my dear friend Jakub Holík, who supported me in the technical and moral aspect of the project. Many thanks to my colleagues at work and friends, with emphasis on BcA. Tereza Šimanovská and Ing. Petr Nohejl, who have always been there for me and provided me with mental support. I also thank my family, who supported me throughout my studies.

Title: Procedural generation of game visuals in shaders and Unity DOTS

Author: Anna Vičíková

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Vojtěch Černý, Department of Software and Computer Science Education

Abstract: This thesis describes an approach to creating a GPU-based rendering system for an evolution-simulating and data-based game in Unity DOTS. The game features a vast array of diverse creatures that are unknown in advance and must be assigned a visualization on the fly. We aim to identify possible methods for generating these specific visualizations within a high-performance environment using Unity DOTS. We also implement an algorithm designed to create an aesthetically pleasing visualization of the content. The result of this work offers a possible solution to use a combination of GPU-based procedural generation and a data-driven approach to create dynamically generated game content that maintains high performance. This work enables the development of specific types of games with high visual demands. The output of this solution will broaden the horizons in the world of procedural generation of game visuals and thus help development teams that do not have a large number of game artists at their disposal in the graphical side of the game.

Keywords: Unity DOTS, procedural generation, shader languages

Contents

Introduction

Video games represent a complex combination of various components, encompassing levels, narratives, characters, visual elements, and soundscapes, each crafted by specialists such as level designers, writers, artists, and sound engineers. While such specialization yields high-quality assets, it entails significant time and financial investment. Procedural content generation (PCG) circumvents the need for human intervention in content creation, facilitating faster and cost-effective game development while maintaining quality. PCG streamlines production and fosters creativity by generating novel solutions to design challenges.[1]

Let's give an example of game content creation. The development of massively multiplayer online role-playing games like World of Warcraft exemplifies the evolution of game development methodologies. Creating expansive worlds and intricate ecosystems in World of Warcraft demanded sizable teams of up to 40 individuals, which later expanded to hundreds of developers to sustain quality and complexity.[1] [2] Conversely, titles such as No Man's Sky (Hello Games, 2016), initially developed by a team of 4 people and later by over 17 developers [3], showcase the potential of procedural generation, offering vast, virtually infinite game worlds.

The use of procedural generation concepts is more or less a common part of game development nowadays. A common practice is to generate the content on the CPU (central processing unit). However, this is not adapted to generate content where the computation requires high parallelizability (e.g. complex terrain). For these cases, it is advisable to generate content on the GPU (graphics processing unit), which is much more efficient and thus saves the CPU efficiency for the logical processes of the game.

In terms of efficiency and performance, the game architecture must also be considered. While traditional game development often relies on object-oriented design (OOD), data-oriented design (DOD) emerges as a paradigm tailored for high-performance and flexible architectures, particularly relevant in contemporary gaming. Unlike OOD's reliance on a single main thread, DOD distributes processing load uniformly across multiple threads, leveraging the efficiency of multi-core processors [4]. Although initially less explored in academia due to OOD's perceived cost and development time advantages [4], the increasing accessibility of tools such as Unity DOTS and Unreal Engine's MassEntity framework has catalyzed the adoption of DOD in game development.

This thesis explores the intersection of procedural generation on GPU and data-driven design, focusing on the challenges inherent in efficiently generating visually complex content. The research contributes to developing an upcoming game depicted in Figure 1, that employs evolutionary simulations within a data-oriented framework, highlighting the potential of these methodologies in shaping the future of game development.

[1] In the beginning, the game's content was created manually. Later, Blizzard Entertainment embraced procedural generation capabilities and a symbolic change was made in the World of Warcraft: Shadowlands expansion, where they introduced a new dungeon, Torghast, with procedurally generated enemy spawns.

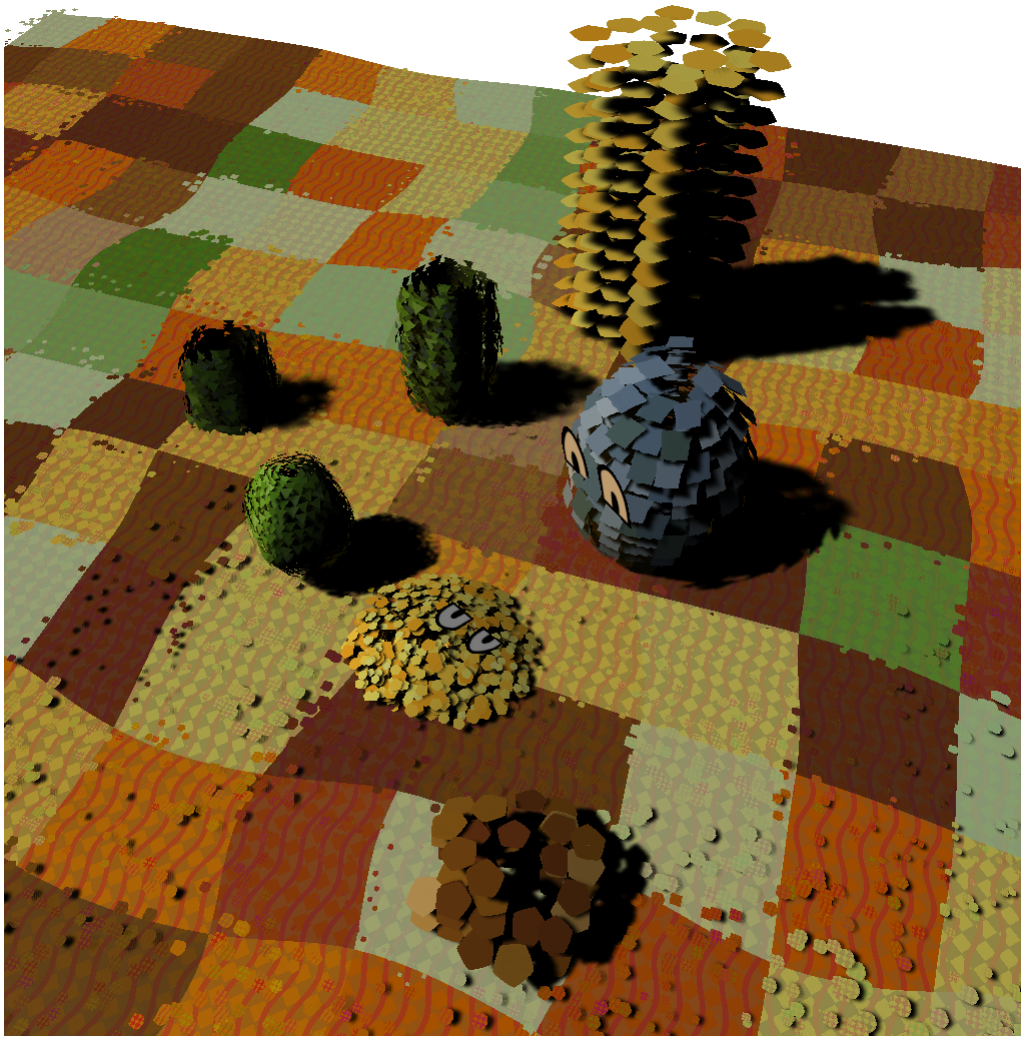


Figure 1 Visual game preview.

Goals

The primary goal of this thesis is to identify and develop a method that effectively meets the conditions set by input parameters for generating visual entities within the game environment. Given the non-standard nature of this approach, the research will involve an extensive exploration of existing procedures for procedural graphics generation, shader programming and data-oriented architecture. Through systematic experimentation and implementation within the Unity DOTS game, the goal is to find a suitable procedural generation technique that fits the defined game design and provide this solution for developers who follow a similar path in combining PCG and data-driven approaches.

Specifically, the research seeks to achieve the following objectives:

Method Identification Analyze existing methods and algorithms for procedural graphics generation, including but not limited to terrain, textures, objects, and other visual elements relevant to the game environment.

Define inputs based on game design From the written game design, identify possible inputs that will influence the generation process and, thus, the final visual.

Implementation Integrate selected procedural generation techniques into the game engine and adapt them to the specific game design requirements and constraints of Unity DOTS. Subsequently, the generated visual effects will be evaluated according to predefined criteria and performance.

Key findings After the evaluation, summarize the important lessons learned during development and make recommendations for other developers who combine GPU-based generation and the use of a high-performance environment through a data-driven approach.

Structure

In chapter 1. Analysis we explore the methods and practices of procedural visual generation in games and the Unity game engine environment with its DOTS framework. We introduce the basic algorithms, the Entity Component System architecture and graphics pipeline.

In chapter 2. Game Design, we will introduce a brief game design and define the important elements for generating the appearance of the game entities.

In order to see which generation method may be appropriate, in chapter 3. Experiments we will focus on experimenting with the observed knowledge from the analysis.

After choosing a concrete method to generate the appropriate game graphics, in chapter 4. Implementation we will dive into the implementation in Unity DOTS and write the appropriate code within the ECS architecture and shaders to render the visuals. In chapter 5. Visual Results, we will look at and analyze the results of the chosen solution.

Related work

Many games are using procedurally generated content. The common usage of PCG is for generating levels, dungeons, and terrain. By far two of the most well-known games contain procedurally generated creatures: Spore (2008) and No Man's Sky (2016). Regarding data-driven games, it's worth mentioning Overwatch (2016) from Blizzard Entertainment and Diplomacy is not an option (2022) by Door 407.

Games using procedural content generation

Spore

Spore was developed by Maxis and published by Electronic Arts on September 4, 2008. The leading creator was game designer Will Wright, who also worked on SimCity and The Sims, where he showed the potential of the simulation theme in games. Spore is an evolutionary strategy game with adventure, RPG and simulation elements. The player experiences the evolutionary development of organisms in five stages, from a small cell in the sea to building an advanced civilization flying into space. The fundamental aspect of the game is the creation of creatures, which is in the player's hands. The motivation behind this is the idea that the player develops a greater empathy for the game if they can create something in it. This game gives the player a "toy" to play with and thus use their imagination to experiment with the world.[5]

The metaball is the cornerstone of mesh generation in Spore. These are blobby implicit surfaces that are topologically robust while lacking the "local control" to allow for polygon-level modelling. This method requires as much robustness as possible, as the game gives the player all the control over the creature's morphology while the skin must be attached to the creature's skeleton.[6]

Figure 2 shows the Spore creature creator, a tool in Spore that allows the

Figure 2 Spore creature creator [7]

player to modify the shape of their creature and add rigblocks to it. Rigblocks are pre-made geometric building blocks representing specific parts of the creature, such as mouths or arms, that the player can attach to the creature's body.[8] Through combinations of modifying the body structure and attaching rigblocks, the player can create a unique creature to his own liking. These creatures can then be sent to the Sporepedia[2] creature database, which contains all published player creations from the world of Spore and its expansions.

No Man's Sky

No Man's Sky[3] is a sci-fi exploration game where players start on planets scattered around the galaxy. They can travel to other planets and discover different types of environments. The game has a solid technical design to generate and simulate the environments on the planets, including the creatures living on them. Hello Games, the studio that created the game, used such algorithms to generate worlds on a scale of two to the power of 64, so there are 18.4 quintillion planets in the game that are generated on the fly, while the game itself requires only 5 GB of disk space. The cornerstone of the contour generation in No Man's Sky is voxels[4]. A voxel is a single point in 3D space with size, colour and opacity comparable to a large pixel. The entire terrain of single planet is divided into regions ($36 \times 36 \times 36m$), and these regions consist of 1-meter voxels, where one voxel occupies 6 bytes of data (density, 2 different materials and material blend). [10] The pipeline consists of noise generation (e.g. Perlin worms) on GPU, polygonisation[5] on CPU and render, physics and population on both GPU and CPU.[11]

Creatures are created by randomly selecting individual parts from a library, with a system running in the background that automatically balances the creature's weight and adjusts its skeleton to make it look more realistic. For example, it can't happen that a creature with a small body has a huge head. [12]

Games using data-oriented approach

Overwatch

Overwatch[6], a team-based first-person shooter released by Blizzard Entertainment in 2016, utilizes an Entity-Component-System (ECS) architecture, a common approach in data-oriented design. This architectural choice enables the

[2] `https://www.spore.com/sporepedia`

[3] `https://store.steampowered.com/app/275850/No_Mans_Sky/`

[4] Another well-known example of a game that procedurally generates its terrain using voxels is 2009's Minecraft, developed by Markus Persson, also known as Notch. Minecraft uses voxels to store terrain data and uses polygon rendering to represent each voxel as a cubic block. [9]

[5] Polygonisation refers to the process of converting a continuous implicit surface into discrete polygonal mesh. The best-known techniques are Marching Cubes and Dual Contouring algorithms. Innes McKendrick, lead programmer at Hello Games, at his GDC 2017 lecture, said that they selected Dual Contouring over the Marching Cubes algorithm because Marching Cubes perform poorly with points that are not on the edge of a voxel. This makes it not very suitable for models with sharp angles. But in No Man's Sky, they polygonise multiple times with different polygonisatiors to get different sets of data, including polygonisating using the Marching Cubes algorithm as it can be seen in Figure 3. [10]

[6] `https://overwatch.blizzard.com/en-us/`

Figure 3 No Man's Sky generated environment generated using voxels and polygonised using Dual Contouring and Marching Cubes algorithms. [10]

development team to achieve rich gameplay variation, optimization, improved code readability, and control over various independent system behaviours, such as AFK (Away from keyboard) checks, player broadcast network messages and player name resolution. The transition to ECS began with a prototype during development, with the legacy code being gradually refactored over three years. ECS facilitates the integration of disparate systems with minimal coupling, simplifying the task for engineers working on the glue code that binds all systems together. [13] Due to the fact that ECS is modular, it is easy to add or even remove components and systems without redoing a large amount of code. Timothy Ford, Lead Gameplay Engineer at Blizzard Entertainment, in his GDC 2017 talk highlights this approach as being code-friendly because it allows developers to manage the complexity of a rapidly growing codebase. [13]

Diplomacy is not an option

A compelling example of the data-oriented approach in game development is the real-time strategy and tower defense game Diplomacy is Not an Option[7] by indie studio Door 407. In 2016, a developer known as "eizenhorn" initiated a discussion on the Unity DOTS forum, focusing on a technology package for the Unity game engine that supports data-oriented game development. In this thread, eizenhorn detailed DOTS implementation in the game's production. The game excels in rendering vast numbers of animated assault units—ranging from hundreds to hundreds of thousands—while simultaneously managing tens of thousands of visualized resources and population entities. Despite this complexity, the game remains highly optimized, maintaining high frames per second (FPS). This efficiency is achieved using the ECS architecture combined with custom-written shaders and a proprietary animation system [8]. Additionally, the game features procedural map generation, enhancing its dynamic and replayable nature. Figure 4 is a screenshot taken in-game, where thousands of units are displayed in real time and the player blows them up via an ability blast. [14]

[7] `https://store.steampowered.com/app/1272320/Diplomacy_is_Not_an_Option/`

[8] As of now, Unity DOTS does not provide any animation system in its package and needs to work around this with its own systems, third-party tools, or using a hybrid approach of GameObjects and ECS

Figure 4 Screenshot from game Diplomacy is Not an option. [15]

1 Analysis

In this chapter, we discuss the problem of procedural generation of game visuals and list several approaches to generating terrain and characters. Subsequently, the analysis will look at the basic aspects of the data-driven approach in game development and a specific implementation in Unity DOTS. Finally, we define the game and its game elements, which is the basis for implementing this work's solution.

1.1 Motivation

The landscape of video game development is continuously evolving, with technological advancements pushing the boundaries of what is possible in terms of immersive gameplay experiences. This is why game development is fascinating and allows game developers to try to combine established practices with original and experimental techniques. Numerous fundamental algorithms exist for generating terrains and natural features, and these can be adapted, enhanced, combined, and tested with various input parameters to produce unique results suitable for integration into video games. When focusing on generation, particularly real-time generation, it is crucial to prioritize algorithm performance and code optimization to ensure the game remains playable at high frames per second (FPS) and preserves the player's gaming experience. The Entity Component System (ECS) architecture, a cornerstone of the data-oriented approach in game development, supports these goals. Despite its potential, ECS architecture is not yet widely adopted in the video game industry, making its exploration, particularly in conjunction with procedural visual generation, a fascinating study area.

1.2 Procedural generation of visuals in games

Procedural generation (PCG) has many approaches to achieve content generation in games. We can consider PCG as a viable solution to content generation challenges. PCG can be used to rapidly create low-detail elements like grass to generate complex game mechanics after extensive computational processing. Each application presents unique requirements and constraints, highlighting the importance of speed, reliability, controllability, expressivity/diversity, creativity, and believability. These properties are not isolated but often entail trade-offs; for instance, achieving higher speed may compromise the quality of generated content, while maintaining expressivity and diversity without sacrificing quality poses a significant challenge. It is necessary to control over generated content, enabling users or algorithms to influence specific aspects of the generated output. [1]

Several well-known algorithms are considered basic algorithms for the purpose of procedural content generation in games. While the most well-known use is in terrains and landscapes, the following algorithms can also generate self-similar structures. [16] We will describe these basic algorithms and how they could be used for this thesis.

1.2.1 Noises

Noise functions are fundamental techniques to create realistic terrains and landscapes in procedural content generation. When generating natural landscapes, the resulting environments must exhibit randomness while adhering to specific constraints and exhibiting particular characteristics. Noise is a mathematical sequence of pseudo-random numbers used to perturb a flat surface, generating intricate and navigable terrains. A prominent application of noise in terrain generation is the use of heightmaps. Heightmaps are scalar fields that assign a height value to each point on a grid, effectively defining the elevation of the terrain relative to a baseline. To generate random terrain, one might initially consider using a random-number generator to populate a heightmap, however, this approach results in unrealistic, spiky terrain because it generates independent random values, lacking the spatial correlation observed in natural landscapes. [1] The following noise types are used to increase the realism and correct the correlation:

Value noise

Value noise deals with interpolating random values on a coarser grid at each point of the height map. The basis is **linear interpolation** (often called "lerp"). Linear interpolation is a mathematical function that returns a value between two given values and a parameter t in the closed unit interval from 0 to 1. Its definition can be seen in code snippet 1.1.

```
float Lerp(float startingValue, float endValue, float t)
{
    return startingValue + (endValue - startingValue) * t;
}
```

Listing 1.1 Definition of linear interpolation in C#

In two-dimensional space, **bilinear interpolation** is used to interpolate a weighted average in the horizontal and vertical direction. The linear interpolation is then used to create sharp peaks and valleys. To soften these sharp features, it is advisable not to use a linear curve but an S-curve shape. A mathematical slope function *s(x)* is commonly used,

$$s(x) = -2x^3 + 3x^2$$

and interpolated in both directions on a 2D grid. This method is called **bicubic interpolation**. [1]

Gradient noise

An alternative method for generating peaks is to use random numbers to create random gradients, which determine the steepness and direction of the slopes. To calculate height values from these gradients, start by setting the height at each lattice point to zero. Then, for each non-lattice point, calculate its height by taking the dot product of the gradient vector at a nearby lattice point and the vector from the lattice point to the non-lattice point. Repeat this process for all surrounding lattice points and combine the resulting values using interpolation. One type of gradient noise is called **Perlin noise**, which can be seen in the figure 1.1. [1]

Figure 1.1 Perlin noise. Generated using Perlin Noise function from Unity Mathf library.

1.2.2 Fractals

A fractal is a geometric shape that relies on self-similarity in nature, where we look at a natural phenomenon on a larger scale and then see the same kind of variation on a smaller scale. Fractals are often used in terrain generation, but they can also be used in abstract game visuals and game design[1]. The classic method is to iteratively generate noise layers at multiple scales, where we scale by the inverse of their frequency and where these layers are then summed. If we have a function to generate noise, we name it *noise(f)*, where f is a frequency, then we can use the relation 1.1 to generate a fractal to a specified level. [1]

$$noise(f) + \frac{1}{2}noise(2f) + \frac{1}{4}noise(4f) + ... \tag{1.1}$$

There are many methods for generating fractals. One well-known one is the **Mandelbrot set** and the related **Julia set**. The Mandelbrot set denotes the set of points in the complex plane where its corresponding Julia set is continuous. The Julia set is defined using the initial complex number

$$z = x + yi, \tag{1.2}$$

where $i^2 = -1$ and x, y are the coordinates of a pixel in the interval from -2 to 2 [18]. The function for the iterative calculation is given by a complex quadratic polynomial (a quadratic polynomial where the coefficients and variables are complex numbers):

$$f_c(z) = z^2 + c, \tag{1.3}$$

where c is the complex number given by the specific Julia set. The pixel is in the Julia set if the maximum size is 2 [18]. The Julia set is a subset of the complex space defined as

$$K(f_c) = \{z \in \mathbb{C} : \forall n \in \mathbb{N}, |f_c^n(z)| \leq \mathbb{R}\}, \tag{1.4}$$

[1]Jonathan Blow, an American game designer and programmer, gave a talk at GDC Europe in 2011 about highlighting fractals and looking at game design with them.[17] At the time of this talk, he was developing The Witness (`https://store.steampowered.com/app/210970/The_Witness/`), a puzzle video game released in 2016 about exploring an open-world island filled with natural and man-made structures to solve puzzles mostly using provided tables. The concept of fractals in game design can be seen in the game as individual puzzles, where some build on each other when zoomed out, can be grouped together with a self-similarity element.

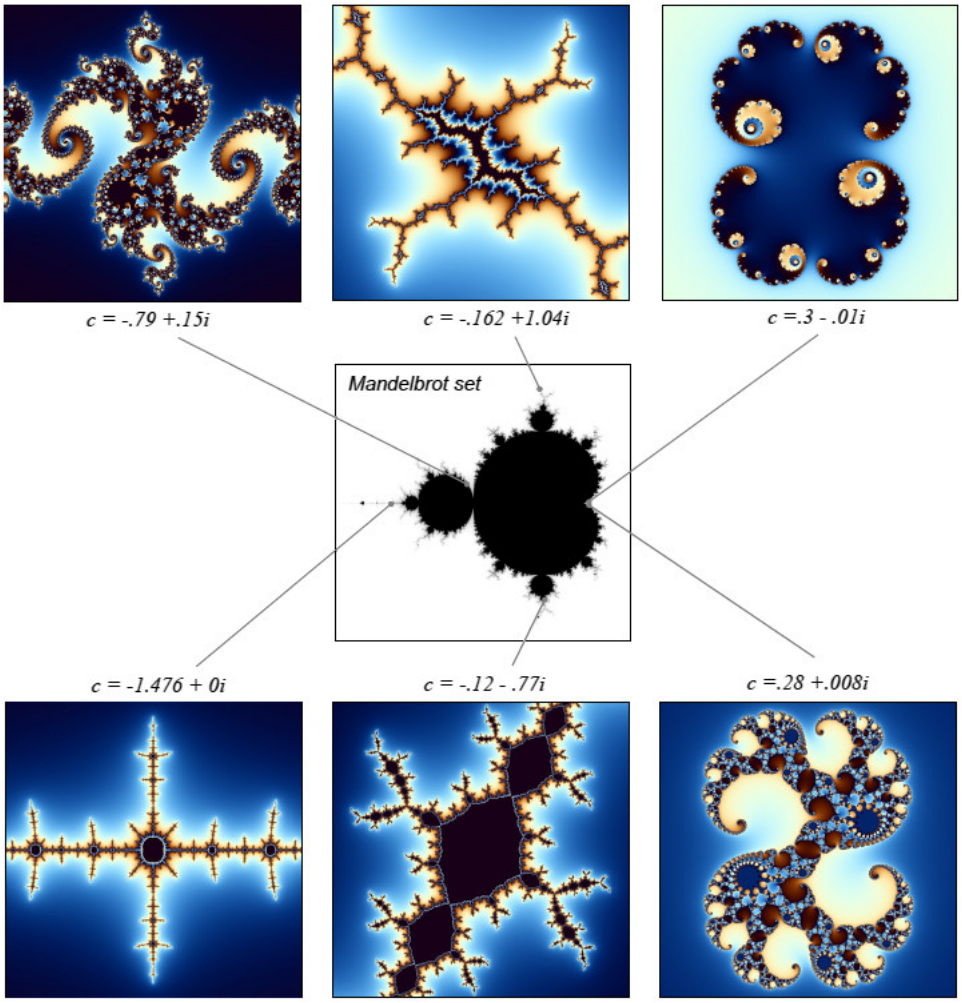


Figure 1.2 Visualisation of Mandelbrot set and its six Julia sets. [18]

where $f_c^n(z)$ is nth iterate of $f_c(z)$, then Julia set of this function is the boundary of $K(f_c)$. [19] The Mandelbrot set contains Julia sets at their specific location, as given by their parameter value c. Typically, we can see interesting Julio sets near the edges of the Mandelbrot set [18], as shown in Figure 1.2.

Another way to create fractals is by using L-systems described in the next section.

1.2.3 L-systems

L-systems, or Lindenmayer systems, are a formalism used to model the growth and topology of plants by describing the relationships between cells or larger plant modules. L-systems utilize a rewriting process, which involves iteratively replacing parts of an initial object with a set of predefined rules or production terms. This method of defining complex structures through simple, recursive rules is a powerful tool for plant modelling and has been widely applied in procedural graphics. For instance, the Koch snowflake curve is a well-known example of fractal pattern generation through this rewriting process. The fractal nature of L-systems makes them particularly effective for simulating the intricate and self-similar patterns often observed in botanical structures. [20]

The simplest class of L-systems is **DOL-systems** (deterministic context-free Lindenmayer system), which, by its name, is deterministic and context-free. The formal definition of this system is as follows.

Let us have an alphabet V, the set of all words V^* of alphabet V, and the set of all non-empty words V^+ of alphabet V. Then the *string* is an ordered triple

$$G =< V, \omega, P >, \tag{1.5}$$

where V is the mentioned *alphabet* of the system, $\omega \in V^+$ is a nonempty word called *axiom* and determines the initial state of the system, and $P \in V \times V^*$ is a finite set of *rewrite rules*. A rule $(a, \chi) \in P$ is written as $a \to \chi$, where $a \in V$ is a letter called *predecessor* and $\chi \in V^*$ is a word called *successor*. It is necessary that for any letter a there exists at least one word χ. If no rule is defined for a given predecessor, we define the *identity* $a \to a$ into the set of rules P. [20]

An example of using the L-system is to model the growth of the alga *Anabaena catenula*, which has two types of cells that form chains. These two cell types can be labelled as A and B in our alphabet. The initial state is A and the rewrite rule is $A \to AB$ and $B \to A$. So we start with A, for the first iteration AB, for the second iteration ABA, for the third $ABAAB$, and so on. For this example, the number of symbols in a given iteration is important. [20]

An interesting interpretation of the L-system is that the symbols can be read as commands to draw or represent parts of the plant model. This rendering is commonly interpreted using **turtle interpretation**. The turtle carries a state (x, y, α) where the Cartesian coordinates (x, y) represent the position and α angle (*heading*) represents the direction the turtle is facing. With the step size d and angle increment δ defined, the turtle responds to commands given by additional symbols. [20] For example, the following L-system:

$$\omega : F - F - F - F$$
$$p : F \to F - F + F + FF - F - F + F$$

specifies an approximation of a quadratic Koch island with an angle increment δ equal to 90 degrees, and the step length d decreases between iterations. Figure 1.3 shows the sequence of iterations. [20]

L-systems are a great source for generating vegetation such as grasses, trees, and plants. In figure 1.4 we show an example of plant generation. This kind of generation makes it easier for environmental artists to populate the space, as many similar artefacts that are recognizable but slightly different from each other are needed. For example, the SpeedTree[2] toolkit has been created to support game studios in helping procedurally generate vegetation in games and movies. This tool is used by major game companies to develop their AAA games, such as Activision, Ubisoft, CD Projekt Red, and Epic Games.

1.2.4 Superellipsoids

Up to this point, we have given the basic methods for generating possible contention in games, which can be found in any source on procedural generation. Most of these can be easily applied to terrain, level, dungeon, and vegetation

[2] `https://store.speedtree.com/`

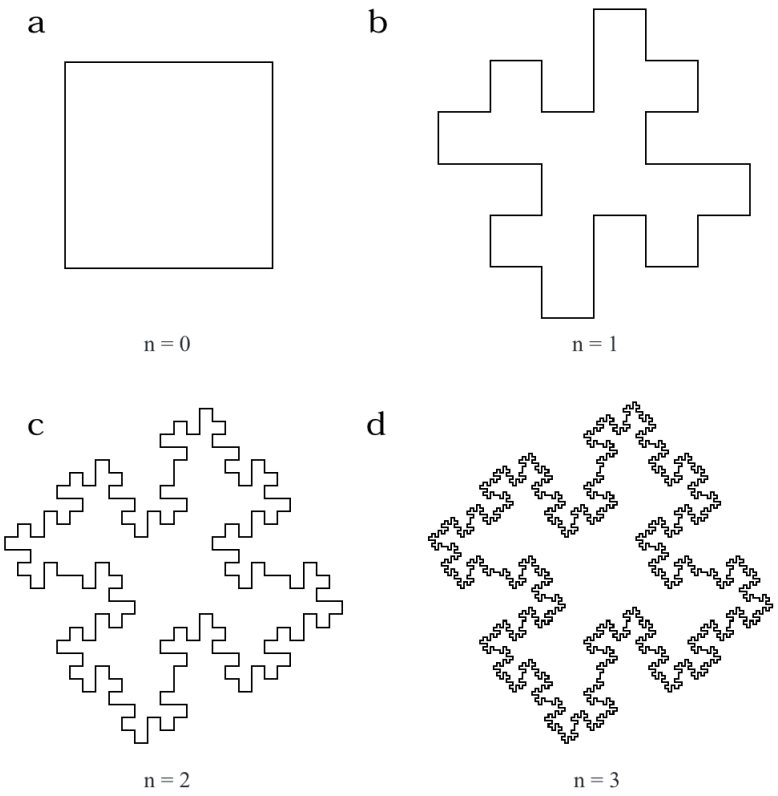


Figure 1.3 Example of generating a quadratic Koch island for the first 3 iterations.[20]

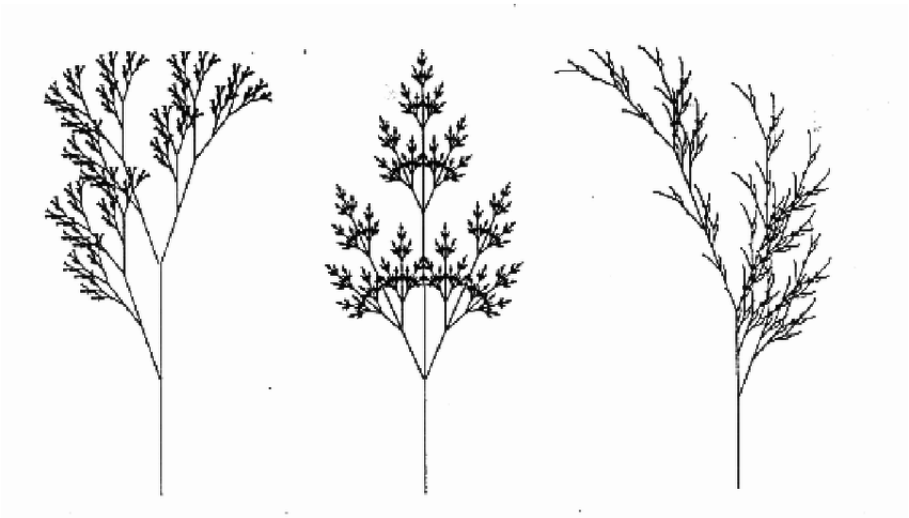


Figure 1.4 Example of L-system fractal plants.[21]

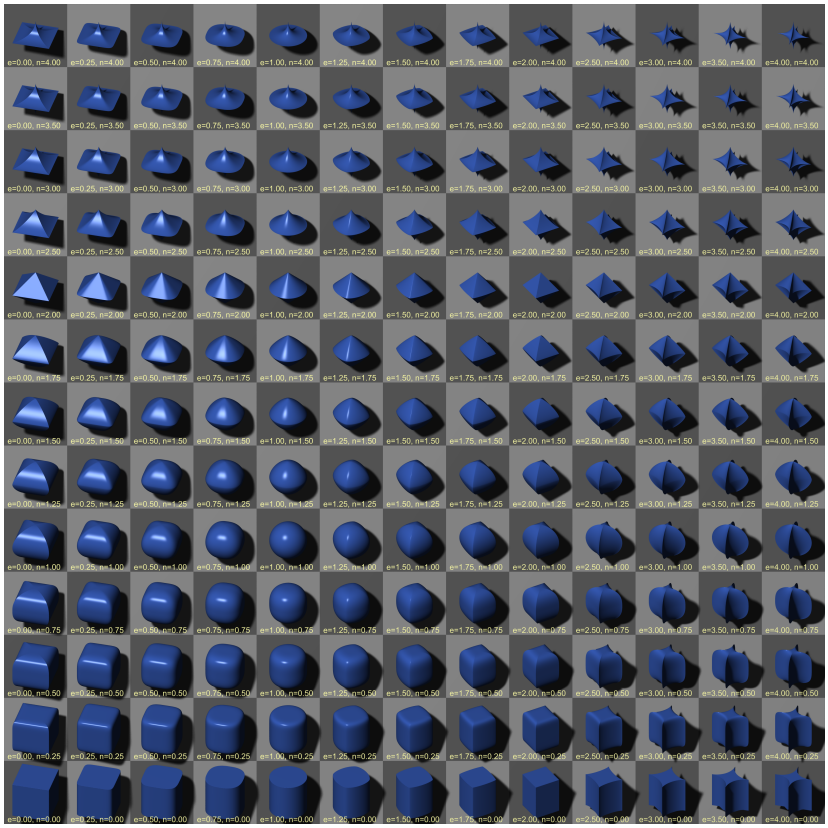


Figure 1.5 Superellipsoid collection with exponent parameters [23]

generation. However, in this subsection, we will look at how to generate meshes for game creatures with a not-so-common method.

Superellipsoids are part of the superquadrics family, which represent common shapes and are used in scientific visualisation, image analysis, graphical modelling, computer vision and robotics. In this work we will try to use this approach in mesh generation for game creatures. In the figure 1.5 we can see a set of different superellipsoids with exponent parameters. [22]

A superellipsoid[3] is an exposed ellipse in three-dimensional space. The shape of the superellipsoid is defined by two real numbers. Its parametric surface vector can be obtained from the spherical product of two superelipses and its implicit function is listed in the equation 1.6

$$\left(\left(\frac{x}{a_1}\right)^{\frac{2}{e_2}} + \left(\frac{y}{a_2}\right)^{\frac{2}{e_2}}\right)^{\frac{e_2}{e_1}} + \left(\frac{z}{a_3}\right)^{\frac{2}{e_1}} = 1 \tag{1.6}$$

where parameters a_1, a_2 and a_3 represent the size of the superellipsoid in the x, y and z dimensions and e_1, e_2 determine the shape. Using this equality, we are able to construct an inside-outside function 1.7

$$F(x;\Lambda) = \left(\left(\frac{x}{a_1}\right)^{\frac{2}{\epsilon_2}} + \left(\frac{y}{a_2}\right)^{\frac{2}{\epsilon_2}}\right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{z}{a_3}\right)^{\frac{2}{\epsilon_1}} \tag{1.7}$$

[3]Its description can be inferred from the name. The basis is a curve. The prefix *super-* indicates that the base curve is exposed. The suffix *-oid* indicates that it occurs in three-dimensional space. Superquadrics also include superhyperboloids (the original curve is a hyperbola) and supertoroids (the latter consists of trigonometric torus equations).

where x is the vector $x = (x, y, z)$ and $\lambda = (a_1, a_2, a_3, \epsilon_1, \epsilon_2)$ is the parameter vector. The inside-outside function gives us information whether the point x is inside ($F < 1$), on the surface ($F = 1$) or outside ($F > 1$) the elispoid. [22] We just need 8 parameters to define the superellipsoid. These few parameters give us the ability to make different shapes.

1.3 Entity Component System Architecture

Entity component system (ECS for short) is a software architectural pattern used to structure code and data. The ECS principle aims to create a modular system that can flexibly organize data and separate this data from the logic. As a result, ECS can enable efficient in-memory data management and a parallel environment for logical computations.

In contrast to the object-oriented approach, where several properties belong to one object that may not always be used, ECS is appropriately based on the principle of *Composition over Inheritance*[4]. The key to this architecture is that components and systems are independent. As the code base increases, ECS simplifies code refactoring and, therefore, makes the code more readable. Conversely, this model can be discouraged by having a much larger code base than in the classic object-oriented approach.

ECS defines 3 basic pillars: entities, components and systems.

Entities are unique identifiers often represented as integers. They do not contain any data or logic.

Components are represented as program data. They are linked dynamically to Entities ([24]). Components tend to be very modular; one component may represent a position, and another component a rotation.

Systems contain the behavior of the program. This is where all program logic occurs.

The figure 1.6 shows an example of an ECS architecture where entities A, B and C are bounded by the *Translation*, *Rotation* and *LocalToWorld* components. The defined system obtains data from the *Translation* and *Rotation* components, performs a multiplication over the data, and stores the result in the *LocalToWorld* components. Components can be added and removed at will without the need for additional code refactoring. Systems filter the entities by the specified components, and skip any entity that is missing.

So in this case, the *Renderer* component for *Entity A* and *Entity B* has no effect on the system. For example, if we remove the *Rotation* component from *Entity B*, the system will only run over *Entity A* without an error. This happens due to the independence of components and systems in this architecture and therefore the architecture is suitable for keeping the code clean, reusable, extensible and working with a large number of entities.

[4]The most common approach in game engines such as Unity is the component-based model (or also called the entity-component model) derived from object-oriented programming. This is the predecessor of ECS, where components are mapped to entities, except with the difference that components contain both data and logic over that data.

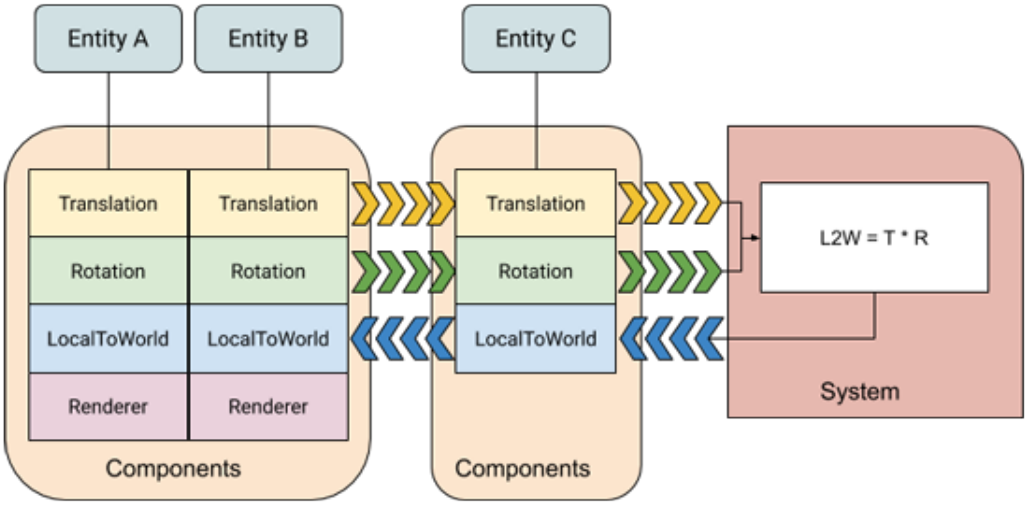


Figure 1.6 Example diagram of ECS architecture. [25]

```
struct Pt {
  float x;
  float y;
  float z;
};
struct Pt myPts[N];
```

(a) AoS

```
struct Pt {
  float x[N];
  float y[N];
  float z[N];
};
struct Pt myPts;
```

(b) SoA

Figure 1.7 Structure of Arrays and Array of Structures design patterns. [26]

Using ECS architecture can lead to the introduction of **data-oriented design**, where data is a key design element. This design is used with ECS to increase game performance in the gaming industry through efficient CPU cache utilization. Data-oriented design thus focuses on proper cache utilization and reducing cache misses. A common pattern for this data organization is **Structures of Arrays** (SoA), which groups arrays of homogeneous data so that they are easy to read and to use in a program. The opposite and traditional way to this is **Arrays of Structures** (AoS), where all data is organized in a single class, instances of these classes are grouped into arrays, and methods are called on each instance. [27] Figure 1.7 shows these concepts' differences.

1.4 Unity DOTS

Unity Data-Oriented Technology Stack (DOTS) is a set of technologies and packages that together provide game developers with a data-driven design approach to for developing games in the Unity engine. This suite allows Unity developers to take advantage of the **ECS architecture** compatible with GameObjects, the **Burst Compiler**, which translates from IL/.NET bytecode to highly optimized native code, and the C# **Job System**, which allows writing parallel code. These functionalities are divided into their own packages and in this section we will describe each of them.

1.4.1 Burst Compiler

Burst compiler allows you to improve the performance of your application. It is installed through the Unity Package Manager and is primarily designed to work with the Unity Job System. It is an LLVM-based compiler that optimizes Unity job's system code. [27] The Burst system compiles code in different ways that depend on the context.

Burst compiles **just-in-time (JIT)** code when the application is enabled in the Editor in Play Mode. The code is compiled asynchronously. Code can also be compiled synchronously in the Editor using the appropriate annotation on the Job definition. When the project is built in Unity, Burst compiles all **ahead-of-time (AOT)** code into the native library that Unity provides with the resulting application. [28] The key difference between the Unity compiler and the Burst compiler is that the Unity compiler uses the Mono runtime and Just-In-Time (JIT) compilation, or IL2CPP for converting IL to C++ for execution, which involves runtime dependencies and limited real-time optimizations. [29]

Burst compilation has several limitations and requirements to be followed to use it. [28] Among them are:

1. You can't use managed objects or reference types: char, decimal, string, managed arrays (instead, you need to use a native container such as *NativeArray*

2. You cannot use Enum methods, such as `Enum.HasFlag`.

3. Cannot access *GameObject* or *Component* code, so better to use with ECS architecture.

4. Cannot write static variables

5. No support for exception catching (it is possible to throw a condition but only in the Editor).

Using the Burst compiler is very convenient, but you must be aware of its limitations. Another difficulty can arise when developing using the Unity ECS architecture. The Entities package (the Unity package that provides a data-oriented implementation of the ECS architecture) can provide operations incompatible with the Burst compiler (when C# features beyond the aforementioned Burst limits are used). In this case, it is recommended to specify a `WithoutBurst` function when creating the Job. [25]

1.4.2 Job System

The Job system is used to create multithreaded code so that the application uses all available CPU cores to execute it. Multithreading provides increased application performance because regular code, without using the job system, runs on a single **main thread** synchronously. Splitting the process into several smaller chunks (**jobs**, a unit of work that performs a specific task), running them on multiple cores and processing them in parallel utilizes core capacity much more efficiently and delivers massive performance. [29]

For multithreading, Unity has its own native job system that depends on the number of CPU cores available on the device. Developers can schedule as many tasks as they need, and the job system itself will ensure that there are enough **worker threads** available, which the CPU core capacity provides. Thus, there is no need to know how many CPU cores are available for task management. The job system also includes its own safety system, which captures potential race conditions (which commonly occur when two parallel operations are run over shared data) and related bugs. With this system, developers can safely and easily write parallelizable code. [29]

Part of the scheduling strategy is **Work stealing**, where when a worker thread processes its tasks the fastest, it takes the tasks that are assigned in the queue of another worker thread for processing. [29]

The job system is used by the Entities package. It is also recommended to use jobs in code wherever possible.

1.4.3 Entities

The Entities package allows developers to take a data-driven approach with the ECS architecture. The use of this pattern in Unity is different in principle and implementation than the object-oriented approach.

The basis for ECS in Unity is the subscene. The subscene contains all the content in the application under development. When *GameObject* and *MonoBehaviour* components are added to this subscene, they will be converted into ECS entities and components using Bakers. [25]

Baking

Baking is the most important process in Unity DOTS. *GameObject* data contains both authoring and runtime data. Although this data model provides some flexibility, it is not required for runtime, and since Unity processes both at the same time, performance may be degraded. However, Unity ECS is designed to represent data much more efficiently. It splits data into two types:

Authoring data: Any data created during the editing of the application. Scripts, assets, etc. It is readable for humans.

Runtime data: ECS processes this at runtime. It is optimized for performance and storage efficiency. Readable for computers. [25]

The baking process transforms authoring data into entities as runtime data. This happens in the Editor only. Baker is instantiated once and its `Bake` method is called multiple times in an unpredictable order. Since incremental baking can occur over extended periods, bakers must be stateless and access data solely through methods, avoiding value caching to prevent issues. [25] The following code snippet is an example of creating an authoring data and its baker:

```
[MaterialProperty("_Size")]
public struct RenderSize_C : IComponentData
{
    public float Value;
}

public class RenderSizeAuthoring : MonoBehaviour
{
    public float value;
}

public class RenderSizeBaker : Baker<RenderSizeAuthoring>
{
    public override void Bake(RenderSizeAuthoring authoring)
    {
        AddComponent(new RenderSize_C()
        {
            Value = authoring.value,
        });
    }
}
```

Listing 1.2 Definition of size component in Unity DOTS project.

Authoring class must be inherited from MonoBehaviour and Baker inherits from the Baker class. The class must be saved in a file named `RenderSizeAuthoring.cs` for the Baking process to work. The Bake method is called for each authoring component that is marked for baking. In the case of **full baking**, all authoring components in the authoring scene are baked. It is also possible to perform **incremental baking**, which bakes those components that have been modified or their dependencies modified. [25]

Archetypes

What happens is that we can have many entities that have a unique combination of component types. This is called an **EntityArchetype**. These are used to group a combination of components under a unique identifier for entities. Consider

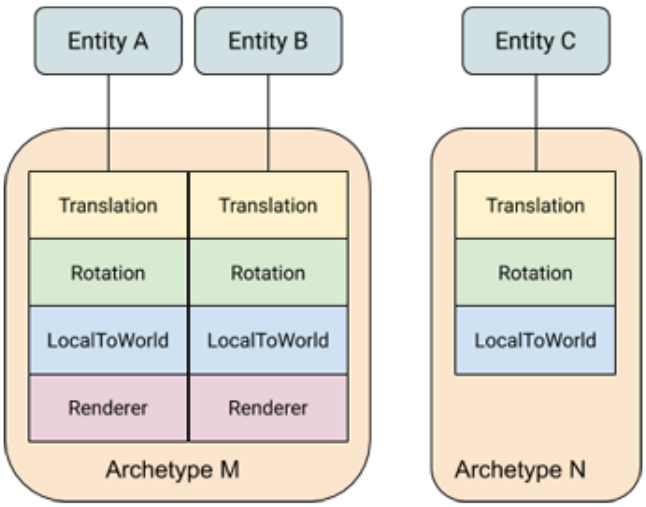


Figure 1.8 Visualisation of entity archetype. [25]

the diagram describing the ECS architecture from section 1.3. In this example, because Entity A and Entity B have the same components, they are grouped into one Archetype and the components of Entity C into another Archetype. If a Renderer component were removed from Entity A or Entity B, the entity's Archetype would change to the same as that assigned to Entity C. [25] The example can be seen in the figure 1.8.

The archetype of an entity dictates the storage location of its components in ECS. Memory is allocated in *chunks*, each represented by an **ArchetypeChunk** object, and a chunk contains only entities of the same archetype. When a chunk is filled, ECS allocates additional memory chunks for new entities of the same archetype. If an entity's components are added or removed, altering its archetype, ECS transfers the entity's components to a new chunk. [25]

Aspects

We can also group components into a single C# struct. Aspect is used to organize component code and simplify queries in systems. In the code snippet 1.3, we define an **Aspect** with several rendering components. According to the documentation, the Aspect definition must be a `read only partial struct` and implement the `IAspect` interface. [25] Code snippet 1.4 shows how to use such an Aspect when filtering entities in the system. Instead of writing down all the components from code snippet 1.3, we can use only this Aspect that encompasses the required components.

```
public readonly partial struct SpawnBeautyRenderAspect : IAspect
{
    public readonly Entity Self;
    public readonly RefRW<RenderColor_C> Color;
    public readonly RefRW<RenderMass> Mass;
    public readonly RefRW<RenderAdornmentSize_C> AdornmentSize;
    public readonly RefRW<RenderAdornmentShape> AdornmentShape;
    ...
}
```

Listing 1.3 Example of Aspect in Unity DOTS project.

```
foreach (var (_, s) in SystemAPI.Query<SpawnBeautyRenderAspect>()
                         .WithAll<Spawn_C>()
                         .WithEntityAccess())
{
    ...
}
```

Listing 1.4 Example of Aspect usage in system.

1.5 Graphics Pipeline

This work deals with generating graphical elements on the GPU, so it is necessary to familiarize ourselves with the graphics pipeline and its different elements. The graphics pipeline deals with rendering a 3D scene description on a 2D screen and the transformation between spaces and data conversion. This section focuses specifically on the general graphics pipeline and the usage in the Unity engine.

1.5.1 Unity Rendering Pipelines

Unity has several rendering pipelines available. [29] Pipelines available:

- **Built-In Render Pipeline** (default with limited customization options),
- **Universal Render Pipeline** (*URP*, which can be customized),
- **High Definition Render Pipeline** (*HDRP*, used for strong customization and options to high-fidelity graphics for high-end platforms).

Each pipeline is designed for different solutions in game development, and a listing of the differences between these render pipelines is provided in the Unity documentation[5].

1.5.2 Render pipeline structure

The rendering pipeline runs in the following steps [29]:

1. Culling, which decides what objects are rendered on the screen, e.g. frustum culling (removing objects outside the camera view), occlusion culling (hidden objects behind other objects)
2. Rendering, where objects with their set lights are rendered into pixel buffers
3. Post-processing, where pixel buffers are modified to generate the final output frame on screen, e.g. color grading, bloom, depth of field.

The pipeline consists of a sequence of operations where the input is vertices and mesh textures and the output is pixel color values on the render target. The figure 1.9 shows such a simplified sequence.

[5] https://docs.unity3d.com/Manual/render-pipelines-feature-comparison.html

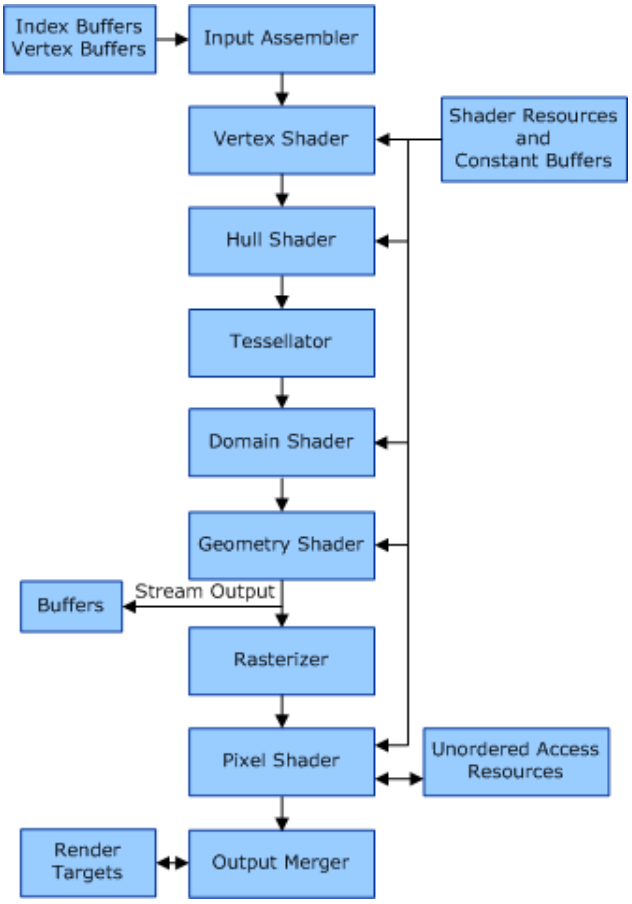


Figure 1.9 Overview of rendering pipeline in Direct3D version 11. [31]

At the start of this sequence, the input assembler collects all raw vertex data from the specified buffers. Subsequently, shaders start working. A **shader** is a program that runs on the GPUs and performs a computation. [30]

The **Vertex Shader** is triggered for each vertex and applies transformations between object space and screen space. It then moves its data further along in the pipeline. [30]

Hull Shader, **Tessellator** and **Domain Shader** are part of the **tessellation** pipeline. The tessellation process is optional and uses the GPU to compute a more detailed surface from a surface that was created from quad or triangle patches or isolines. The Hull Shader produces geometry patches that correspond to each input patch. In the Tessellator, a sampling pattern is produced that represents a geometry patch and generates a set of smaller primitives consisting of a triangle, point or line and these samples are connected. The Domain Shader calculates the vertex position for each domain sample. The use of tessellation is beneficial in memory saving, displace mapping capability, scalable rendering techniques and improved performance. [31]

Geometry shader runs over each primitive and can discard it or create more primitives than it received. [30] The **Stream Output** then connects the primitives from Geometry Shader to output buffers.

In the **Rasterizer** phase, the primitives are divided into fragments (pixels) that fill the framebuffer. This phase clips (stamps) fragments that are outside the raster or that are behind other fragments. [30] This section also implements the

viewport, rendering selection and primitive settings. [31]

The **Pixel Shader**, or **Fragment Shader**, is called over each pixel (fragment) that is available after all other phases and detects the color and depth value of that element. The output of this shader is a single pixel. [30]

The final stage of the render pipeline is **Output Merger**, which applies functions to render-target blend, depth and stencil operations. The pixel is then copied to the framebuffer, which is a buffer of the generated image and can be sent to the user's monitor. [31]

1.5.3 Writing shaders in Unity

In Unity, you can write any shaders that are part of the graphics pipeline. It is also possible to write **Compute shaders** that run on the GPU but outside of the normal rendering pipeline. Compute shaders are used for parallel algorithms and often require a thorough knowledge of the GPU architecture to be effective.

There are two approaches for writing shaders. One is using the **Shader Graph**, which is a Unity tool for creating shaders without writing code. It involves creating nodes in the graph framework and linking them together. Each node represents an operation, function or property. In Figure 1.10, we created a lit shader graph asset (a pre-made asset from Unity) and changed the color using the `Color` node. This example can be found in the project code, specifically in `Assets/Shaders/ShaderGraph`.

The advantage of using Shader Graph is its simplicity and the lack of programming knowledge. Changes are propagated real-time, so developers can recognize the effect immediately. But for that price, ShaderGraph can produce performance overhead compared to manually written shaders. Also, some more advanced settings may be more complicated than writing it manually.

The other approach to creating shaders in Unity is **ShaderLab**. This is a Unity-specific language for writing shaders by hand. It also generates output from the Shader Graph into this type. In this thesis we will write shaders in ShaderLab for more flexibility, convenience and challenge.

1.6 Chapter summary

In this chapter we have introduced some approaches to procedural generation of visuals in games that can be used. We have found that using a noise function to generate fractals is very common. We have also explored the use of L-systems for natural similarity and added superellipsoids to these types of generation, which are interesting to explore for content creation.

We further explored the Entity Component System (ECS) architecture. We described its different parts and how they can interact with each other. Also important with this architecture is the Structures of Arrays programming pattern. We found that the use of ECS is very much related to the data-centric approach and is used to make the CPU cache more efficient and game performance. This architecture is part of the Unity DOTS suite of technologies, for which we learned about its other sub-tools: Burst Compiler and Job System, which provide application performance enhancements and the ability to create multithreaded code.

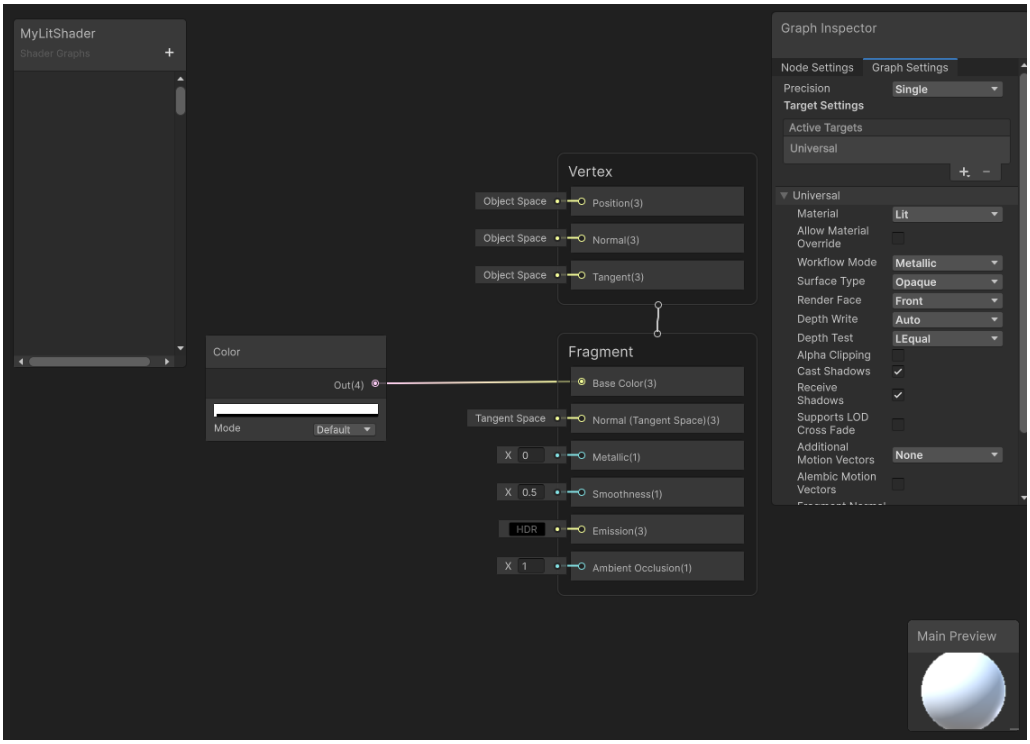


Figure 1.10 Shader graph tool.

Finally, we defined the graphics pipeline and how to write shaders in Unity. Unity has several rendering pipelines that can be used for development and have various advantages. The Shader Graph visual tool is used to create shaders in Unity, however for the purpose of this thesis, shaders will be written manually using ShaderLab.

2 Game design

This chapter is dedicated to information about the game with the working title *"Game of Life"*[1]. All the information is based on the game design document, written by the author of the thesis and the head of the game development team, Jakub Holík.

2.1 Motivation

We are fascinated by the splendour of nature's diversity. From small, absolutely simple steps and procedures, we can create a complex system. As humans, we are growing and learning all the time. We build knowledge, careers, relationships, and families. We set goals and strive to meet them. We live and survive. Motivation is based on 4 principles based on the personal beliefs and ideas of the author and the development team:

1. **Our possibilities are limitless**. The sky is the limit. We can program anything we can think of in programming, even if we don't know what to do. We can cope with difficult situations. Whatever we do has a consequence.

2. **The beauty of mathematics**. Mathematics is based on the universe. Everything in our lives contains numbers, calculations and relationships. Using mathematics, we can express a relation. We can use it to describe a powerful model. We can describe rules and dependencies. We want to let people express themselves through mathematics.

3. **Breaking problems down into sub-problems**. Every problem can be decomposed until we get to the atomic problems. They allow us to solve the initial problem in small steps.

4. **Humans**. People have ideas, abilities, and experiences to contribute to the world. They are incredible creatures that have a wide imagination. Let's give them the freedom to be creative or to try things out. What they come up with, they can show to other human beings. Together, they can build something extraordinary.

Based on this knowledge, we decided to create a game that shows the player these fantastic aspects of our world. The game design provides the following concepts:

- Strategy can be varied. In this game, it's not just about being the strongest or the fastest. It may also be about getting older more slowly or being difficult to eat. To make a refuge and sleep through challenging weather and so survive. Allowing the player to analyze the ecosystem of each run to find a niche their species will fit and thrive in.

[1]The working title is based purely on motivation. It does not refer to Conway's Game of Life, a cellular automaton whose behaviour resembles the evolution of a community of living organisms.

- Create rules using the tools provided, from which more complex rules can be built. These are based on mathematics. The player can use them to construct behaviours that contribute to his strategy.
- Build a diverse system from smaller steps. Complicated behaviours can be assembled from player-defined behaviours, broken down or combined into more sophisticated or simpler elements.
- Share the player's experience and ideas. Anything a player does can be shared with other players who can appreciate the creativity and thoughts behind the solution.

2.2 Game overview

It is a 3D top-down evolutionary strategy game that focuses on analyzing the environment and choosing an evolutionary path that can lead this species to successful colonization of said environment. Players can accomplish given goals. After completing the objectives, they can save their species as part of the game and other players will have a chance to encounter that species. To evolve their species, players have several tools for analyzing the environment and acting upon gathered knowledge.

To analyze the environment, players can do the following:

1. Change their perception of the environment.
    - This is due to changing the rendering of the game world so that the player can discern important information. We call these renderings *Lenses*.
    - Within the environment, the player can choose the displayed information available. We call these sets *Perspectives*. This is a display of all information, called a *God Perspective*, used to analyze the entire environment. The second view, called the *Creature Perspective*, shows the player only the information available through their creature. This method helps to analyze the behavior of your own creature towards the environment.
2. Thoroughly examine any game entity. The player can discern what state the entity is in and what it is doing. The player can look at the numerical values of his creatures.
3. The player uses customized notifications to know about important events.

To evolve his or her creature type, the player can do the following:

Adapt a species. Players have a certain amount of Adaptation points for which they can buy Adaptations. These can add new Actions to creatures of their species, change values of attributes or enhance senses.

Change a species behaviour. Players can set the behaviour of creatures of their species in different situations. This can happen using the Behaviour editor.

Define relations between values in the game. This is the core concept of the game. With Expressions, player can define for example a new Attribute, which value will be calculated from values of other Attributes (the more something is red, shiny and rather small, the more foodlike it is) or they can define a Motivation for choosing a certain Behavior (the more hungry a Creature is the more motivated it is to start searching for food). This can happen using the Expression editor.

2.3 Technical overview

The game is **data-driven**. Individual game elements are treated as assets that can be modified by the player and shared among other players through the online system. Such elements are Lenses, Attributes, Behaviour, Species and Expressions, as introduced in the previous sections.

A level consists of Pieces, which are game entities visible to the player. Each Piece has its own attributes. Creatures, inherited from Piece, can make behavioural decisions. The player develops his own Species and can apply the adaptations and behaviours he creates to these. Each player can have many creatures in their Species.

Each object may or may not have multiple attributes. This may cause some entities to have a low number of attributes, but others may have too many attributes. The most suitable architecture for this game is the **Entity Component System architecture**.

The game is developed in Unity DOTS, in C# programming language.

2.4 Visual design

The game's visual aspect is strongly inspired by **nature** and its diversity. Nature provides different shapes with unique patterns that can be simple yet very appealing. At the same time, it must display the entity's current state for the player to be able to analyze. Therefore, the graphical part must be simple but varied for the player.

The appearance of the game can be divided into two types: Beauty Lens and Attribute Lens.

The **Beauty Lens** shows the world in all its attractive glory. Based on this, the player can distinguish the size, colours, rigidity, or even the creature's mass. The design is abstract, with concrete elements that are manually created and animated. These concrete elements are called **Stickers**, which are stuck to the game object to represent a specific part of the entity, e.g. eyes, ears, horns, and tail. Stickers are arbitrarily glued onto the entity by the player, and the player can decide whether to render these elements at all. This rendering is the dominant way to represent the world.

The **Attribute Lens** helps the player analyze the world more thoroughly. It displays individual attributes and their values in a way that is clearly discernible to the player. Thus, the graph plotting method, such as radar chart (Figure 2.1) or coxcomb chart (Figure 2.2), is recommended. The player can compare the creatures with each other and can tell at a glance how similar they are.

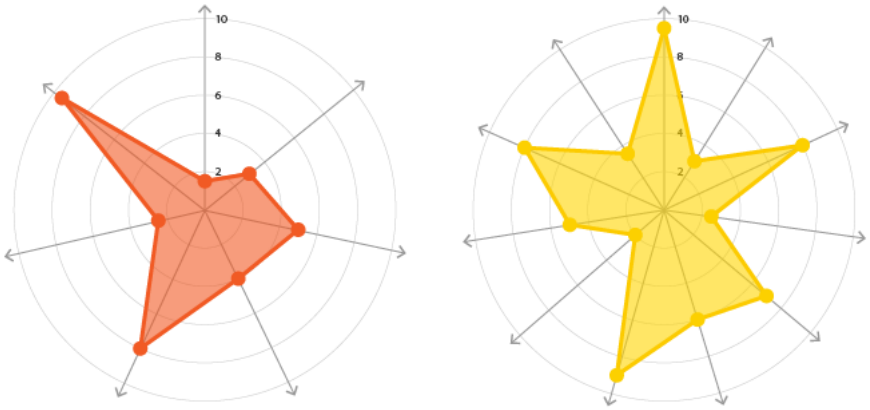


Figure 2.1 Examples of radar chart. [32]

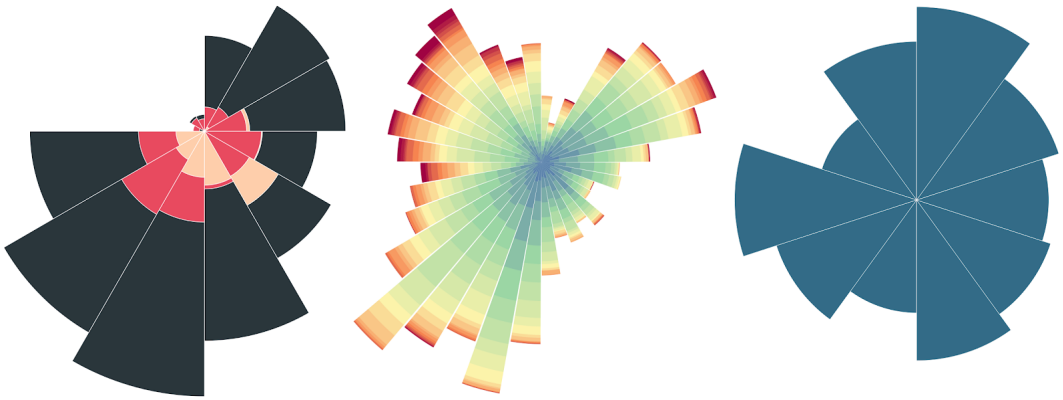


Figure 2.2 Examples of coxcomb chart. [33]

Figure 2.3 Visual target

It is desirable that the visuals of the game be generated in addition to Stickers.

The figure 2.3 shows the design for Beauty Lens created in 3D Max by the project leader, Jakub Holík. Each object consists of smaller particles called **adornments**. The height, width, shape and colour of each object can be distinguished.

2.5 Visual requirements

Based on the design document, visual requirements related to game visuals were written.

VR01: The desired appearance must be continuous. Attributes change over the course of the game, and a slight change in the value of one attribute must not significantly change the appearance of one Piece nor entire game world.

VR02: The visual appearance must show the state of the individual entities. The player can analyze the environment and his creatures through this display.

VR03: Abstract visual elements should be complemented by concrete visual elements. These concrete elements are called *Stickers* and are used to recognize the entity by player in more detail.

VR04: The generation of the appearance must be robust. It is possible to create a sufficient variety of different shapes. At the same time, it must not produce nonsensical shapes for given inputs, i.e. arbitrary combinations of attribute values must always produce a meaningful shape.

VR05: Appearance generation must be efficient and real-time. Changes to attribute values are reflected in the visual immediately.

VR06: The appearance generation process should be extensible. It is possible to slightly modify or improve the generation method. Individual parts can be reusable.

VR07: The shape of creatures must be 3D.

2.6 Chapter summary

This chapter outlined the game's design, tentatively titled Game of Life. The game is a data-driven, 3D top-down evolutionary strategy inspired by natural processes and human cognition. The objective is to complete tasks and develop an efficient species of creature. We discussed various game concepts and the architecture based on the Entity Component System (ECS) principle. Additionally, we detailed the desired visual aesthetics, divided into two categories: Beauty Lens and Attribute Lens. The Beauty Lens features abstract visuals enhanced with Stickers that add concrete elements. The Attribute Lens includes graph elements that allow players to analyze the state of the objects.

3 Experiments

This chapter deals with experimenting with visuals for the game. Several methods were used in this thesis and tested to meet the visual requirements. We then select the most suitable solution to implement in the next chapter.

3.1 2D Shape Interpolation

One of the most straightforward ways to create a mesh is to interpolate between existing shapes. We created several 2D shapes (square, circle, triangle, pentagon) to interpolate between. For this purpose, we will use the `Lerp` function provided by the Unity Mathf library. The figure 3.1 shows the example of interpolation. This method is insufficient, but we can combine it to generate terrain and creatures as unique visual additions. Interpolating shapes also gets us closer to the visual target, and we will call these additions **adornments** as we have defined them in the 2 Game Design section.

3.1.1 Adornments

For generation purposes, let's make it more interesting. Let's generate multiple of these adornments within a single mesh. For each vertex, we'll generate its own adornment using the geometry shader. This means we generate four more vertices for each vertex, which we position around the original vertex and set to height. We can randomize the position of the adornments to a certain distance and influence it with the displacement map. Figure 3.2 shows an example of such adornments applied to a plane mesh.

3.2 Golden ratio

As an experiment for a natural look, we tried using Golden Ratio. Golden ratio is the relationship between two numbers if their ratio is the same as the ratio of their sum to the larger of the two quantities. This variable is denoted by the Greek letter Φ and indicates the number 1.618…. The Golden Ratio is related to the Fibonacci numbers, where ratio of two subsequent numbers of the sequence is very close to the Golden Ratio.

Figure 3.1 Shape interpolation. The interpolation starts with a circle to a pentagon (the shape on the left), then from the pentagon to a square (the shape in the middle) and from the square to a triangle (the shape on the right).

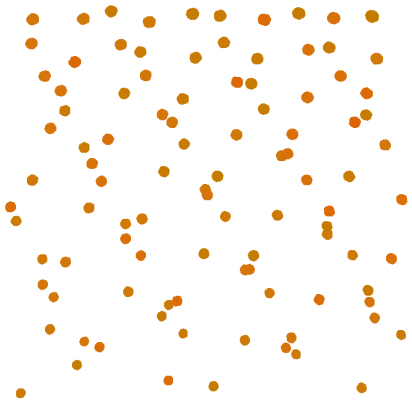


Figure 3.2 Adornments.

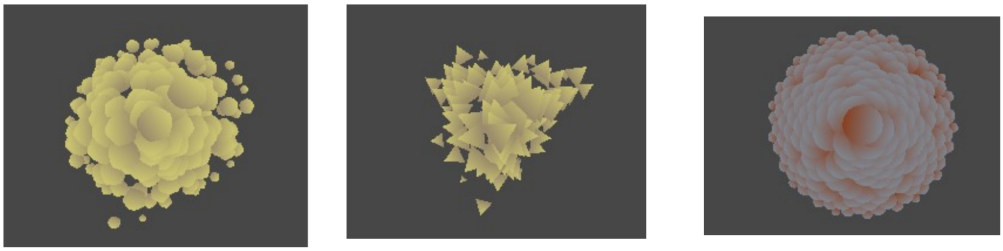


Figure 3.3 Generation using Golden ratio and 2D shape interpolation.

The beauty of the golden ratio lies in its depiction of natural aspects. Tree leaves and pine cone seeds grow in a patterning that is around the golden ratio and the petal spirals approach Φ value. This motivated us to try spirals in generating our visuals as well and the result can be seen in figure 3.3. Indeed, we are able to generate spirals that give us natural aspects, but the resemblance to plants is very concrete and can be distracting when large in number. Nevertheless, it is a very interesting idea to generate at least 2D creatures or some plant object in an environment.

3.3 Fractals

Fractals, a natural phenomenon that rely on self-similarity, as described in Chapter 1 Analysis, section 1.2.2 Fractals, were also explored for generating visuals. The advantage of fractals is that they give infinitely many possibilities to explore. The problem is that if we create a fractal, such as a Mandelbrot set, containing values from the attributes of a given environment or creature, a slight change in one attribute would cause the fractal to change significantly. Although fractals can be fascinating to look at visually, we need something continuous into which we can project attribute values, especially for generating creatures. Fractals deepen their detail through their iterations, which is not appropriate for creature generation.

The question may be whether we can use fractals to generate a level environment. In the picture we have a generated part of a Mandelbrot set, created in

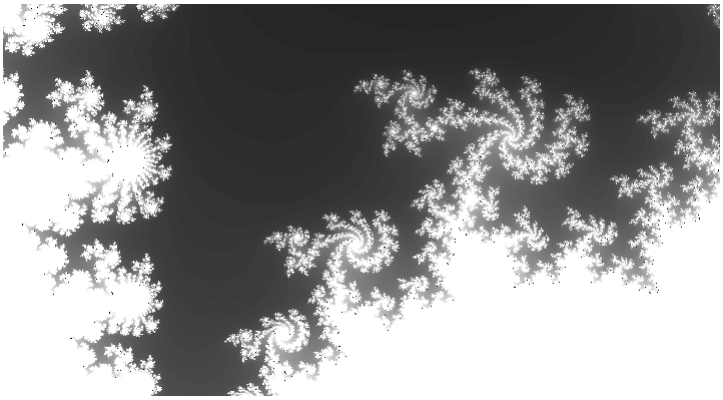


Figure 3.4 Subset of the Mandelbrot fractal.

ShaderToy[12]. Such a visual could be used for a 2D space. However, in the context of game design, each level fractal needs to exhibit a logical sense of place, why it appears there and what is in that particular place.

3.4 L-Systems

L-systems may provide the possibility of generating natural phenomena in the landscape or the possibility of growth of the generated creature. As for fractals, this is inconsistent with the visual requirement for robustness, as the generated L-system looks similar to other attribute values. Also, like fractals, they have the problem of not operating in a continuous space, i.e., if the value of one attribute is changed slightly, the generated shape is reshaped entirely.

3.5 Superellipsoids

This thesis challenges the use of super ellipsoids. In terms of shape generation in 3D space, it seems particularly suitable for creature generation. We can control the shape using only a few parameters.

As part of this thesis, we have tested an algorithm to generate a superellipsoid based on a paper [22] by Paulo Ferreira (2018). The first step is to generate a point cloud. Thanks to the paper mentioned above, we can also obtain each vertex's normals to the generated point cloud. We then process these on the GPU to create the creature's imaginary skin.

3.5.1 Point sampling

To sample points, we need to use the superellipse description. Superellipses can be described as

$$x(\theta) = \begin{bmatrix} a\cos^{\epsilon}(\theta) \\ b\sin^{\epsilon}(\theta) \end{bmatrix} \tag{3.1}$$

[1]Code available: `https://www.shadertoy.com/view/XXyXR1`

[2]ShaderToy is an online community tool for writing shaders using WebGL and share them between other users. It is suitable for quick experimenting, learning and visualising.

where a_1 and a_2 are the semiaxis and ϵ is the roundness parameter. We use the relation to calculate the $\Delta_\theta(\theta)$ as described by Pilu and Fisher (1995) [34]

$$\Delta_\theta(\theta) = \frac{D(\theta)}{\epsilon}\sqrt{\frac{\cos^2(\theta)\sin^2(\theta)}{a^2\cos^{2\epsilon}(\theta)\sin^4(\theta) + b^2\sin^{2\epsilon}(\theta)\cos^4(\theta)}} \tag{3.2}$$

where $D(\theta)$ is the arclength and can be set to a constant. The angles of θ are obtained by iteratively updating θ_i in a dual manner:

$$\theta_i = \theta_{i-1} + \Delta_\theta(\theta),\ \theta_0 = 0,\ \theta_i < \frac{\pi}{2} \tag{3.3}$$

$$\theta_i = \theta_{i-1} - \Delta_\theta(\theta),\ \theta_0 = \frac{\pi}{2},\ \theta_i > 0 \tag{3.4}$$

where first, we increment θ from 0 while to less than $\frac{\pi}{2}$, and for the second, we decrement from $\frac{\pi}{2}$ to 0.

The author of the paper, Paulo Ferreira (2018), used these relations and adapted them for 3D space, i.e., for both supervises used as spherical products of superquadrics. The definition of the superellipsoid as the spherical product of two superellipses is

$$r(\eta,\omega) = \begin{bmatrix}\cos^{\epsilon_1}\eta \\ a_3\sin^{\epsilon_1}\eta\end{bmatrix} \otimes \begin{bmatrix}a_1\cos^{\epsilon_2}\omega \\ a_2\sin^{\epsilon_2}\omega\end{bmatrix} = \begin{bmatrix}a_1\cos^{\epsilon_1}\eta\cos_2^{\epsilon}\omega \\ a_2\cos^{\epsilon_1}\eta\cos_2^{\epsilon}\omega \\ a_3\sin^{\epsilon_1}\end{bmatrix}, \tag{3.5}$$
$$-\frac{\pi}{2} \le \eta \le \frac{\pi}{2},\ -\pi \le \omega < \pi$$

To sample η angles for the first superellipse according to the paper, we substitute for the equation 3.5

$$\theta = \eta,\ \epsilon = \epsilon_1,\ a = a_1,\ b = a_2 \tag{3.6}$$

and for the ω angles, we substitute

$$\theta = \omega,\ \epsilon = \epsilon_2,\ a = 1,\ b = a_3. \tag{3.7}$$

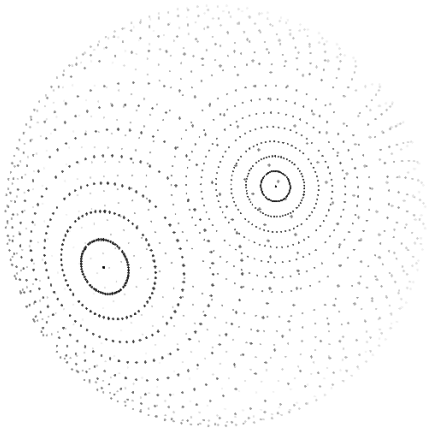
The superellipsoids are symmetric concerning the three axes, so we only sample from 0 to $\frac{\pi}{2}$ and mirror the result.

We will put this knowledge into practice. When generating points, we create two arrays of calculated θ, one for parallel and one for meridian, where the parameters for parallel are 3.7 and for meridian are 3.6. We use the function to update θ, described in equation aquation 3.2 and aquation 3.3, which returns the Δ_θ. We do this update in an iteration until the newly computed θ is greater than $\frac{\pi}{4}$ (we use the `Clamp` method on these values afterwards to keep the interval $(0, \frac{\pi}{2})$).

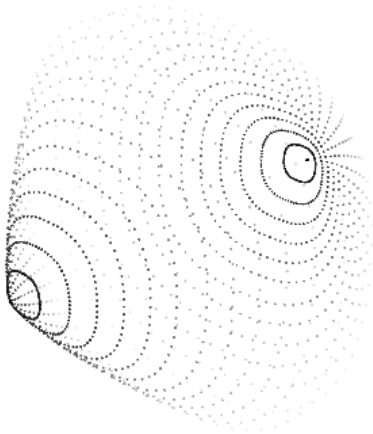
Having obtained these θ values, we can now move on to sampling the points using the equation 3.5 relation to calculate the coordinates on each axis. We should note that the exponentiation using epsilon is a signed power function, thus

$$\cos^{\epsilon}(\theta) = sign(\cos\theta)|\cos\theta|^{\epsilon}. \tag{3.8}$$

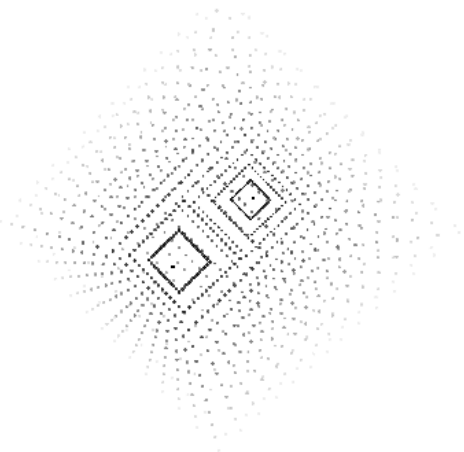
The figure 3.5 shows an example of generated superellipsoid point clouds. Since we control the shape with parameters ϵ_1 and ϵ_2, we get a unit sphere in Figure 3.6a by defining these parameters $\epsilon_1 = \epsilon_2 = 1$ and $a_1 = a_2 = a_3 = 1$.



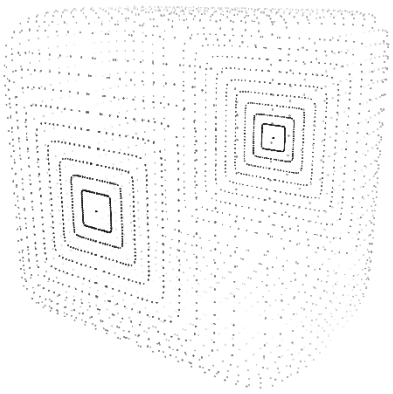
(a) $\epsilon_1 = 1, \epsilon_2 = 1, D = 0.1$



(b) $\epsilon_1 = 0.8, \epsilon_2 = 1.8, D = 0.1$



(c) $\epsilon_1 = 2, \epsilon_2 = 2, D = 0.1$



(d) $\epsilon_1 = 0.15, \epsilon_2 = 0.5, D = 0.1$

Figure 3.5 Superellipsoid point sampling for various parameters

3.5.2 Normal calculation

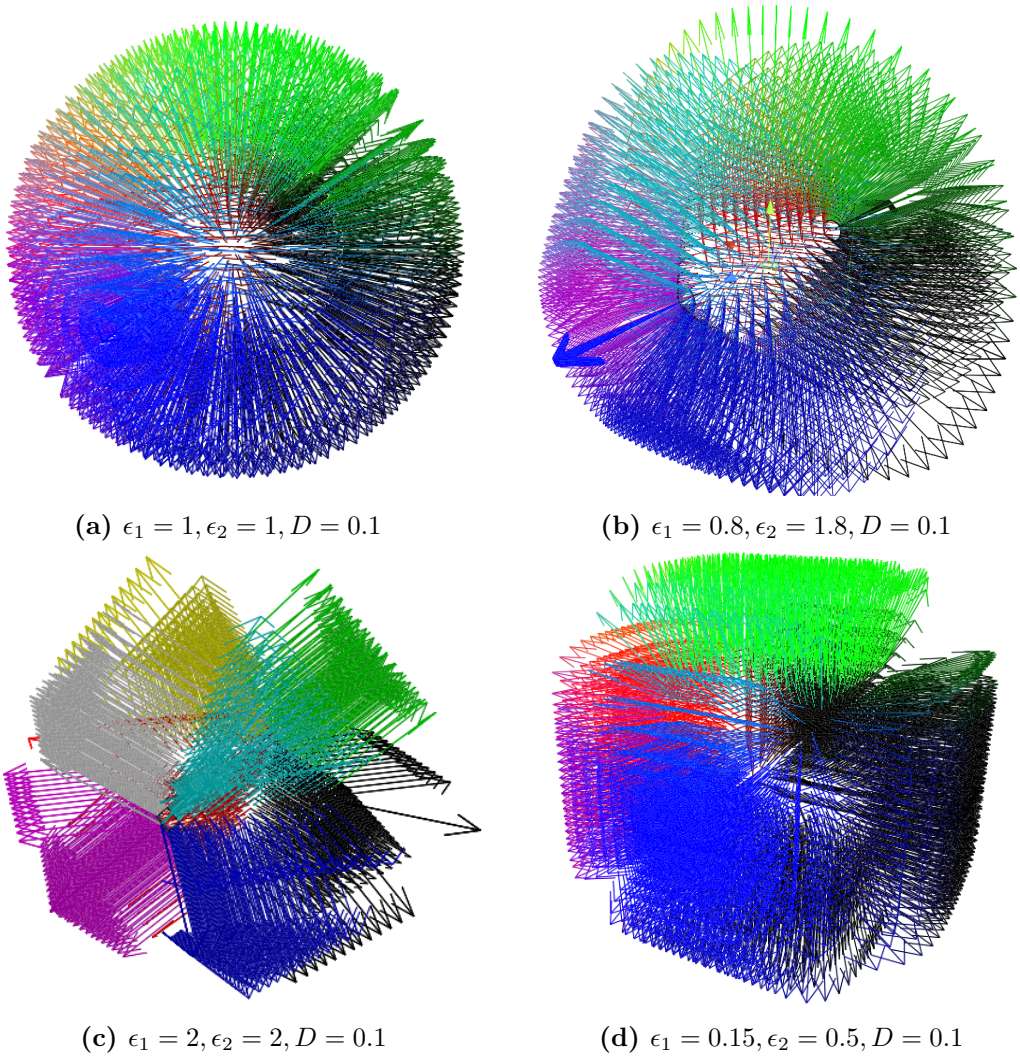
To compute the normals for each vertex, we use the relation obtained from the paper by Paulo Ferreira (2018):

$$n(\eta, \omega) = \begin{bmatrix} \frac{1}{x}\cos^2\eta \ \cos^2\omega \\ \frac{1}{y}\cos^2\eta \ \sin^2\omega \\ \frac{1}{z}\sin^2\eta \end{bmatrix} \tag{3.9}$$

Figure 3.6 shows the calculated normals. As can be seen, the computed normals are not optimal, and the non-uniform distribution of vertices can be seen as well. This can cause us problems when setting certain parameter values.

3.5.3 Shading

Now, for the calculated point cloud and normals, we are able to give this shape a ”skin.” We will use the adornment rendering method from section 3.1.1 Adornments, where we generate one adornment for each vertex in the compute shader. Examples of already-known shapes can be seen in Figure 3.7. The different shapes have



(a) $\epsilon_1 = 1, \epsilon_2 = 1, D = 0.1$ **(b)** $\epsilon_1 = 0.8, \epsilon_2 = 1.8, D = 0.1$

(c) $\epsilon_1 = 2, \epsilon_2 = 2, D = 0.1$ **(d)** $\epsilon_1 = 0.15, \epsilon_2 = 0.5, D = 0.1$

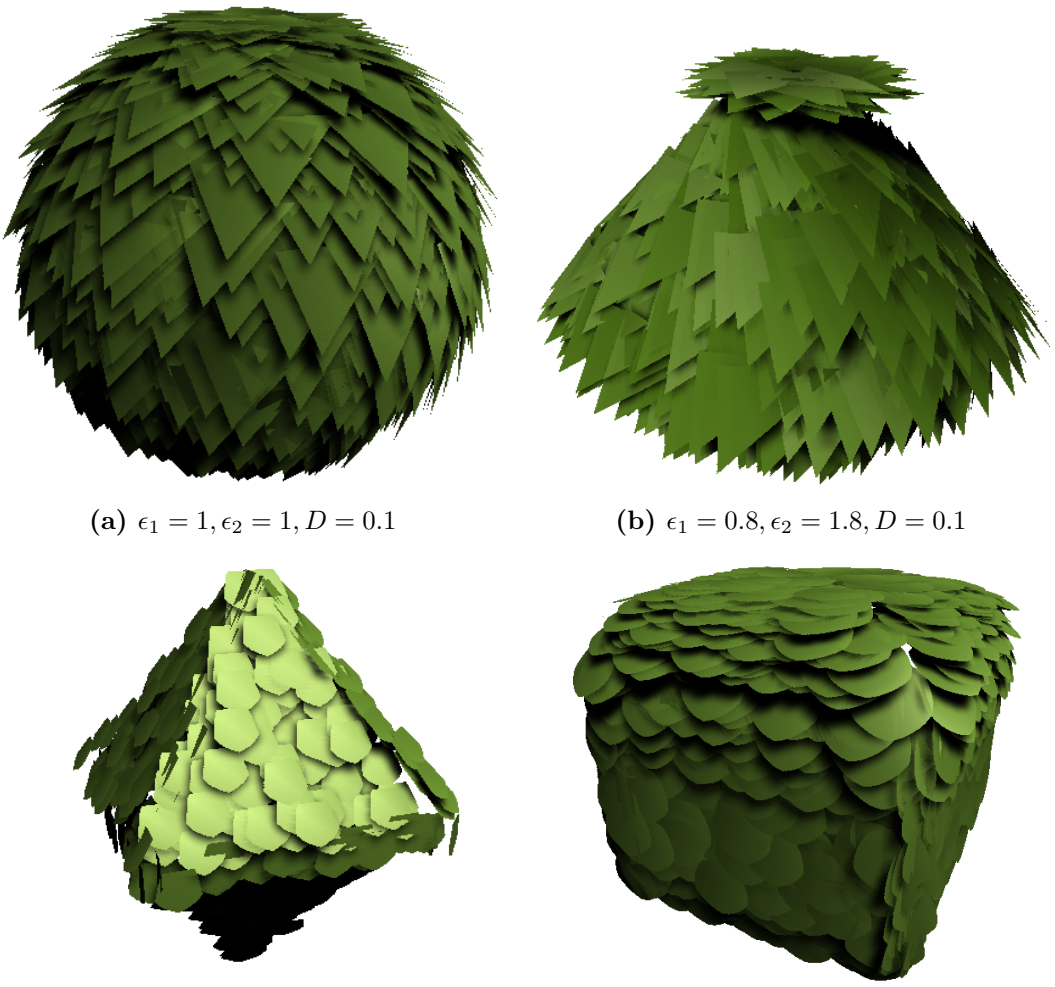
Figure 3.6 Superellipsoid normal sampling for various parameters

different settings, such as adornment shape, size and adornment variability, to show the variety between these shapes. We can see that shape 3.7d contains a few holes. Again, these holes are due to the non-uniform distribution of vertices.

3.5.4 Results

Creating a point cloud and calculating the normals may take more time for more extensive sampling. We measured the time of the initial creation and the update time of the existing superellipsoid instance, and the result is in the following table 3.1. Measurements were performed in Unity version 2023.3.0b5 on hardware: CPU Processor 13th Gen Intel(R) Core(TM) i7-13700F, 2100 Mhz, 16 Core(s), 24 Logical Processor(s) and GPU NVIDIA GeForce RTX 4070 Ti.

The initial generation takes around three to four milliseconds, and each frame update takes around two milliseconds. This is all assuming $D = 0.1$. However, we would like to know if we can safely decrease (or increase) this parameter with no performance dept. Fewer vertexes are sampled as this parameter increases, which helps performance, but the algorithm's performance suffers with more extensive vertex sampling.



(a) $\epsilon_1 = 1, \epsilon_2 = 1, D = 0.1$ **(b)** $\epsilon_1 = 0.8, \epsilon_2 = 1.8, D = 0.1$

(c) $\epsilon_1 = 2, \epsilon_2 = 2, D = 0.1$ **(d)** $\epsilon_1 = 0.15, \epsilon_2 = 0.5, D = 0.1$

Figure 3.7 Shader applied superellipsoids for various parameters.

Let's take one of the example parameters, the sphere. We change its size (the parameters a_1, a_2 and a_3 in the equation 3.5), and the parameter D. Table 3.2 contains the measured values for various D and parameters a_1, a_2 and a_3. We can see that the lower the value of D, the longer the generation time increases, as well as increasing values of width and height. This means that for optimal entity generation, we would have to choose a constant and highest possible D for which the generated visual would make sense. It is more complicated for width and height, as these will be affected by the entity attributes if the entity grows. From our observations, the generation time tripled if we changed only height and width with the same D.

3.6 Voxel-based superellipsoids

The idea of voxel-based generation is already present in games that pride themselves on procedural generation. Voxels are also a beneficial approach in our case.

Instead of generating a point cloud of the superellipsoid, we generate uniform

Parameters	Create time (ms)	Avg. update time (ms)
(a) $\epsilon_1 = 1, \epsilon_2 = 1, D = 0.1$	3.8017	1.9178
(b) $\epsilon_1 = 0.8, \epsilon_2 = 1.8, D = 0.1$	1.5609	1.5214
(c) $\epsilon_1 = 2, \epsilon_2 = 2, D = 0.1$	3.7189	1.8864
(d) $\epsilon_1 = 0.15, \epsilon_2 = 0.5, D = 0.1$	4.9804	2.5908

Table 3.1 Superellispoid generation measurements.

Parameters	Create time (ms)	Avg. update time (ms)
$D = 0.1, a_1 = a_2 = a_3 = 0.5$	0.9633	0.9660
$D = 0.05, a_1 = a_2 = a_3 = 0.5$	3.2408	3.0929
$D = 0.1, a_1 = a_2 = a_3 = 1$	3.8017	1.9178
$D = 0.05, a_1 = a_2 = a_3 = 1$	9.387	9.4190
$D = 0.1, a_1 = a_2 = a_3 = 2$	9.2289	8.6747
$D = 0.05, a_1 = a_2 = a_3 = 2$	35.914	54.4560

Table 3.2 Measurments of the superellipsoid's sphere for various D, height and width.

vertices in a cube. This achieves the desired uniform distribution of vertices that we could not achieve in the previous section (the used paper promised a close-to-uniform result due to nonlinear computations). We also don't have to recalculate the point cloud for different superellipsoid parameters, which allows us to have a more stable generation time.

We created a script to generate a voxel point cloud. The resulting points are sent to the shaders for processing, which moves the problem of Superellipsoid and normal calculation to the shaders.

We will use the Inside-Outside function of Superellipsoid 1.7 from Chapter 1.5, Section 1. This function represents whether a given point is inside the Superellipsoid, on the surface, or outside. Since we already have the vertex information, we can call this function and, if the vertex falls within the Superellipsoid, continue to work with it (generate an adornment for it). Regarding normals, we can use the equation 3.9 and implement it in the shader.

The resulting voxel point cloud and the rendering of the Superellipsoid within this voxel are shown in Figure 3.8. The complexity of generating points in a voxel is $O(n^3)$. On the test hardware, the generation takes about 1.8 ms. This approach gives us an advantage over Superellipsoid's point cloud generation because we are not parameter-dependent.

3.7 Chapter summary

In this chapter, we experimented with the knowledge gained from the analysis and explored ways to generate visuals. Our experiments found that to get closer to the visual target defined in the previous Game Design chapter; it was appropriate to use 2D shape interpolation and adornment generation through the geometry shader. We can combine the generation of adornment with another way to generate visuals. For now, a plain surface combined with adornments and the possible use

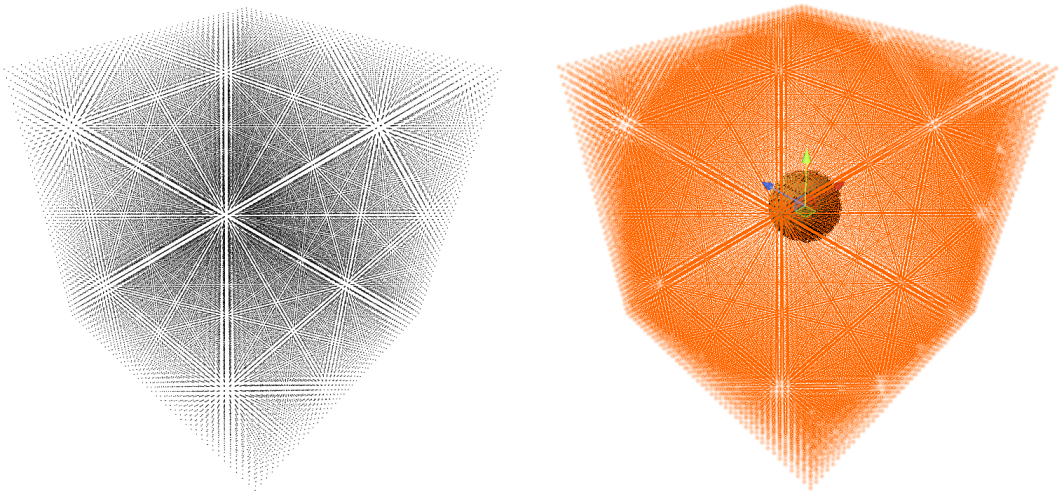


Figure 3.8 Superellipsoid generation within a voxel. The point cloud voxel is shown on the left; the superellipsoid is plotted on the right.

of noise will be enough to create the environment. We will use the superellipsoid function to generate the creatures, as it seems the most efficient of all the methods and in line with the visual requirements. At the end of our experimentation, instead of generating the point cloud on the CPU, which is a laborious process, we can create the point cloud as a voxel grid, where we process the vertices that fall into the Superellipsoid on the GPU.

4 Implementation

In this chapter, we will describe the practical part of the thesis. We will explain the setup of Unity DOTS, creating an attribute as input to shaders, and discuss in detail the required solution from the previous chapter 3 Experiments for terrain and creature generation.

The source code can be found in the thesis appendix (A.1). The source code is a part of the game under development, which has been trimmed down to only the part this thesis focuses on and the author has worked on. Thus, there may be empty systems in the code or additional comments about functionality that are beyond the scope of the thesis and are there because of dependencies between systems (e.g., order of execution of systems or having necessary components).

4.1 Unity DOTS Setup

First, we need to set up the appropriate DOTS environment. When we create a project, we need to download two packages via Package Manager: Entities and Entities Graphics. These packages will also download other dependent packages, such as Mathematics and Burst, needed for ECS to work. That's it for running Unity DOTS and using the ECS architecture. We must create a **Subscene** to create the first entity. A Subscene is specific to ECS. All objects under the Subscene are converted into entities using the Baking process as described in section 1.4.3 Entities.

4.2 Generation pipeline

To generate visuals, we need to prepare the inputs for our shaders. The figure 4.1 shows a diagram of how we can pass these inputs from components to the shader and what systems are needed to run the generation and rendering. First, we need to determine what components we need. Such elements are different for terrain, creatures, and the Lens type.

Components are added and removed according to the selected Lens. At the beginning of the run, we start the *Render Init System*, which prepares the default Lens and sets up the necessary components in the game's first frame. When the Lens is changed, the *Render Init System* is run to remove the original components from the previous Lens and add components to the newly selected Lens. After Lens change, the *Pass Attributes System* takes control over passing values from attributes to render components. We need a third system, the *Render Mesh System*, that regenerates the mesh and sets up the material when the Lens is changed. Then, the values from the components are passed to the input of the shader of the set material.

This scheme applies to both terrain and creatures. We aim to use the same logic to render anything from an ECS perspective. However, we need to distinguish creatures from terrain in these systems, and it is worth considering splitting into multiple systems so that terrain rendering is not dependent on creature rendering and vice versa.

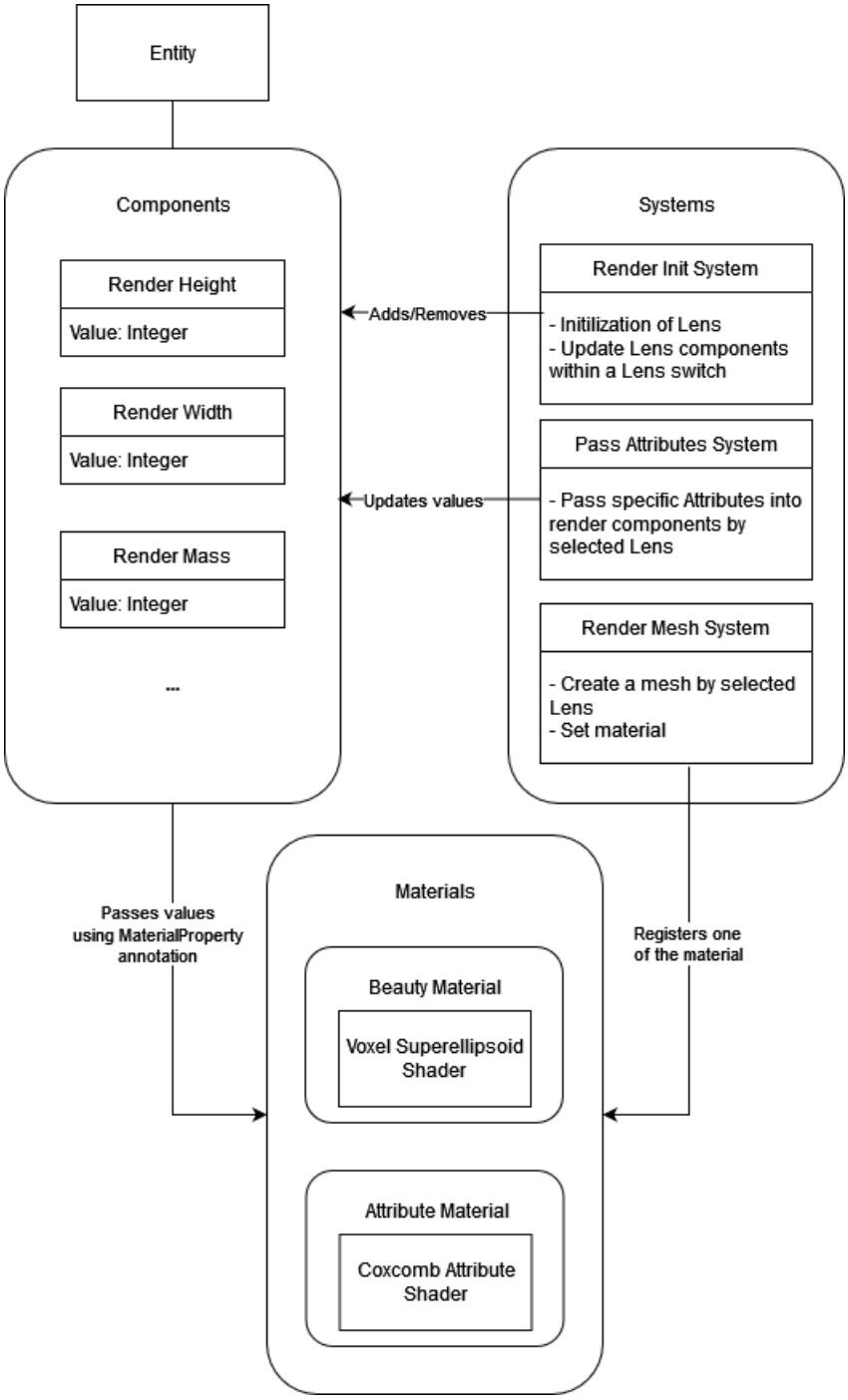


Figure 4.1 Scheme of the ECS architecture and pass data into material's shaders

Terrain	Creature
Color	**Color**
Adornment size	**Adornment size**
Adornment shape	**Adornment shape**
Roughness	**Roughness**
X coordinate	Mass
Y coordinate	Fuzziness
	Variance
	Height
	Width
	Hardness

Table 4.1 Creature and Terrain components in Beauty Lens.

4.3 Component and Render systems preparation

According to Game Design, there are two types of lenses: **Beauty Lens** and **Attribute Lens**. Specific components define these Lenses. We can render these specific components when creatures and terrain have these components. In Table 4.1, we define the components for Beauty Lens.

We use bold-coloured components for both creatures and terrain. Each has some extra components.

As for the Attribute Lens, this consists of generic components that the player will be able to define in terms of the values involved. That's why we'll generously name them *Attribute1*, *Attribute2*, etc. This thesis has a maximum of 5 attributes in the Attribute Lens. We have created it only for creatures[1].

4.3.1 Create a component

We showed an example of creating a component in 1.2, section 1.4.3 Baking of the Analysis chapter. We want to store the value of each render component. The script in which the component we defined must follow the *<component name>Authoring* convention. We recommend putting Baker components in the same script.

For the component's value to be written to the shader in the set material, the component must have the `[MaterialProperty(<Property name>)]` annotation. This annotation marks the component data as input in the material property in the shader. The name of the property we insert into this annotation must be the same as the name of the property we define in the shader.

In this way, we have created the components we need for rendering. We store all these rendering components that are used for creatures and terrain in the path `Assets/Scripts/Render/LensValuesComponents` under the convention *Render<name>Authoring*.

In addition to the rendering components, we will also create components for the Lens itself. These components reference the attributes that affect the rendering

[1]Attribute Lens will also be made for terrain in the future (for the sake of analyzing terrain), but this method is not yet designed, nor is it part of the game's minimum viable product.

and define the given Lens. The definition for Beauty Lens creatures (we call them *SpawnBeautyLens*) is in the following code snippet 4.1.

We reference each Attribute using a SerializableGuid. This Guid is generated automatically when we create the Attribute.

```
public struct SpawnBeautyLens : IComponentData
{
    public SerializableGuid AdornmentSizeAttribute;
    public SerializableGuid AdornmentShapeAttribute;
    public SerializableGuid ColorAttribute;
    ...
```

Listing 4.1 Spawn Beauty Lens component.

4.3.2 Create an attribute

To put a component on a given entity, we create an *Attribute* for the component. All the Attributes are stored in `Assets/Data/Attributes`. Attributes are game assets that we load when the game starts, and their values are affected during the game. Their Attribute values are linked to the rendering components, affecting the rendering.

An Attribute is a *ScriptableObject*[2] that contains the values of a component. In the current state of the work, these are values such as *Max* (the maximum value that the component can reach), *Min* (the minimum value), *Absolute Value* (the absolute value of the component with adaptations and buffs applied), *Relative Value* (the relative value of the component), and so on. For our work, we only need the absolute and relative values as the input to the shader.

4.3.3 Add components to Lens

Both attributes and Lenses are assets. We add a reference to attribute assets within the asset Lens through the UI. The attribute assets defined in the Lens asset must match the attributes defined in the Lens component. The figure 4.2 shows such an asset definition with attributes and its generated Guid. We can also fill in the Summary (description of the Lens) and Creator[3] (Guid of the Lens creator). Lenses created by game authors have zero Guid.

We use the Aspect functionality described in Analysis 1.4.3 to link the values from the attributes to the render components. This Aspect bundles all the render components and has methods to add or remove them and update the component values from the attributes.

This pipeline is the same for Attribute Lens creation.

4.3.4 Create Render Systems

For rendering, we have systems that meet the following three requirements:

- When a player starts the game, set the default Lens and, with it, the necessary components. We set components for all terrain and creature

[2]ScriptableObject is a data container. [29]

[3]In the future, players can create their Lenses to pass to other players in the Asset store.

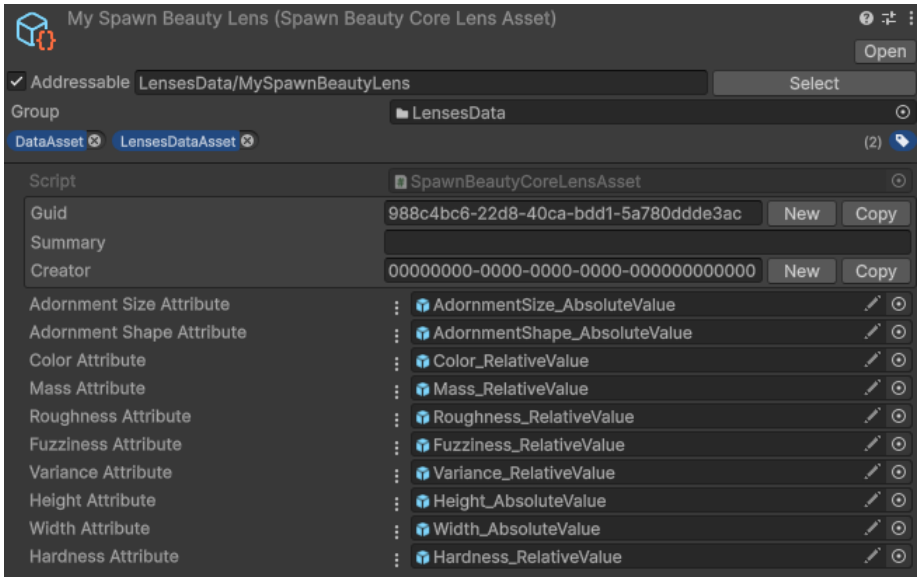


Figure 4.2 Spawn Beauty Lens Asset definition in Unity Editor.

entities. During the gameplay, if player changes the Lens, remove the components of the current Lens and assign the components needed for the next Lens. [4]

- Pass data from attribute components (4.3.2) to render components (4.3.1). This way, the render components will have the current values and render the objects correctly. [5]

- Generate a mesh according to the current Lens and register its correct material. If the current Lens is a Beauty Lens, generate a point cloud voxel. If the current Lens is an Attribute Lens, create a point cloud for the 2D plane (4 vertices). [6]

4.4 Terrain

This section details the creation of the game's terrain. We specify the terrain's characteristics and outline the processes for generating and integrating the terrain into the game. A simple terrain is sufficient for current purposes, as the work has become more focused on creature generation. In this thesis, we only consider Beauty lens for the terrain.

4.4.1 Terrain Characteristics

The overall terrain in the game is characterized by a 2D grid, where each square is an entity, we call **tile**. We can freely add and remove components to each square. This is desirable because the components define the square characteristics: whether the field is grassy, whether an object surrounds it, whether it is accessible, etc.

[4]Code: `Assets/Scripts/Render/RenderInitSystem.cs`
[5]Code: `Assets/Scripts/Render/PassAttributeDataToRenderComponentsSystem.cs`
[6]Code: `Assets/Scripts/Render/RenderMeshSystem.cs`

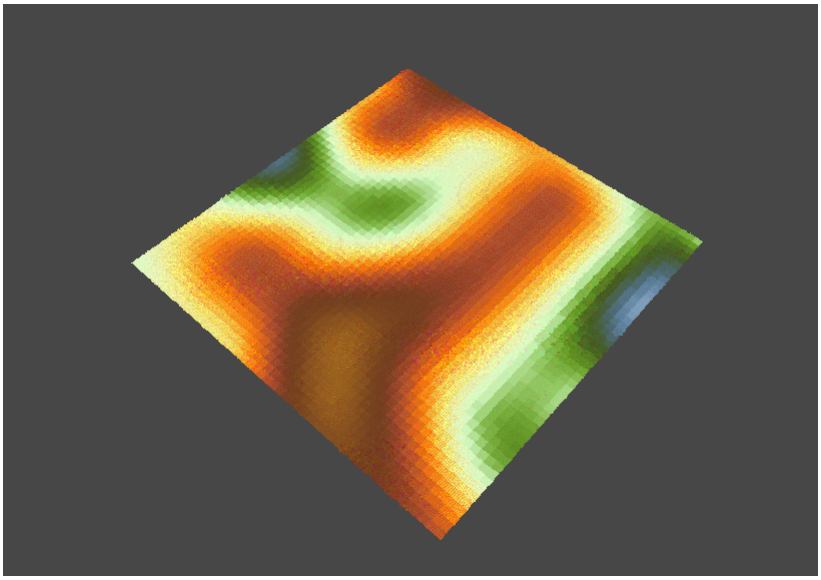


Figure 4.3 Terrain with generated colour using Perlin Noise

4.4.2 Terrain Creation Pipeline

We created our own tool, "GridCreator", for terrain generation. It gets tile prefab, colour data, feature shape, and roughness on its input. The scene needs the `GridPrototype` prefab to generate it in the scene. In GridCreationUtils[7] script, we define methods for generating individual tilings by specified width and height and processes of applying textures to material and initial components from them.

Generating terrain means creating a new level. At the moment, each level contains only one generated terrain. To generate a terrain and, therefore, create a new level, it is necessary to do the following:

1. Create a new scene and add `GridPrototype` prefab into the scene.
2. Create a new **level creator asset**. With this creator asset, we generate a terrain with our specific or generated input data.
3. Create a new **level asset**. We connect the scene with the terrain to the level asset.

4.4.3 Terrain Rendering

We render a 2D grid with minor adjustments for terrain and adornments. In the shader, we get data from attributes, i.e. Adornment Shape, Adornment Size, Roughness and Color as input. We add extra variables to the shader, such as displacement strength, scale, and wind. This data is mainly used in the Geometry shader, where we create either a square grid (to which we can apply displacement, see figure 4.5) or adornments (to which we can apply wind). The rendering of adornments is based on the experiments chapter (3.1.1). We create quads over a given grid, and in the Fragment shader, we apply an Adornment shape to it through clipping. Figure 4.4 shows a wireframe of several tiles.

[7]Code: `Assets/Scripts/Grid/Editor/GridCreationUtils.cs`

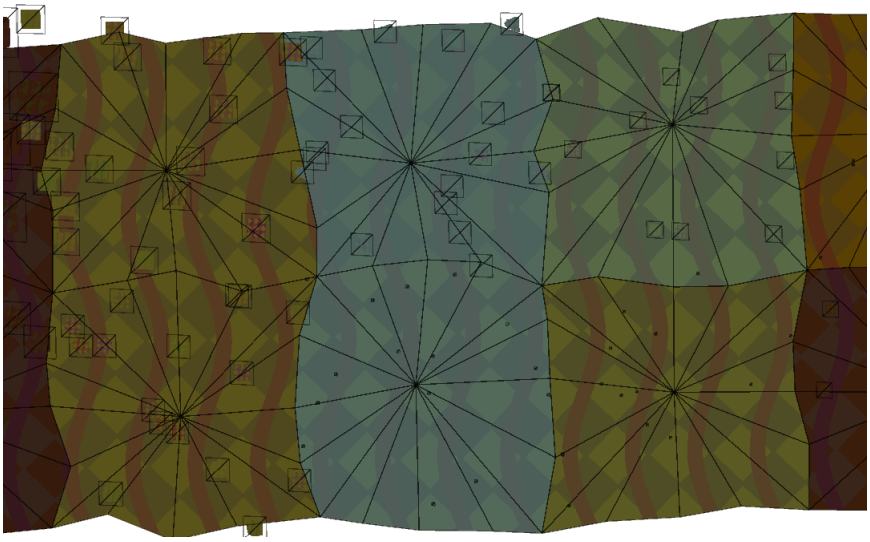


Figure 4.4 Wireframe of shaded terrain tiles with adornments.

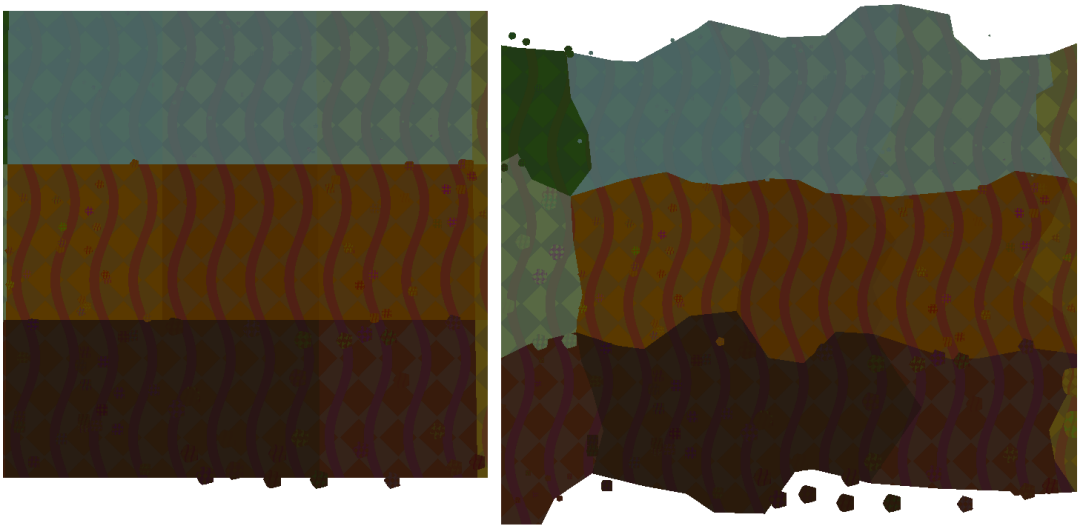


Figure 4.5 Example of displacement in tile rendering. The left picture is before displacement; right picture is with displacement (Displacement strength = 0.51, Displacement scale = 2.1)

4.5 Creatures

This section deals with the implementation of creature creation and creature rendering. Based on chapter 3 Experiments, we decided to render the mesh using a voxel point cloud and use the idea of superellipsoids in Beauty Lens rendering (3.6). This section also continues to deal with Attribute lens rendering.

4.5.1 Creating Creature mesh

The `RenderMeshSystem` creates the mesh and then updates it. A filter query is called according to the deployed components in its Update loop. If the mesh hasn't been created, it will be. Otherwise, the current mesh is updated, which is especially important for setting the height at which the creature occurs, and the correct material is applied. We set the material from the RenderMesh component, which contains the necessary materials for Lenses. The principle of creating a mesh and updating it also applies to Attribute Lens. The difference between these two lenses lies in their components. When deciding whether to use a mesh for the Beauty lens or the Attribute lens, the query checks a component that belongs exclusively to either the Beauty lens or the Attribute lens. It doesn't matter what that particular component is, but it matters to which Lens it belongs. As a result, one component can't belong to both Lenses. Another critical situation in this system is that we can't use the Burst compiler because we are access managed types. We will address this problem in the future.

4.5.2 Creature Beauty Lens Rendering

To render the creature, we need a mesh with the material correctly generated and the rendering components properly set up. The creature is rendered using Beauty Lens through the following steps[8]:

1. Based on the voxel grid size and spacing, the **vertex shader** computes new vertex positions. Voxel grid spacing depends on the size of the adornments. This size is provided as an input to the shader, which we obtain from the attribute. Position modifications such as variance and fuzziness are then applied to the vertex.

2. In the **geometry shader**, we use the inside-outside function to detect a point in the superellipsoid. Using equation 1.7 and the parameters a_1, a_2 as width of the creature and a_3 as height of the creature, we determine if the vertex is inside or on the surface of the ellipsoid. If it is, we then compute the normals using the equation 3.9 and generate the adornments for it. We generate the adornments for the creatures bilaterally to avoid holes and thus the possible passing of the light beam through the creature.

3. In the **fragment shader**, we set the adornment shape and clip everything outside the given shape. In this situation, we also consider the value of the creature's mass. If the creature's mass is less than 1, we clip certain fragments based on the Mass map. Then, we set the surface and light data. We use the available lighting model Blinn Phong from Unity.

[8]Code: `Assets/Shaders/SpawnBeautyVoxelSuperelipsoid.hlsl`

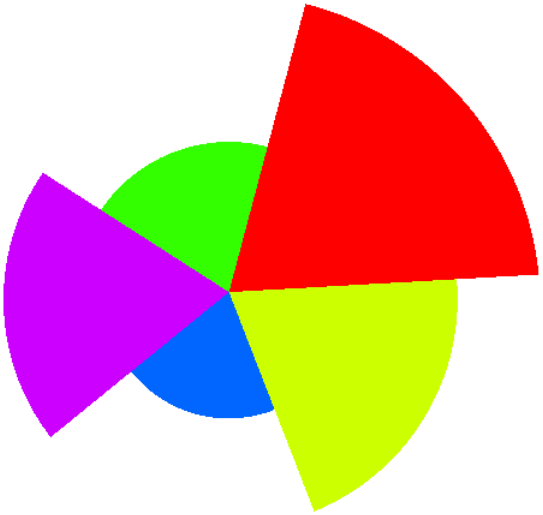


Figure 4.6 Attribute lens render.

4.5.3 Creature Attribute Lens Rendering

The creature's Attribute Lens is intended for the player to analyze the values of the creature's attributes. The player can use this view to decide if the two creatures are similar. We chose to render these values using a **Coxcomb chart**. Any five attributes the player can choose will be input to the shader[9].The chart is generated in the fragment shader by dividing the circle into segments, with each segment's size corresponding to the value of a particular attribute. Each segment is then assigned a distinct color. Then, we discard the fragments that are not coloured so that only the chart is rendered. Figure 4.6 shows an example of coxcomb chart for one entity.

4.6 Stickers

Stickers are concrete elements rendered on abstract bodies of creatures[10]. These elements are created manually and animated manually by an external animator, who can plug them into the Unity editor. Therefore, the Sticker functionality must be user-friendly, even for a non-programmer. Stickers are again stored as game assets.

To make Stickers function properly, several components are required, each serving a distinct purpose:

- **Sticker component**[11], where we store values for the position of the Sticker on the creature (meridian and parallel), the height and width of the creature and the generated Guid of the Sticker asset.
- The player can choose whether to render the Sticker. Another component, **ShouldRenderStickers**, does this and serves as a "tag" for whether to render a Sticker on the entity.

[9]Code: `Assets/Shaders/SpawnAttribute.hlsl`
[10]We want to make Stickers available for terrain in the future
[11]Code: `Assets/Scripts/Render/Sticker/StickerAuthoring.cs`

- The **Species Sticker component**[12] determines what Stickers are available for the player's Species monsters. This component implements the `IBufferElementData` interface, which allows this component to be stored in a dynamic buffer. In this component, we store the same data as the base Sticker component, except that we have a direct reference to the Sticker entity.

- **Spawn Sticker component**[13] that stores a reference to the Sticker entity. This component is deployed on specific creatures.

The **Sticker Init System**[14] handles sticker initialization. It reads the Sticker assets associated with a given creature asset and instantiates a Sticker entity for each entity with a Spawn Sticker component on it.

4.6.1 Sticker Rendering and Animation

The Sticker contains a shader[15] to render the texture. The input is a sprite sheet that includes the individual frames of the drawn element. In figure 4.7, we have hand-created a sprite sheet of the eyes we would like to glue onto the creature. Unity DOTS doesn't have its own animation module, but there are a bunch of third-party plugins to handle animations in the ECS world. We took the hybrid animation path. Since it involves switching frames in the sprite sheet, we're still feeding frame count data to our shader input and will switch between frames directly in the shader using texture offset. Code snippet 4.2 shows the texture scale calculation to display only one frame, setting the offset and then the texture transformation.

```
float offset = 1.0 / _FrameCount;
float2 uvMultiplier = float2(offset, 1);
float2 uvOffset = float2(offset * _AnimationFrame, 1);
uv = TRANSFORM_TEX((IN.uv * uvMultiplier) + uvOffset, _BaseMap);
```

Listing 4.2 UV scale and offset calculation for Sticker sprite sheet.

In the figure 4.8, we adjust the frame number and the effect gives us a simple animation. Unity's Animator component takes care of setting the frame number value. This animation can also be handled on the GPU side, but the Animator provides us with the state machine we want to use for Stickers (Sticker eyes can have an animation for opening and closing eyes as a "blink" or an eye animation for sleeping). The Animation Controller can be set to influence the `Animation Frame` parameter through the material as we show in Unity editor in figure 4.9, making it easy to animate the Sticker.

4.6.2 Spawning an entity

The `RunFactory` script[16] takes care of the initial data loading. When the game starts, it loads the level asset and sets up the `UninitializedTag` and

[12]Code: `Assets/Scripts/Render/Sticker/SpeciesStickerAuthoring.cs`
[13]Code: `Assets/Scripts/Render/Sticker/SpawnStickerAuthoring.cs`
[14]Code: `Assets/Scripts/Render/Sticker/StickerInitSystem.cs`
[15]Code: `Assets/Shaders/SpawnBeautySticker.hlsl`
[16]Code: `Assets/Scripts/Run/RunFactory.cs`

Figure 4.7 Sticker eyes sprite sheet.

Figure 4.8 Creature with animated Sticker eyes. The `Animation Frame` parameter starts at 0 (left image) and increments by one until it reaches the last frame (right image).

`NeedsSpeciesInit` components, which trigger specific systems to handle the data loading. Creatures are created in the game in the `Spawning System`[17], then spawned at a random location in the level (via `Movement System`[18][19]).

4.6.3 Lens switch

The player can change the Lens while playing. Currently, we can shift Lenses using the command line. We use the Quantum Console[20] plugin for the command line, which adds in-game command console functionality using `Command` annotation to the methods to be executed. We have a function to change the Lens in the `Lens Utils` script[21]. Calling the function to change the Lens only makes minimal changes: it adds the `NextLens` component to the queried Lens and the `PreviousLens` component to the original Lens. These components don't carry extra data; they act as tags. The `Render Init System` removes components from the original Lens with the `PrevousLens` tag and adds components from the Lens with the `NextLens` tag. Then the Render Mesh System detects the current Lens and sets the mesh and material accordingly, as described in 4.5.1.

[17] Code: `Assets/Scripts/Run/RunFactory.cs`

[18] Code: `Assets/Scripts/Grid/MovementSystem.cs`

[19] Movement system is not an ideal place to spawn a creature when you already have a Spawning System. However, this is modified for the thesis, as other spawn methods are implemented in the original systems. However, this is beyond the scope of this thesis

[20] https://assetstore.unity.com/packages/tools/utilities/quantum-console-211046

[21] Code: `Assets/Scripts/Render/LensUtils.cs`

Figure 4.9 Example of setting keyframes for material property `Animation frame` in Animation Controller.

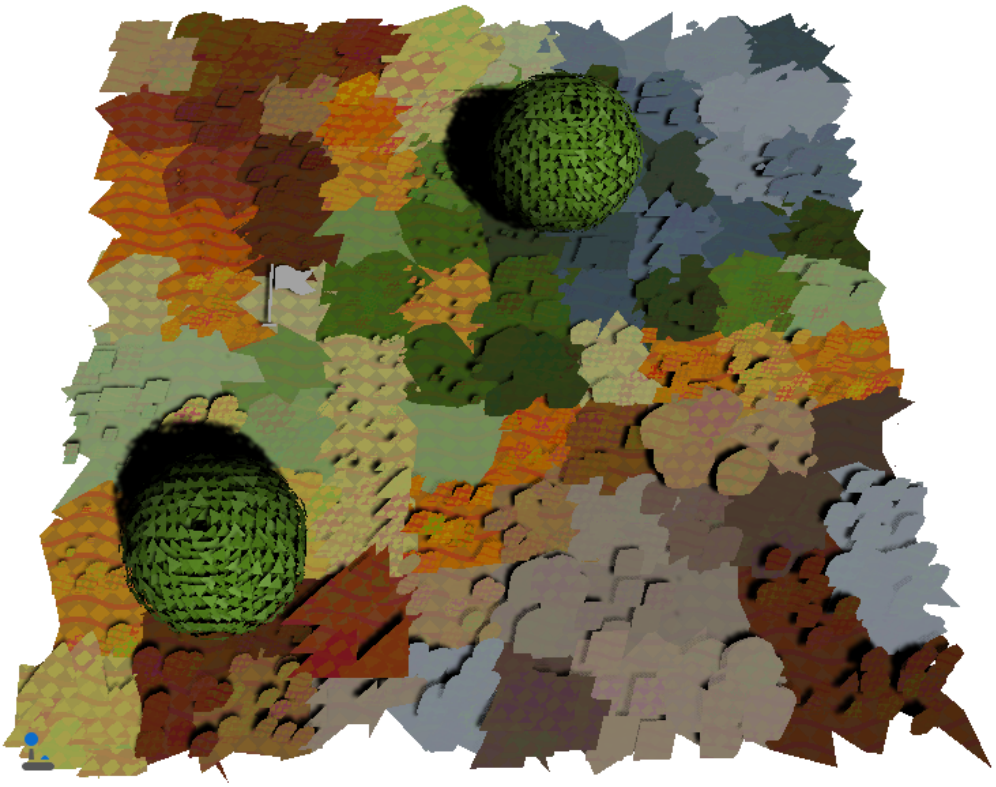


Figure 4.10 In-game Beauty Lens.

Figure 4.10 shows a sample of the visual when the game is running. You can see Beauty Lens on the terrain and the creatures. We can switch the current Creature Lens to any other using the `DebugChangeSpawnLens <entityID> <version>` command. At the moment, we only have the Attribute Lens available. We need to look in the Entity Inspector to get the entity ID and version of the Attribute Lens. We can find the entity with the `SpawnAttributeLens` component there. We can then run the command, and the Attribute Lens visuals on the creatures will appear, as shown in Figure 4.11.

4.7 Chapter Summary

In this chapter, we have discussed the implementation in Unity DOTS. We have designed a pipeline for generating visuals, described the creation of components for the required Lenses, and prepared the systems that operate on the data and run the functions for the generation and rendering functionality. We have discussed terrain and creature rendering in detail and supplied specific visual components, Stickers, that add a touch of realism to the visuals. Towards the end, we covered how entity spawning works and how to change Lens on the fly.

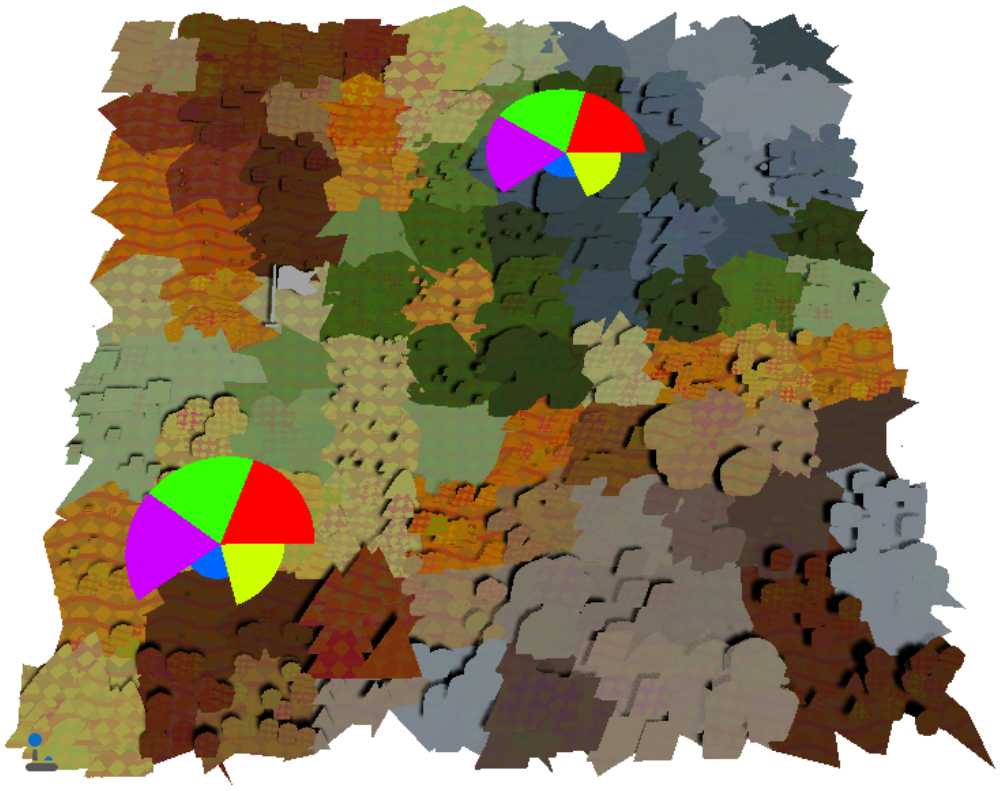


Figure 4.11 In-game Attribute Lens.

5 Visual Results

In this chapter, we will examine the results of implementing the visual rendering of the game under development. We will examine their relation to the visuals required from the game design and its described visual target. For creature rendering, we will see what results can be achieved with a given solution under different parameters, i.e., what variability the solution has.

5.1 Meeting visual requirements

We will look at the visual requirements described in the Game Design chapter and verify that the implemented solution meets these requirements.

VR01: The desired appearance must be continuous. The range of values in the parameters is continuous. There are no jumps between values, and it is not reflected in the game's visuals. Slight changes to parameters do not disrupt creature or terrain changes either. This requirement is met with the chosen solution.

VR02: The visual appearance must show the state of the individual entities. Attributes define the states of game entities. Terrain and creatures have specific attributes written into the rendering components, which affect the rendering. Based on attributes, the state of an entity can, therefore, be discerned. For example, the height or width of a creature will be reflected in its appearance. This means that the requirement is fulfilled.

VR03: Abstract visual elements should be complemented by concrete visual elements. By abstract visual elements, we refer to shape entities. We have added concrete visual elements, the Stickers. They are currently only available for creatures, and assigning Stickers to a generated shape adds a more realistic look to the creature. This way, we have fulfilled the requirement.

VR04: The generation of the appearance must be robust. Modifying the parameters for generating visuals produces results that are coherent and free of glitches. The parameter values are limited because they do not form an infinite set. Thus, we have satisfied the requirement.

VR05: Appearance generation must be efficient and real-time. Changes to attribute values are immediately reflected in the game. This is thanks to the GPU generation and the use of ECS architecture. Burst compilation on Lenses would provide further performance improvements, but it is unnecessary and is more of a premature optimization. The requirement is met.

VR06: The appearance generation process should be extensible. Thanks to the ECS architecture, we can add or remove attributes or rendering components that affect the visuals. In doing so, they have no dependencies on other components, so it doesn't break the visual or the rendering. The requirement is fulfilled.

VR07: The shape of creatures must be 3D. Thanks to superellipsoids we are able to generate 3D shapes. The requirement is fulfilled.

All requirements have been met, and so the solution is suitable for the basis of the defined game design.

5.2 Terrain

Regarding terrain generation, the variability is only in surface colour and adornment settings. However, this thesis aimed not to generate more sophisticated terrain that would provide more options for generating varied terrain, so this is sufficient for us. In Figure 5.1, we show possible variations of 10 x 10 terrains that can be created by changing the parameters of texture color, adornment shape, adornment size, and displacement.

5.3 Creatures

We generated several types of creatures with different parameters. In the provided solution, there may be very similar looks, but we also added some variance parametes to add to the variability of the look. The Figure 5.2 shows several generated creatures with different parameters.

(a) Terrain A

(b) Terrain B

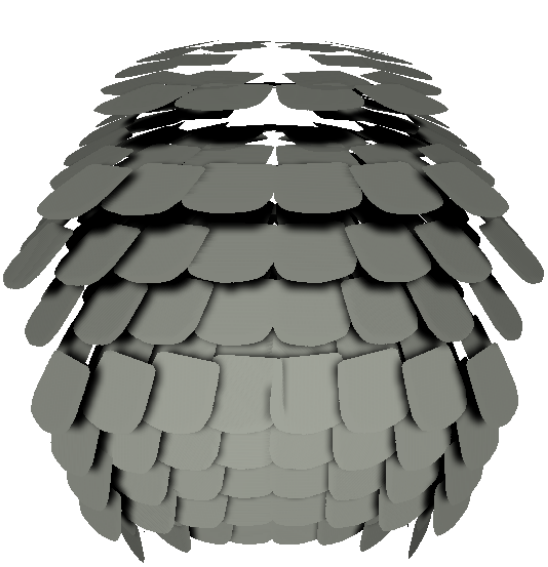
(c) Terrain C

(d) Terrain D

(e) Terrain E

(f) Terrain F

Figure 5.1 Terrain showcase with random parameters.

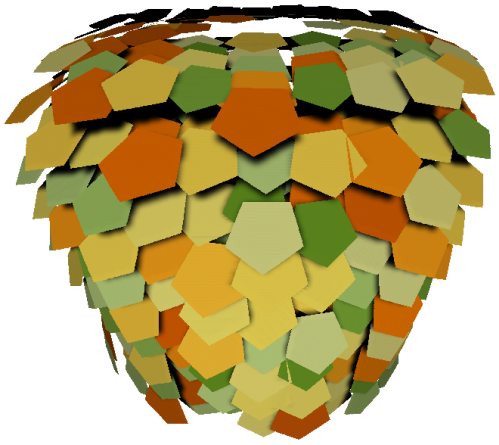


(a) Creature A

(b) Creature B

(c) Creature C

(d) Creature D



(e) Creature E

(f) Creature F

Figure 5.2 Creature showcase with random parameters.

Conclusion

In this thesis, we aimed to identify and implement a method suitable for generating visuals in a game world based on different input values in a high-performance environment using Unity DOTS and shaders. We reviewed standard procedural generation methods, and explored their application in game visuals, specifically proposing the use of superellipsoids for shape generation. We prototyped with the proposed methods and evaluated their suitability for the game under development, based on the visual requirements outlined in its design. Our findings indicated that voxels were well-suited for the overall game design, while superellipsoids were most appropriate for creating creature shapes. The results of the implemented method are positive because they meet the game design's visual requirements and can produce many assets that can help game developers in time.

However, additional work is needed to refine the visuals for the early version of the game. The current solution does not yet achieve an acceptable FPS while running the game, but it is an excellent cornerstone for further work and improvements.

Further Work

This thesis has explored several issues in procedural terrain and creature generation while implementing it and delving into the data-oriented world. Therefore, these different areas need to be explored in greater detail individually, which is beyond the timeframe of this thesis.

For terrain generation, the plan is to redesign how the level is generated. In the current situation, we generate terrain through our own editor tool and save the level as a subscene. However, this method is unsustainable, and components must be assigned to individual fields at runtime. This means that we have no idea what a given level might look like and work with before generating it for the player. As we develop the game, we plan to implement the saving of individual assets (creatures, expressions, and levels), where components will also be saved in the asset data. This will eliminate the need for component matching and, thus, possible generation errors.

As far as entity rendering is concerned, much work must be done to perfect it. As the game continues to expand, we will have more options to render creatures or terrain more accurately based on new attributes. Terrain rendering is fundamental, and over the course of future development, we will rework this way of terrain creation to make the terrain more exciting and provide an incentive to explore.

The next steps include enhancing Stickers by developing an editor that allows players to position Stickers on their creatures. Currently, there is no functionality to modify the position of Stickers.

Game development is a very complex business, specially when we are both developers and designers of the game. In the process, we may develop new thoughts and ideas that we can incorporate into the visuals to make them more special or customizable. However, the current work serves to prototype these ideas and help us to complete the development.

Key findings

Creating data-oriented games and generating creature visuals is rare in practice. Immersing yourself in a different programming approach can increase development time. During the development process, we observed a few key insights that may help future developers, especially in Unity DOTS.

1. **Have a thorough knowledge of ECS architecture, Burst compilation and Job system.** Unity DOTS has good documentation, many resources, examples of using the ECS architecture, descriptions of Burst compilation, and a Job System on the internet. It is also necessary to understand the context of memory management to achieve the potential that Unity DOTS provides. This also relates to data design efficiency to avoid unnecessary cache misses.

2. In our opinion, **developing in Unity DOTS is much easier and more beneficial when done synchronously and without Jobs first.** The Job system is sound as an optimization solution, but it is also hard for a beginner to understand. Therefore, we see the initial development without Jobs as practical for developers to think about how the Job system can help them.

3. **Unity DOTS provides tools for debugging, profiling, and inspecting additional entity and system windows.** Use the **Unity profiler**[1], which shows information about specific Entity modules for profiling. **Entities Journaling**[2] is the core for debugging ECS. It is helpful to enable the **Entities Hierarchy**[3] and **Systems**[4] windows for scene overview. In Entities Hierarchy, you can find out everything you need to know about the deployed components on an entity (Components tab), the aspects (Components tab) and which systems the components on the entity are used in (Relationships tab). The System window lists all systems, lists the queries called (Queries tab) and which entities match a particular query (Relationships tab). There are also **Archetypes**[5] and **Components**[6] windows, which we rarely use during development and debugging.

4. **Frame Debugger**[7] **is very useful for debugging visuals.** This provides information about the rendering of the frame being captured, such as what textures are used, what shader is used, the number of vertexes, etc. Unity

[1] `https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/profiler-modules-entities.html`

[2] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/entities-journaling.html`

[3] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/editor-hierarchy-window.html`

[4] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/editor-systems-window.html`

[5] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/editor-archetypes-window.html`

[6] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/editor-components-window.html`

[7] `https://docs.unity3d.com/Manual/FrameDebugger.html`

has also introduced shader debugging through Visual Studio[8], which can be helpful.

5. **Follow the latest news in Unity DOTS development.** Unity is constantly evolving its technology, and even though Unity DOTS advanced to version 1.0 in September 2022, Unity developers still need to add a large amount of functionality, such as the character controller and animation module or fix known bugs with lighting. We recommend following the **Unity DOTS Roadmap**[9] and **Known Issues page**[10] in their documentation.

6. We highly recommend **posting on the Unity forum**[11] **for questions, confusion, issues with Unity DOTS, and anything related to game development in Unity tools.** The Unity developer community is large and supportive of resolving development issues. Posting on the Unity forum helps other developers in a similar situation. There are also discussion threads regarding using Unity DOTS in production for promotional and motivational purposes.

The future of game development is bright and full of potential. This thesis's key findings, tips, and best practices provide a solid foundation for creating high-performance, visually stunning applications and open up a world of creative possibilities. Embracing the principles of Unity DOTS, procedural generation, and mastering the art of shader programming empowers developers to push the boundaries of what is possible in interactive experiences. By harnessing the full power of modern hardware and optimizing every aspect of our code, we can deliver immersive, responsive, and visually captivating games and applications that stand out in a competitive market. As we progress, it is crucial to stay curious, keep experimenting, and never stop learning. The world of game development is constantly evolving, and our ability to adapt and grow will ensure that we remain at the forefront of this dynamic field.

[8] `https://docs.unity3d.com/Manual/SL-DebuggingD3D11ShadersWithVS.html`

[9] `https://unity.com/roadmap/unity-platform/dots`

[10] `https://docs.unity3d.com/Packages/com.unity.entities.graphics@1.3/manual/known-issues.html`

[11] `https://forum.unity.com/forums/entity-component-system.147/`

Bibliography

1. Shaker, Noor; Togelius, Julian; Nelson, Mark J. Procedural content generation in games. 2016.
2. Hendrikx, Mark; Meijer, Sebastiaan; Van Der Velden, Joeri; Iosup, Alexandru. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*. 2013, vol. 9, no. 1, pp. 1–22.
3. Wikiversity. *Digital Media Concepts/No Man's Sky — Wikiversity*. 2019. Available also from: `https://en.wikiversity.org/w/index.php?title=Digital_Media_Concepts/No_Man%27s_Sky&oldid=2011255`. [Online; accessed 6-May-2019].
4. Fedoseev, K; Askarbekuly, N; Uzbekova, A E; Mazzara, M. Application of Data-Oriented Design in Game Development. *Journal of Physics: Conference Series*. 2020, vol. 1694, no. 1, p. 012035. Available from doi: `10.1088/1742-6596/1694/1/012035`.
5. Wright, Will. *Spore, birth of a game*. 2007. Available also from: `https://www.ted.com/talks/will_wright_spore_birth_of_a_game`.
6. Hecker, Chris. *My Liner Notes for Spore*. 2022. Available also from: `http://www.chrishecker.com/My_Liner_Notes_for_Spore`.
7. Inc., Electronic Arts. *Spore creature creator*. 2009. Available also from: `http://www.spore.com/creatureCreator`.
8. Choy, Lydia; Ingram, Ryan; Quigley, Ocean; Sharp, Brian; Willmott, Andrew. Rigblocks: player-deformable objects. In: *ACM SIGGRAPH 2007 sketches*. 2007, 83–es.
9. contributors, Wikipedia. *Voxel*. 2024. Available also from: `https://en.wikipedia.org/wiki/Voxel`.
10. McKendrick, Innes. *Continuous World Generation in no man's sky*. YouTube, 2017. Available also from: `https://www.youtube.com/watch?v=sCRzxEEcO2Y`.
11. Murray, Sean. *Building Worlds in No Man's Sky Using Math(s)*. YouTube, 2017. Available also from: `https://www.youtube.com/watch?v=C9RyEiEzMiU`
12. Kratsch, Benjamin. *An interview with Sean Murray of "no man's sky"*. 2016. Available also from: `https://www.inverse.com/gaming/19373-sean-murray-interview-no-man-s-sky-space-whales-elon-musk`.
13. Ford, Timothy. *Overwatch Gameplay Architecture and Netcode*. YouTube, 2017. Available also from: `https://www.youtube.com/watch?v=W3aieHjyNvw`
14. Eizenhorn. *Showcase - Unity Dots Case Study in production*. 2024. Available also from: `https://forum.unity.com/threads/unity-dots-case-study-in-production.561229/`.
15. 407, Door. *Diplomacy is not an option on steam*. 2022. Available also from: `https://store.steampowered.com/app/1272320/Diplomacy_is_Not_an_Option`.

16. Amato, Alba. Procedural content generation in the game industry. *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*. 2017, pp. 15–25.

17. Blow, Jonathan. *Truth in game design*. 2011. Available also from: https://gdcvault.com/play/1014982/Truth-in-Game.

18. Sims, Karl. *Understanding julia and Mandelbrot sets*. 2022. Available also from: https://www.karlsims.com/julia.html.

19. Wikipedia contributors. *Julia set — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=Julia_set&oldid=1218760211]. 2024. [Online; accessed 1-July-2024].

20. Prusinkiewicz, Przemyslaw; Lindenmayer, Aristid. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

21. Furmanek, Piotr. POLYHEDRAL COVERS BASED ON L-SYSTEM FRACTAL CONSTRUCTION. [N.d.].

22. Ferreira, Paulo. Sampling superquadric point clouds with normals. *arXiv preprint arXiv:1802.05176*. 2018.

23. contributors, Wikipedia. *Superellipsoid*. 2024. Available also from: https://en.wikipedia.org/wiki/Superellipsoid.

24. Raffaillac, Thibault; Huot, Stéphane. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proceedings of the ACM on Human-Computer Interaction*. 2019, vol. 3, no. EICS, pp. 1–22.

25. Technologies, Unity. *Entities overview*. 2024. Available at https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual.

26. Mei, Gang; Tian, Hong. Impact of Data Layouts on the Efficiency of GPU-accelerated IDW Interpolation. *SpringerPlus*. 2016, vol. 5, pp. 1–18. Available from doi: 10.1186/s40064-016-1731-6.

27. Bayliss, J. D. The Data-Oriented Design Process for Game Development. *Computer*. 2022, vol. 55, no. 05, pp. 31–38. issn 1558-0814. Available from doi: 10.1109/MC.2022.3155108.

28. Technologies, Unity. *About Burst*. 2024. Available at https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual.

29. Technologies, Unity. *Unity User Manual 2022.3 (LTS)*. 2024. Available at https://docs.unity3d.com/Manual/.

30. Overvoorde, Alexander. *Introduction*. 2023. Available also from: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction.

31. Lorihollasch. *Pipelines for Direct3D version 11 - Windows Drivers*. 2021. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/pipelines-for-direct3d-version-11.

32. Ribecca, Severino. *Radar Chart*. [N.d.]. Available also from: https://datavizcatalogue.com/methods/radar_chart.html.

33. Flerlage, Ken. *Tableau Coxcomb Chart template*. 2022. Available also from: https://www.flerlagetwins.com/2021/12/coxcomb.html.

34. Pilu, Maurizio; Fisher, Robert. Equal-Distance Sampling of Superellipse Models. 1995. Available from doi: 10.5244/C.9.26.

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
PCG	Procedural Content Generation
ECS	Entity Component System architecture
Unity DOTS	Unity Data-Oriented Technical Stack
GPU	Graphics Processing Unit
CPU	Central Processing Unit
OOD	Object-Oriented Design
DOD	Data-Oriented Design
GDC	Game Developers Conference
AFK	Away From Keyboard
FPS	Frames Per Second
SoA	Structures of Arrays
AoS	Arrays of Structures
JIT	Just-In-Time compilation
AoS	Ahead-Of-Time compilation
URP	Universal Render Pipeline
HDRP	High Definition Render Pipeline

A Attachments

A.1 Unity Project

The Unity project is stored in the *source* folder. It contains most of the visual generation experiments mentioned in this thesis, the implementation of the chosen graphics solution using voxels and Superellipsoids, and its integration in Unity DOTS. The project is stored in Unity version 2023.3.0b5.

A.2 User Documentation

User documentation describes how to start and navigate a Unity project. It also includes how to start the game running and lists options to modify the parameters of various generation methods. User documentation is stored in the *documentation* folder.