KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

CAMPUS GROUP T LEUVEN

Parameterized procedural planet generator

Development of a Unity editor tool

Douwe RAVERS

Supervisor(s): Stef Desmet
Co-supervisor(s): Jeroen Wauters

Master Thesis submitted to obtain the degree of Master of Science in Engineering Technology: Electronics and ICT Engineering

Academic Year 2021 - 2022

Acknowledgements

During my studies at KU Leuven, Group T, I have encountered many interesting topics, from creating CAD drawings to assembly programming on a microcontroller. During the course of software development, I discovered that with the skills of that course, I could make games in the Godot game engine. This turned into one of my favorite hobbies with participation in a game development competition *Ludum Dare* every half year.

I'm grateful that Group T allowed me to propose and execute this thesis which is a summation of both my knowledge gained as a student and the knowledge from the many prototypes I have built over the years as a hobbyist game developer.

I also want to thank both Jeroen Wauters and Stef Desmet for their guidance throughout the development of the tool and during the writing of this thesis paper.

Also, I like to thank my girlfriend, parents, and sister for both their advice and feedback on the written paper.

Finally, I like to thank the Unity forum user Lex-DRL for freely sharing an MIT licensed noise shader for Unity based on the `https://github.com/ashima/webgl-noise` GitHub repository.

Abstract

In this thesis, the development of a planetary terrain generation tool for game developers is described. The interactable tool is developed in the Unity game engine as a unity package. The thesis describes an implemented solution to multiple challenges associated with planetary terrain systems in games. To generate the terrain a procedural generation algorithm is proposed which is capable of generating terrain on different detail levels.

The tool provides three different methodologies to handle the high detail of the planetary terrain, two of which are built on top of the Unity LOD component and the Unity terrain system. The third system is a quadtree which is a custom implementation for this tool.

The procedural generation algorithm provides a GUI which enables the user to design a planet using sliders and a low-detailed preview. The algorithm makes use of the GPU to increase performance.

Contents

Acknowledgements iii

Abstract iv

List of Figures viii

List of Tables x

List of Symbols xi

1 Introduction 1

1.1 The problem statement 1

1.2 Research goal 2

1.3 Project source 2

1.4 Outline of the paper 2

2 Related work 3

2.1 Similar research 3

2.2 Tools providing a planetary terrain 4

2.2.1 Space Graphics Toolkit 4

2.2.2 Orbis 4

2.3 Commercial games using planetary terrain 5

2.3.1 No Man's sky 5

2.3.2 Kerbal space program 6

3 Requirement Analysis 7

3.1 User challenges 7

3.1.1 User interface 7

3.1.2 User feedback 7

3.1.3 Predictability 7

3.2 Technical challenges . . . 8
3.2.1 Unity Engine . . . 8
3.2.2 Data configuration . . . 8
3.2.3 Visualization . . . 10
3.2.4 Performance . . . 11
3.3 Proposed solution . . . 12
4 Conceptual design . . . 14
4.1 The overall design of the planetary terrain tool . . . 14
4.2 The design of the editor tool . . . 15
4.2.1 The in editor GUI . . . 15
4.2.2 Preview planet . . . 18
4.2.3 Data storage . . . 19
4.3 The design of the runtime planet system . . . 20
4.3.1 The quadtree system . . . 21
4.3.2 Terrain system . . . 21
4.4 The design of the parameterized procedural algorithm . . . 22
4.4.1 Procedural data . . . 23
4.4.2 Mesh and texture matching . . . 24
4.4.3 Procedural mesh . . . 25
4.4.4 Procedural material . . . 25
4.4.5 Data textures . . . 26
4.4.6 Color textures . . . 26
4.4.7 Effect textures . . . 27
4.4.8 Applied to the Unity terrain system . . . 28
5 Implementation . . . 30
5.1 Unity Engine . . . 30
5.2 Procedural Data . . . 30
5.3 Mesh generation . . . 31
5.3.1 The definition of a mesh . . . 31

5.3.2 Basic meshes 32
5.4 Material generation 33
5.4.1 Base texture generation 33
5.5 In-editor GUI 36
5.6 Gradient 2D 36
5.6.1 Color interpolation 37
5.6.2 Using the gradient in texture generation 37
5.7 Preview planet 37
5.8 Run-time planet system 38
5.8.1 LOD system 38
5.8.2 Quad tree system 39
5.9 Procedural algorithm 41
5.9.1 Adjusting the mesh 41
5.9.2 Generating Materials 43
5.9.3 Generating terrain data 46
6 Future work. 48
6.1 Future improvements 48
6.2 Future research 49
7 Conclusion 50

List of Figures

2.1 Space graphics planet from [29] 4
2.2 Orbis planet from video frame [25] 5
2.3 A screenshot from No man's sky [7] 5
2.4 Satellite above Kerbin 6

3.1 L shaped mesh 8
3.2 L shaped voxel shape 9
3.3 A heightmap texture with a wireframe mesh above it from `daac.hpc.mil` 10
3.4 Schematic overview of CPU vs GPU processing 12

4.1 An overview of the design 14
4.2 The design of the editor tool 15
4.3 GUI modes 16
4.4 preview modes 19
4.5 The design of the planet system 20
4.6 quadtree example 21
4.7 Terrain orientation visualization 22
4.9 Base texture examples 25
4.10 Effect texture examples 28
4.11 Example of regular placement of vegetation 29

5.1 The saving and loading of the procedural data 31
5.2 An overview of the different mesh arrays 32
5.3 A static function creating a quad mesh with UV coordinates 33
5.4 The layout of space used by the UVs 34
5.5 A schematic overview of the branch systems 35

5.6 Example code of how a GUI can be coded and the result of the example GUI . . . 36
5.7 The shader code for calculating the gradient color. 37
5.8 A schematic overview of the gradient sampling 38
5.9 A schematic overview of the branch systems 40
5.10 The structure of the procedural generation algorithm 41
5.11 Procedural mesh generation of the base quad for the quadtree 42
5.12 A schematic overview of the subdivision modifier 43

List of Tables

4.1 A overview of the procedural properties 23

5.1 Every cube side and the calculation from texture to mesh space 35

5.2 Level of detail mesh and texture data 39

List of Symbols

Acronyms

DOTS	Data-oriented technology stack
MMORPG	Massively multiplayer online role-playing game
LOD	Level of detail
OOP	Object-oriented programming
DOP	Data-oriented programming
GPU	Graphics processing unit
CPU	Central processing unit
HLSL	High level shader language
GUI	Graphical user interface
JSON	JavaScript object notation
API	Application programming interface
VR	virtual reality

1 INTRODUCTION

During the COVID-19 pandemic, the video game industry saw rapid growth compared to many other entertainment industries. [26] [8] Even before the pandemic the gaming industry already was the second biggest media industry globally. [4] As the industry gets bigger, game developers introduce new concepts and features to let their games stand out. One of these concepts is a space setting with entire planets to explore. Games like No Man's Sky [7], Elite Dangerous [3], Space Engineers [2], Kerbal space program [14], Star Citizen [10], and Bethesda's newly announced Starfield [6] all feature large-scale game worlds with real-size approximating planets. In this thesis, a tool is proposed that enables game developers to generate large-scale planets that can be used in their games.

1.1 The problem statement

It is not uncommon for games to have large outside environments. These large environments contain a lot of detail, this makes these environments resource-intensive to visualize. To tackle this problem, game developers introduced the concept of a terrain that dynamically rises the detail of the environment the closer it is to the player. [17]

A terrain is optimized to work for large areas with high detail. The design of the terrain can be created by manually shaping the surface into the desired topology[1]. This can be a very work-intensive job. A solution to this problem is to automatically shape the terrain surface. This process is called procedural generation. Using several parameters, an algorithm calculates a certain height and color value for every point on the surface. This way the level designer only has to design the parameters of the generation algorithm. [23]

Planets are large environments, that can contain multiple environmental situations. Because of this, a procedural algorithm for one type of environment does not suffice. Instead, a procedural algorithm that can generate multiple environmental situations and apply them to one single planetary terrain is introduced. This can be done by generating different areas onto the surface and altering the behavior of the algorithm in these areas.

In order to make a practical tool, it requires game developers to generate planets that fit their game. This means the procedural algorithm can not be a closed system. Instead, it should be a system that is adjustable by the developer. This is achieved by parameterizing the procedural algorithm and exposing these parameters to the game developer. This can be done by providing an API [2] or GUI [3].

[1] Topology: The shape of the terrain surface.

[2] API: Application programming interface see chapter 3.1.1

[3] GUI: Graphical user interface see chapter 3.1.1

1.2 Research goal

The goal of this thesis is to design a system that can generate planetary terrain which has a spherical nature, large surface area, and high detail surface. This system is generated using a parameterized procedural algorithm that offers multiple environmental situations.

1.3 Project source

The implementation of the tool is open-source and can be found at `https://github.com/DouweRavers/ThesisProject-ThePlanetEngine`. Using the git protocol the tool can be integrated into the Unity editor.

1.4 Outline of the paper

The thesis is divided into three parts. The first part is the analysis. Here the problem statement is analyzed by reviewing related work (chapter 2) and a requirement analysis (chapter 3) is performed which defines the problem into separate more specified challenges. The second part is the description of the tool. Here the design (chapter 4) and implementation (chapter 5) of the tool are described. The last part is the reflection. Here possible future work is discussed (chapter6) and a conclusion is drawn (chapter 7).

2 RELATED WORK

In this chapter, a study of related research, similar tools, and games using similar systems are performed.

2.1 Similar research

To orientate the current research into the topic, papers covering similar research are examined.

A paper by Ricardo B. D. d'Oliveira and Antonio L. Apolinario Jr. [5] describes a system that combines procedural generation with multi-resolution planetary terrain[1]. The paper proposes a pipeline that makes use of GPU capabilities to accelerate procedural content generation. The multi-resolution planetary terrain is achieved by implementing a quadtree[2] system. The procedural generation makes use of noise[3] generation to create low surface content. A similar goal is defined for this thesis. However, the paper does not cover an exposed parameterized algorithm.

A paper by Vitacion Ryan and Liu Li [22] performs a performance test on noise generation algorithms. The noise algorithms are chosen in the context of planetary terrain generation. The test is performed by creating a parameterized noise generation application that tracks the running speeds of the algorithms and the memory usage. The application is built using Unity[4] and the algorithms are implemented on the CPU[5]. After the performance tests on the noise algorithms, a VR[6] application is implemented which makes use of a planetary terrain plugin for Unity which is fed by the noise algorithms to create planetary terrain in a VR application. One of the conclusions of this paper is the requirement for a method of better parameterizing the noise algorithms. This in part is the goal of this thesis.

[1] Multi-resolution planetary terrain means a planet system which has multiple levels of detail.

[2] A quadtree system is a hierarchy of detail levels that dynamically alter the detail level. This is discussed in more detail in chapter 4.3.1

[3] Noise is a description of random values. This can be purely random or a randomized pattern.

[4] Unity is a game engine. This is discussed more in later chapters.

[5] CPU: Central processing unit

[6] VR: Virtual reality

2.2 Tools providing a planetary terrain

There already exists software that provides planetary terrain. Two examples are discussed.

2.2.1 Space Graphics Toolkit

This toolkit is aimed to be a full-featured solution for space games. It contains a large collection of components and visuals covering different aspects of space games. These visuals are elements like star flares, asteroid belts, auroras, and gravitational lensing. Another important feature of this toolkit is the planetary terrain system with atmospheric support as the one shown in figure 2.1. The toolkit also offers a floating origin system[7] that enables games to be truly infinite. [29]

The planetary terrain uses procedural generation to create a surface shape of the planet and populate it with additional shapes. The procedural algorithm can work complementary with a height map[8] which allows the planet to have a predetermined overall shape. Here the procedural algorithm is used to add additional detail once the height map resolution is too low. The tool enables the procedural generation algorithm to be adjusted, depending on different areas. These named areas can also have a different ground texture and can be populated by certain shapes. In order to render such large terrains, the tool uses a dynamic level of detail (LOD)[9] for rendering[10] the surface at a variety of distances.[28]

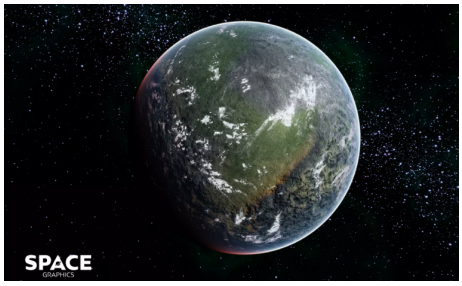


Figure 2.1: Space graphics planet from [29]

2.2.2 Orbis

Currently, the unity engine is going through a major technical makeover. As the Unity engine is object-oriented, integrating multi-threading into game code is very difficult. As a result, Unity has decided to gradually switch from an object-oriented paradigm to a data-oriented paradigm. This new system is called the data-oriented technology stack (DOTS). It is already available but still incomplete and in development. Orbis offers a terrain system using this new DOTS system. [25] An example of this planetary terrain is seen in figure 2.2.

[7] Floating point origin is a system that moves the world instead of the player. This prevents float precision issues but adds complexity and requires additional resources.

[8] Height map is an image which contains data to increase or decrease the surface height. This is discussed in section 3.2.2.

[9] LOD: Level of detail is a way of defining the same shape in different detail levels. This is discussed in section 3.2.3.

[10] Rendering is the act of drawing the three-dimensional geometry to the screen.

The terrain system uses a blend of procedural generation and height maps. The tool also offers procedural foliage[11] placement which adds grass and trees to the surface. The game developer can also preview the terrain in low resolution. An additional feature of this tool is the ability to edit the terrain at runtime if the game developer allows this. Meaning that a player can modify the terrain height and color after the generation process. The terrain will keep track of these changes and update the terrain accordingly. The tool also provides a floating origin system to handle large-scale terrain.[24]

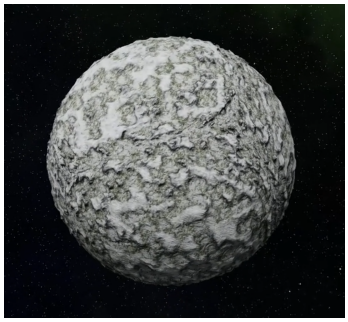


Figure 2.2: Orbis planet from video frame [25]

2.3 Commercial games using planetary terrain

There are already several games using planetary terrain systems. These games all developed their own dedicated terrain system. Thus the developers did not use a tool to generate the terrain for them. Meaning a lot of development time and resources for these games went to creating these systems.

2.3.1 No Man's sky

No Man's Sky is an action-adventure survival game created by Hello Games. The core concept of the game is to procedurally generate an infinite universe and allow the player to explore this universe alone or in multiplayer. On top of generating an infinite universe, the game also populates this universe with procedurally generated content like creatures, plants, and spaceships. This content is visible in figure 2.3. [7]

Figure 2.3: A screenshot from No man's sky [7]

[11] Foliage represents small details onto the terrain surface. For example grass or small stones.

2.3.2 Kerbal space program

Kerbal space program is a space program management and space flight simulation game from the game development studio Squad. The goal of the game is to navigate space using realistic space travel physics. [14] The planets in the game are inspired by our solar system but are 6.5 to 10 times smaller. [15]

This game is developed using the Unity game engine and makes use of a quadtree system for handling large-scale planets. This is interesting as the tool developed for this thesis will also be developed for the Unity engine and the terrain system is also based on a quadtree system. This means that the developers of this game would have benefitted from a tool as proposed in this thesis. [1]

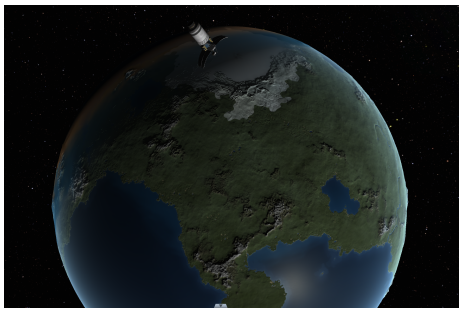


Figure 2.4: Satellite above Kerbin

3 REQUIREMENT ANALYSIS

In this chapter, the challenges regarding this thesis are explored. First, in regard to the usage of the tool by a potential user, then by a technical analysis regarding how the system can be implemented.

3.1 User challenges

3.1.1 User interface

In order to interact with the tool, some method of interface is required. This can be done in two ways. The first way is to expose the planet generation code through an API. This way game developers can generate planets by passing parameters through code and calling the generation functions. A second way is to develop a graphical user interface (GUI). This way a game developer can visually adjust the properties of the planet without having to write code. For this tool, the GUI methodology is chosen. This allows users with no programming background to use the tool as well.

3.1.2 User feedback

Closely linked with the user interface is user feedback. If an interface just exposes the properties without some kind of feedback, the user has no idea what kind of planet is generated. Running the game and seeing what kind of planet gets generated is one method of feedback, but this requires the user to restart the game every time a property has changed. This method also hides the impact of a single property as the user will only see the result of the full planet generation. This makes it hard to decide which properties to adjust and what the impact of the adjustment is. To increase usability an instant feedback system is introduced into the tool. By creating a simplified version of the final planet which regenerates instantaneously after altering a property, a more intuitive design experience results. The preview will highlight the property that is being changed, so the user is aware of the impact of the change.

3.1.3 Predictability

Since only a minimal amount of data is required to procedurally generate a terrain, even the slightest change to that data can result in a totally different terrain shape. If only one parameter unintentionally changes over time, the designing process would be impossible as the planet would change shape as the parameter changes over time. Therefore it is important to clearly define every input parameter of the algorithm and keep track of this data. In the proposed tool every parameter fed to the algorithm is stored in a save file. This way the planet is saved persistently[1].

[1] Persistent data is a type of data that is stored so that even after an application has ended, the data is still accessible.

3.2 Technical challenges

3.2.1 Unity Engine

For a development environment, the Unity engine is chosen. This decision is made because Unity is the most popular game engine at the moment and it is used in other courses given at KU Leuven, Group T. The tool will be implemented as a Unity package[2] which is hosted on GitHub[3]. This allows all Unity developers to download the package by feeding the git URL to the Unity editor. In chapter 5.1 a technical overview of the engine is provided.

3.2.2 Data configuration

In order to define a terrain shape, different methods of data configuration are possible.

Mesh data

The most common method of storing a geometrical digital shape is to store it as a mesh. A mesh is a description of a three-dimensional shape in computer graphics. A more detailed explanation of a mesh is described in chapter 5.3.1. Figure 3.1 shows the structure of an L-shaped mesh.

A downside of using a mesh to store terrain data is that a mesh only has one level of detail. This means that to fully describe the terrain, every detail should be incorporated into the shape. This does not mean it is always drawn in full detail, but in order to visualize the terrain first all information must be loaded in and then processed to reduce the detail if required.

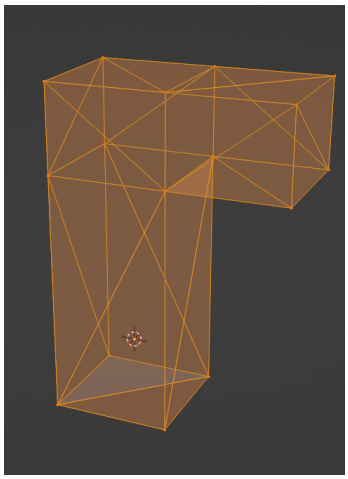


Figure 3.1: L shaped mesh

[2] A unity package is an extension of the Unity editor.
[3] GitHub is an online git repository service that allows software developers to share code.

Voxel data

A second way of describing three-dimensional shapes is by voxels. Describing a shape in voxels means describing the shape in a three-dimensional discrete grid. In contrast to meshes, voxels don't describe the surface of a three-dimensional shape, but instead its volume. This can be seen in figure 3.2 where the previous L shape is recreated but now in voxels. A similarity with 2D graphics can be made. Here a vector drawing[4] is like a mesh, it describes the contour of an object. While a Voxel is like a pixel of a 2D image, it describes the value of a given discrete point in space. This means that for most models the amount of data is actually higher for voxels than for meshes. That's because when a shape increases in size, the volume will increase at a power of three while the surface will increase at a power of two. The advantage of voxels is that they enable the storage of an entire terrain volume instead of just the surface. In terms of planetary terrain, this would mean a description of the entire contents of the planet instead of just what is visible on the surface. The game Minecraft[27] is a good example of a voxel-based terrain. A disadvantage of voxel data is that it requires to be converted to a mesh in order to be visualized. [12]

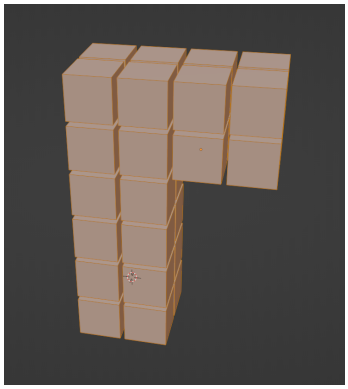


Figure 3.2: L shaped voxel shape

Height maps

A method of reducing the size of the terrain data is to use heightmaps. Using a heightmap requires the assumption that the terrain follows a predefined overall shape, this can be a flat plane, a sphere, or something else. The height map is an image that wraps around the mesh and every pixel in the image represents the amount of height that should be added to that point. This results in the general shape from the assumption, but with small height modifications all over the surface. The figure 3.3 shows how the image at the bottom alters a planar surface which is visible by the mesh projected on top. [13]

[4] Vector drawing is a digital image type that defines shapes as polynomials.

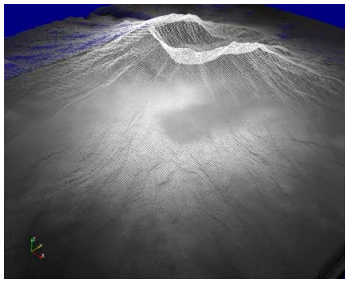


Figure 3.3: A heightmap texture with a wireframe mesh above it from daac.hpc.mil

Procedural properties

The last method of reducing the size of the terrain data is by the use of procedural generation. Instead of storing the shape of the terrain, procedural generation stores some simple properties and creates a description of the surface based on mathematical calculations using the provided properties. This methodology is used by the tool to store terrain shapes.

3.2.3 Visualization

Visualizing the terrain requires a type of mesh that defines the shape of the terrain. A larger mesh requires more resources to be drawn to the screen. As a result, methods of reducing the size of the mesh are introduced.

Mesh data

A game engine contains a rendering engine which is a system that converts three-dimensional meshes to a two-dimensional image that can be projected to the screen. The projection is done by a series of linear transformations which project all mesh data points to a planar surface in three-dimensional space that represents the screen. This receiving projection plane is called the camera.

As a mesh increases in size, the contained data becomes larger. More data also means more transformation calculations as the transformations are applied to every data entry. Because of this, an increase in mesh data results in an increase in rendering time, which lowers the framerate. This is something that is undesired in game development and should be avoided. Thus visualizing a planet using a mesh is only an option if the planet has a low detail level.

[11]

Level of detail

A game world is comprised of multiple objects. For example a house, a car, and a machine gun. Every object is visualized using a mesh. As all these objects are visualized by the render engine, it is not the size of the largest mesh that defines the framerate, but the sum of all the mesh data that defines the framerate. So if too many objects are present in the game, the framerate will drop to undesired levels.

A solution to this problem is to reduce the mesh size of meshes that are far away or that are very small. As those objects are only a small part of the final image, no detail seems to be lost visually. Once the player moves, the meshes should be updated so all close-by objects are high detail and all far-away objects are low detail. This system is called the level of detail. The tool uses this methodology to visualize planets that are far away from the camera. [9]

Dynamic level of detail

Due to the size of planetary terrain, the amount of detail contained by one object can already reduce the framerate to undesired levels. For this situation, a dynamic level of detail system is introduced. By adjusting the mesh of the object to only contain detail close to the player, the mesh data is reduced. Applied to a planetary terrain, this would result in highly detailed terrain close to the player, while in the distance only the rough shape of the terrain is visible. The tool uses a quadtree as mentioned in section [**?**] which is a type of dynamic level of detail. [5]

3.2.4 Performance

A major challenge regarding procedural generation is the performance of the generation algorithm. Creating terrain requires generating meshes and textures constantly. In order to generate this, a lot of calculations are performed on a large set of data.

Design for lower speeds

Instead of methods for increasing the speed of the algorithm, designing the system to cope with low-speed algorithms is an option. When a slow algorithm runs, the entire frame[5] is halted. This results in temporary or constant low framerate. A method to avoid this problem is by the use of a second thread. If the algorithm is generating the content using a separate thread, the game can keep going while the second thread is computing. Once the thread is done, the content can replace the previous content and no framerate drop is introduced to the game flow. [11]

As mentioned during the Orbis-related work case, the object-oriented nature of the Unity engine does prevent the efficient use of threads. The engine however provides a different system for handling slower algorithms, called coroutines. These systems allow the execution of a method to be halted mid-processing and resume execution at a later time. By designing the procedural algorithm as a coroutine, the algorithm can be paused once the frame duration is getting too long and resumed the next frame. [19]

Unity DOTS

The new Unity DOTS system enables the development of highly multi-threaded algorithms. By separating the data from the execution, the DOTS system can run multiple threads on different cores, which greatly improves the execution time of the algorithm. Using the DOTS system means shifting

[5] A frame is a single image generated by the renderer. Games mostly use 60 frames per second.

from object-oriented programming (OOP) to data-oriented programming (DOP) which, depending on the developer can be an advantage or disadvantage. [16]

Shaders

The main reason the procedural generation is slow is that it has to calculate the same transformation on a lot of data. For this problem, most devices contain an additional hardware component called the graphics processing unit (GPU). This component works similarly to the Central processing unit (CPU), as it also processes data using compiled code. The difference between the CPU and GPU is the high parallel design of the GPU. This device is not optimized to process a complex algorithm with complex pathways as the CPU is, but instead to run a simple algorithm on different data entries simultaneously. Figure 3.4 gives a schematic overview of how data is sent in, processed, and send out by the CPU, while with the GPU many data entries are sent in simultaneously, processed, and send out.

The program that runs on a GPU is called a shader. By implementing the procedural generation algorithm in shader code. The main bottleneck of the procedural generation process is solved as a lot of data can now be processed at the same time. Unity provides shader programming using the High-level shader language HLSL. This shader will be used by the tool to perform most of the procedural calculations. [18]

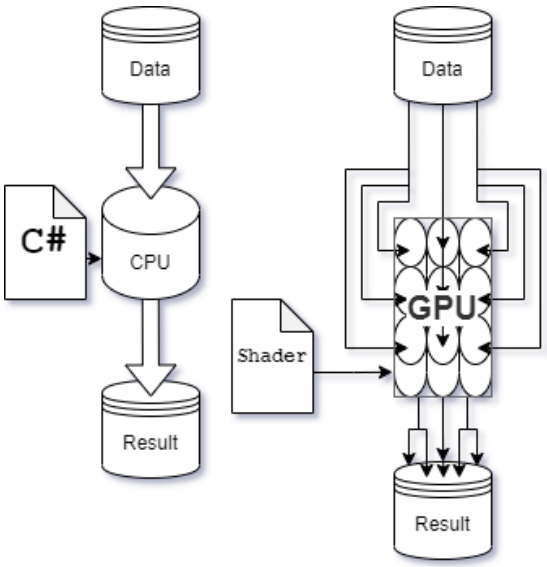


Figure 3.4: Schematic overview of CPU vs GPU processing

3.3 Proposed solution

The tool will contain an interactive GUI which exposes the procedural properties to the user. The GUI is linked with a game object that generates a preview planet. This preview planet will highlight the changes made or show the final planetary result in real-time inside the editor.

The runtime planetary system will use a combination of the static mesh LOD methodology and two

types of dynamic LOD methodology. These two types are a quadtree which will apply dynamic LOD on a spherical object and the built-in terrain system of Unity which is only used once very close to the surface. This is because the Unity terrain only supports flat terrain. The advantage of using the terrain is the accessibility to all the additional features, being foliage and tree placement support, terrain collision, and ground texture blending.

The procedural algorithm will be fully deterministic so the same properties will generate the same planet. This way the properties can be saved to a file that allows for sharing planets and regenerating the planets after Unity restarts. The algorithm will be implemented using shaders to deal with the performance challenges as Unity provides an internal interface for running shader code.

4 CONCEPTUAL DESIGN

In this chapter, the design of the planetary terrain tool will be discussed. Starting with a brief overview of the entire tool, followed by a more specified discussion about the editor tool, the planet system, and the procedural algorithm.

4.1 The overall design of the planetary terrain tool

The design of the tool is divided into two main phases: the editor phase and the runtime phase. The bridge between these two phases is a save file that contains all the procedural data. During the editor phase, a GUI exposes the procedural properties to be changed by the user. As seen in figure 4.1 the changes made by the GUI are directly saved to the save file. In order to get visual feedback, a preview planet is generated in the Unity editor. This preview planet uses the procedural algorithm to generate a spherical mesh and colored material[1]. Once a change is detected by the GUI, the preview planet will regenerate with the latest procedural data.

During the runtime phase, the data from the save file is loaded and the planet starts generating the meshes for the LOD and quadtree system as seen in figure 4.1. The LOD system does this by requesting the procedural generation system for planet meshes of different detail levels and materials with different texture resolutions. This is further explained in section 5.8.1. The quadtree system does this by generating some root branches for the quadtree. These root branches also request a branch mesh and branch material from the procedural algorithm. In section 5.8.2 the quadtree system is explained in greater detail.

Depending on the distance of the target, one of the three systems will activate. Assuming the target starts far away, the LOD system activates. Once the target gets near to the planet, it toggles from LOD to a quadtree. After further approach, the planet will toggle from the quadtree to the Unity terrain and reorientate the target. If the target moves away from the planet, the same behavior, but in reversed order will occur.

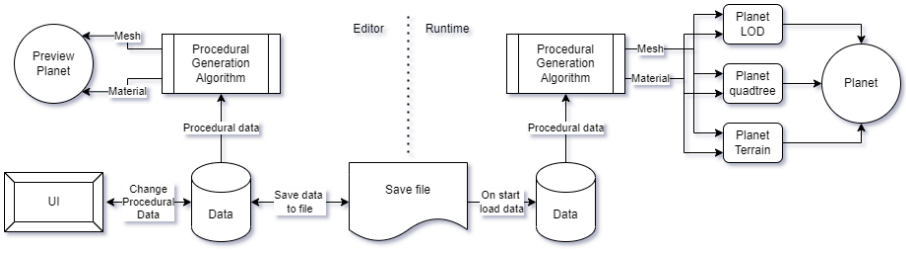


Figure 4.1: An overview of the design

[1] A material is a description of how the surface of a mesh should be visualized.

4.2 The design of the editor tool

The objective of the editor tool is to alter the procedural data. As seen in figure 4.2, the GUI displays the procedural data contained in the data object. When the GUI detects a change in values, a signal to the preview planet is sent which is visualized by the green arrow. The preview planet will request a new mesh and material from the procedural generation. This is shown by the yellow arrow. Once the procedural algorithm is finished generating, the preview planet replaces the old mesh and material with the new ones as indicated by the red arrow.

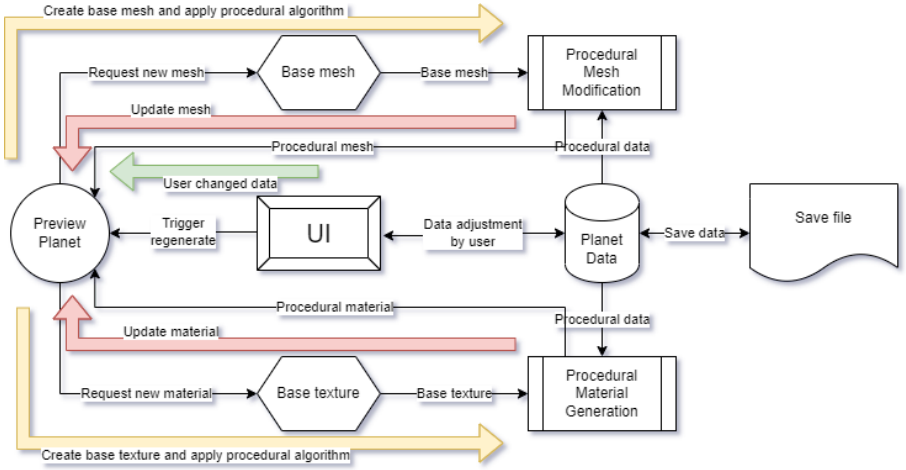


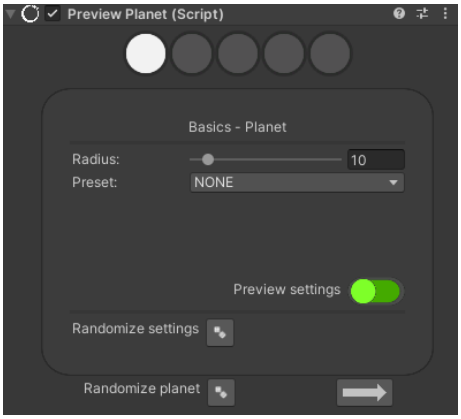
Figure 4.2: The design of the editor tool

4.2.1 The in editor GUI

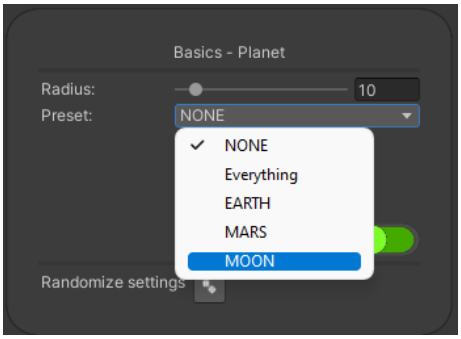
Interacting with the tool is done through the GUI. The design process of the planet is divided into multiple phases. Every phase exposes related procedural properties. The GUI starts at the first phase which exposes the celestial properties after this phase the next one will show and so on until all procedural properties are set. The GUI provides a way of revisiting previous phases without losing the progress of the later phases.

Looking at figure 4.3a, the full GUI in basics mode is visible. The multiple circles at the top refer to the different phases. A user can toggle between phases by clicking the circles. The white circle is the currently active one. A second way of navigating between phases is by pressing the left and right buttons at the bottom of the GUI. The slightly less gray rectangle between the top circle buttons and the down arrow buttons displays the phase GUI. This will be different for every phase. Next to the arrow buttons, there is a randomize[2] planet button. This button will randomize all values of every phase. Every phase also contains a randomize button which only randomizes the properties of the currently active phase. Every phase also has the preview settings button. This button determines if the preview of the final planet is shown or only the settings of the current phase.

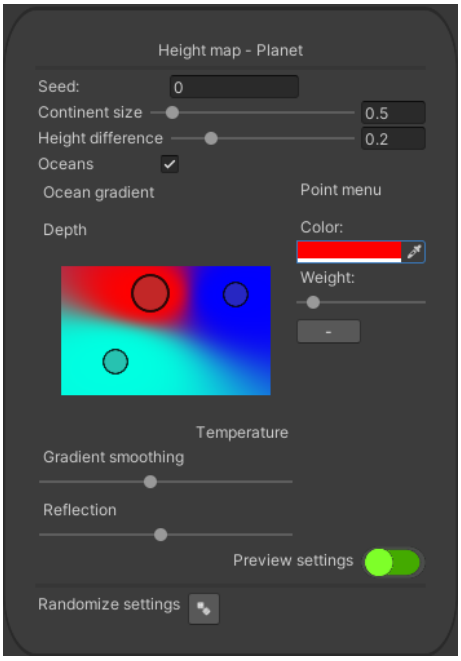
[2]Randomizing means changing the data values to a random value within the allowed range.



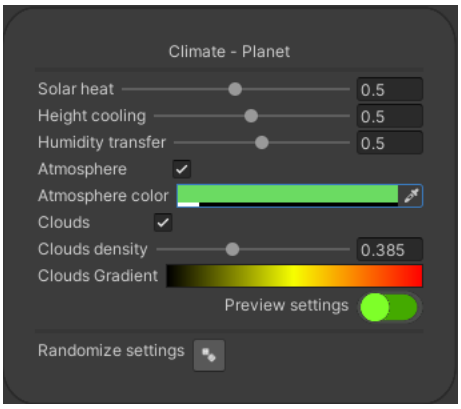
(a) The full GUI in the basics phase



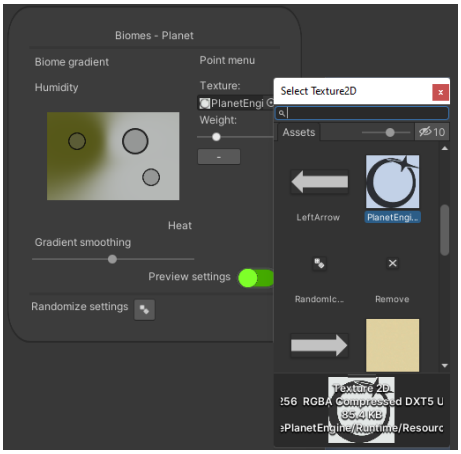
(b) The GUI of the basics phase



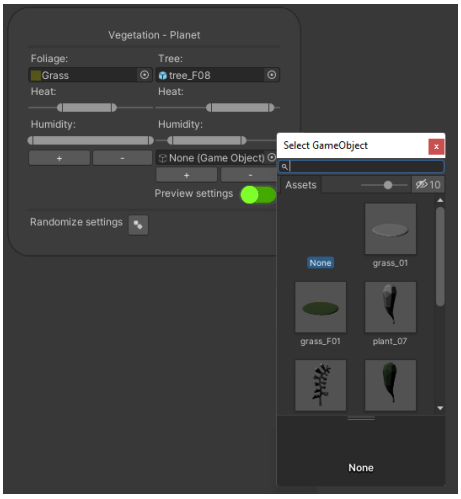
(c) The GUI of the height map phase



(d) The GUI of the climate phase



(e) The GUI of the biome phase



(f) The GUI of the vegetation phase

Figure 4.3: GUI modes

Basics

The GUI for the basic parameters is seen in figure 4.3b. In this phase, the user can adjust basic values like celestial properties or choose presets. Currently, the only celestial property is the radius of the planet. The presets are predefined configurations of the planet. This allows the user to start the planet design from a certain configuration which can be a better starting point if the target planet is similar to one of the preset configurations.

Height map

The GUI for the height map parameters is seen in figure 4.3c. In this phase, the user can adjust the height map of the planet. The properties exposed by this phase are the seed for the Simplex noise[3] generator, the size of the continents, the maximum difference in height, and the occurrence of an ocean. When the ocean is enabled, an additional GUI for the configuration of the ocean gradient[4] will show. The design of the gradient GUI is described in section 4.4.1.

Climate

The GUI for the climate parameters is seen in figure 4.3d. In this phase, the user can configure the climate of the planet. The assignment of different biomes to the planet is done through climate properties. The climate factors are the heat and the humidity on the surface. The dispersion of heat is defined by two factors: solar heat and height cooling. Solar heat indicates how much heat is received by the planet. The height cooling uses this data to lower the heat as the terrain gets higher. This results in the appearance of colder climates on higher surfaces, which is something that also occurs on earth. The humidity is calculated based on the distance from the closest body of water. This is explained in more detail in section 5.9.2.

Biomes

The GUI for the biome parameters is seen in figure 4.3e. After defining the climate of the planet, different biomes can be added to the planet. Biomes are created as points into a two-dimensional gradient. By adding points to a Heat-Humidity plane, the climate of the planet will sample this gradient and assign a biome to the surface of the planet. The difference between this gradient and the ocean gradient is that the points do not take color as input but instead a texture. This texture is used when at runtime a player comes close to the surface of the planet. In the preview and at higher distances from the surface, the averaged color of the texture is used to color the surface. This process is explained in section 5.6.

[3] Simplex noise is a type of noise which gradually changes values over an n-dimensional space

[4] A gradient defines a range of colors based on the interpolation of several base color points. This range can be used to sample different color variations for the ocean. This is discussed further in section 4.4.1

Trees and foliage

The GUI for the tree and foliage parameters is seen in figure 4.3f. The GUI shows two lists: one for the trees and one for the foliage. An object field allows for the selection of a foliage texture or a tree model from the project assets[5]. Once an object is selected, the humidity and heat slider show. These sliders define the range in which climate the tree or foliage occurs. In the current state of the tool, the assignment of climate is not functional. This is because the development of the terrain generation did not succeed within the project scope. The plus and minus button at the bottom allows, adding or removing entries to or from the foliage and tree lists.

4.2.2 Preview planet

While adjusting the values of the procedural algorithm using the GUI, the planet object in the game scene[6] generates a preview of how the planet looks. This preview planet uses the procedural generation system that is also used by the runtime planet system to generate a single mesh planet. The difference between the preview planet and the runtime planet is that the preview generation makes use of different coloring algorithms to visualize different properties linked to different phases and does not apply the height map onto the mesh. If the preview settings are set to show the final result, the material will always output the effect of all planet data combined. If preview settings are set to preview the current phase, the material will highlight the planet data contained in that phase. In order to show the different procedural properties, the procedural material generation algorithm uses different preview modes to highlight certain properties. The implementation is explained in section 5.7

Ocean preview

Figure 4.4a shows the ocean preview. During the height map phase, the material is in ocean preview mode. In this mode, the height map texture is displayed as color in gray values. If the planet contains an ocean, this will overwrite the negative height values and replace them with the sampled color from the gradient. The ocean color is dependent on the depth, which is directly read from the height map and the heat, which is read from a heat map generated using the values of the climate phase.

Heat preview

In figure 4.4b the heat preview can be seen. During the climate phase, the heat preview is shown when the user is adjusting the heat properties. This uses a three-colored gradient to indicate hot, medium, and cold areas by assigning a red, yellow, and blue color respectively.

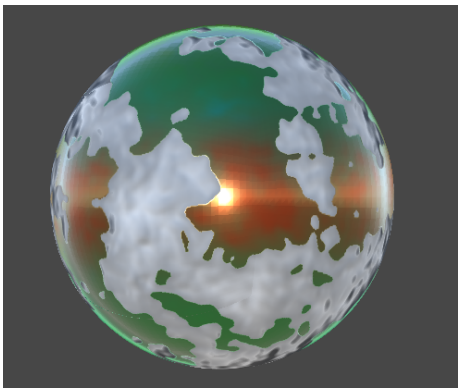
[5]The project assets are additional files like images, models, and audio, which are added to the Unity project.
[6]The game scene is the window inside the Unity editor where the visual aspect of the game is shown.

Humidity preview

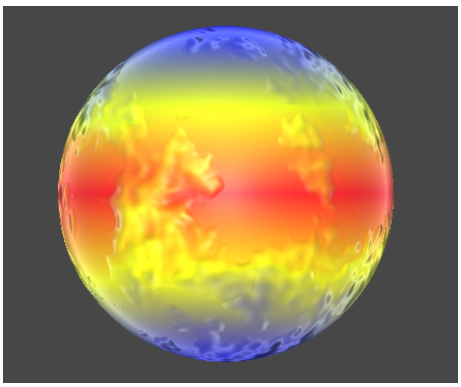
In figure 4.4c shows the humidity preview. During the climate phase, the humidity preview is also shown, when adjusting humidity properties. Here, the intensity of blue indicates the humidity.

Biome preview

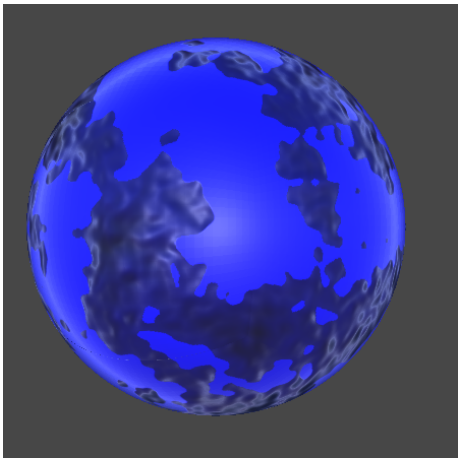
In figure 4.4d the biome preview can be seen. During the biome phase, the biome preview is shown. This mode generates a fully colored planet with the colors of the ground texture as surface colors.



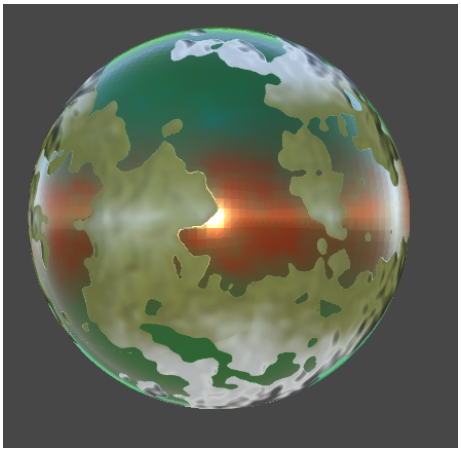
(a) The ocean preview mode



(b) The heat preview mode



(c) The humidity preview mode



(d) The biome preview mode

Figure 4.4: preview modes

4.2.3 Data storage

The procedural properties are stored in a save file. In the editor phase, the data is loaded into a data object, which exposes its values through the GUI. Once the GUI detects a change in values, the data object overrides the save file with the new values. During runtime, the file is loaded once

and a data object exposes the values to the planet system. The save file uses JSON [7] encoding. As procedural generation requires very little stored data, this will not impact the performance and allows for easier debugging. The biome gradient contains textures that are not supported by the JSON, thus the path to the textures is stored instead. This has the disadvantage that the save file is dependent on the project structure.

4.3 The design of the runtime planet system

When the user is done editing the planet parameters in the GUI, they press play to generate the final, detailed version of the planet. This is handled by the runtime planet system. This system makes use of the saved planet data from the editor designer as seen in figure 4.5. The planet will have the same size and will be located at the same position as the preview planet. This allows the user to place the preview planet at the right position in the editor and design levels with multiple planets which are all positioned differently. The runtime planet system has three states of detail handling. The first state is the single mesh state using multiple LOD's. In this state, the planet is far away and only limited detail is required. Once the player comes too close to the planet, it will switch to the second state. In this state, the planet uses a quadtree system which allows for uneven detail over the surface of the planet. Finally when the player approaches the planet's surface closely, the third state is enabled. In this state, the planet changes to a default Unity terrain system using the procedural generation of the planet surface to color, heighten, and populate the terrain with vegetation and foliage. When the player backs away from the planet the previous states will trigger again. This allows for a seamless transition from being far away from a planet to go to its surface and then back.

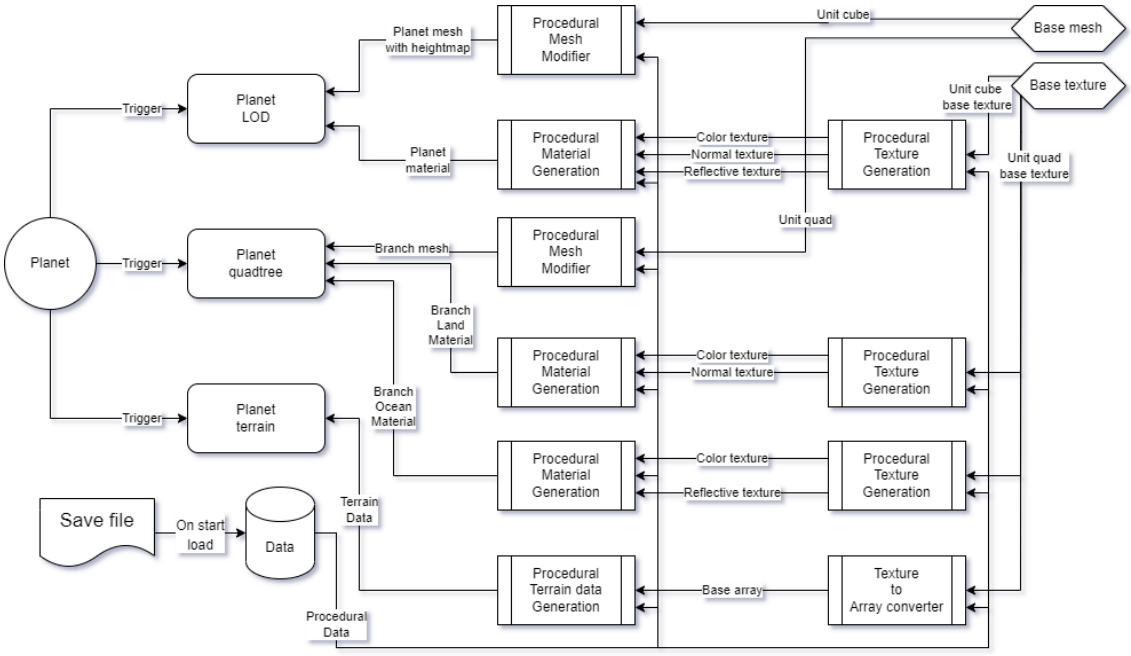


Figure 4.5: The design of the planet system

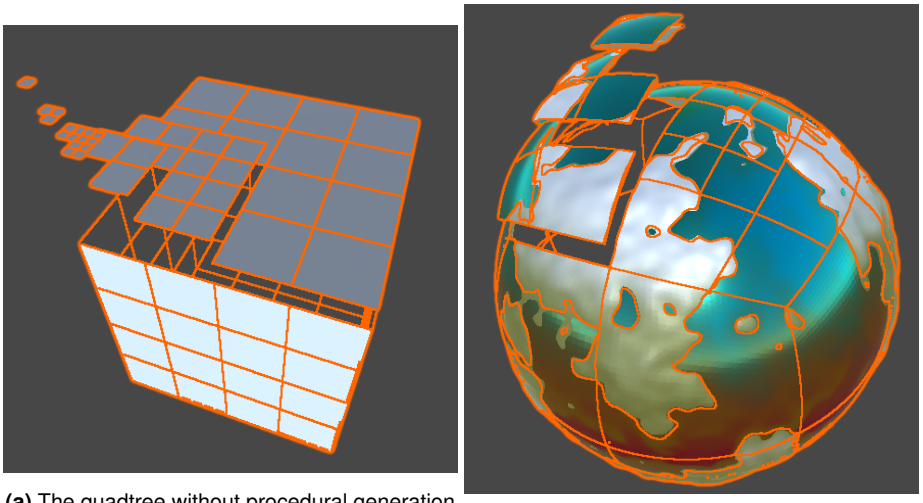
[7] JSON: JavaScript object notation is a data format for storing data in textual format. It is used to be readable by both software and humans.

4.3.1 The quadtree system

Once the player is too close to the planet, a single planetary mesh with homogeneous detail is not ideal. Instead, a dynamic LOD system takes over the default LOD system. This dynamic LOD system is called the quadtree system. The quadtree is a dedicated implementation for this tool, as Unity does not provide a spherical dynamic LOD system.

A quadtree system is a hierarchy of branches. These branches will split up into four smaller branches once a target gets close. As every branch contains the same amount of detail a division into four smaller parts means four times the detail. These new branches have the same behavior which introduces a recursive behavior. Depending on the size of the branch, the trigger distance for splitting changes as well. Once a target moves away from the branches, they will unite back to one bigger branch decreasing the detail again. As a result, only close to the target a high level of detail is achieved.

The branches in a quadtree are quads[8]. As seen in figure 4.6a the hierarchy of branches represents a unit cube[9]. To convert this cube to a planet as seen in figure 4.6b the procedural algorithm uses the quad as a base mesh and modifies it to the required shape. This base mesh input can be seen in figure 4.5 in the top right corner.



(a) The quadtree without procedural generation

(b) The quadtree with the procedural generation

Figure 4.6: quadtree example

4.3.2 Terrain system

When the player approaches the planet very closely, the built-in Unity terrain system takes over since it contains features that the quadtree system does not provide. For example, the Unity terrain engine supports foliage and tree placement which allows planets to be populated with vegetation. It also provides collision, which means the physics engine of Unity interacts with the terrain surface. Lastly, the terrain provides a dynamic LOD system just like the quadtree. However, the terrain

[8] A quad is a shape that represents a square in three-dimensional space.

[9] A unit cube is a cube with sides of 1 unit length.

system is always a horizontal planar shape with the added height map. This means that if a player approaches the planet from the side, the terrain will suddenly appear orthogonal to the planet surface.

Once the planet sends the trigger to go from quadtree to terrain, the terrain positions itself at the closest point to the player on the planet's surface. Using this location, a base texture is defined, which is used by the procedural generation to generate an object called terrain data. This terrain data defines the surface texture, vegetation, and height map of the terrain. To remove the sudden change of surface direction, the camera is moved to a new position and rotated to have the same field of view. As seen in figure 4.7.

The development of the terrain system did not finish entirely during the period of the thesis. As a result, the terrain does not support the generation of oceans.

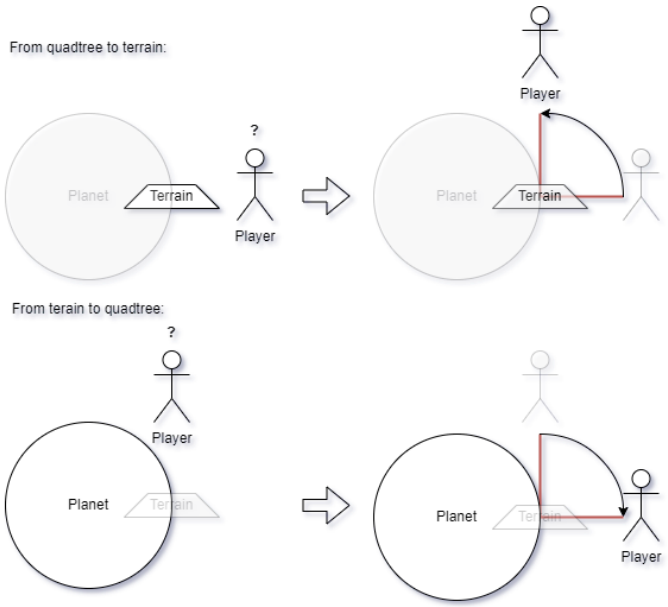


Figure 4.7: Terrain orientation visualization

4.4 The design of the parameterized procedural algorithm

The procedural algorithm is a system that defines the planetary terrain by using planet data and a base mesh or base texture input. The algorithm computes a modified mesh or texture on which the planetary data is applied. The procedural part of the algorithm is indifferent to which target system it is generating for. So the computational procedure for the mesh, texture, and terrain data is the same.

4.4.1 Procedural data

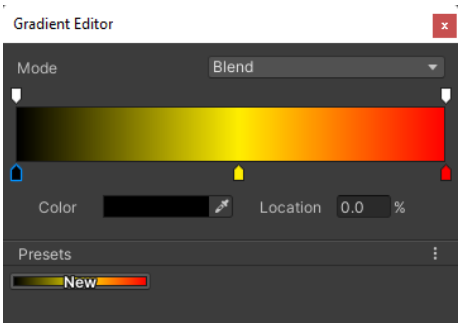
The procedural algorithm can generate deterministic planets using the procedural data defined in the editor. These data entries are values used in the procedural calculation. Two planets with the exact same properties are identical, but a slight difference in one of the values can alter the outcome of the calculation greatly. In table 4.1 an overview of the properties is given.

Procedural properties	
Property	Description
Seed	The seed is used by the Simplex noise generator to always generate the same noise.
Radius	The radius is the radius of the sphere of the planet. It is independent of the height map which is applied afterward.
Continent scale	The size of the continents. The larger this value the larger oceans and landmass are.
Height difference	The amplitude is used to heighten or lower the terrain when applying the height map.
Has ocean	An indicator if the planet has an ocean or not.
Ocean reflectiveness	The reflectiveness of the ocean material. Meaning the amount of undistorted light that is coming from the ocean surface.
Ocean gradient	The gradient 2D contains the colors of the ocean.
Solar heat	The amount of heat reaching the surface. This is a way of defining how close a planet is to the sun.
Height cooling	The amount of heat reduction because of increased height.
Humidity transfer	The amount of humidity received land inwards. It indicates the amount of rain that is received by the land surface.
Biome gradient	The gradient 2D contains the texture of the terrain and describes the color of the landmass.
Foliage	An array of foliage textures with a heat and humidity range.
Trees	An array of tree objects with a heat and humidity range.

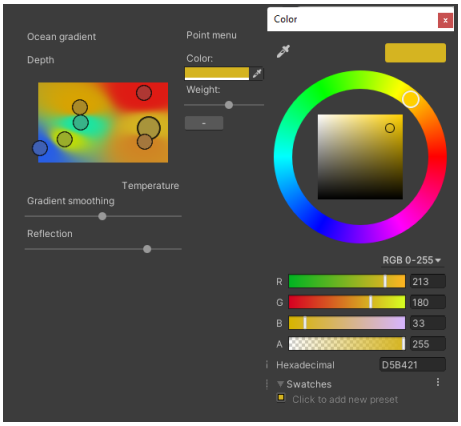
Table 4.1: A overview of the procedural properties

Two-dimensional Color Gradient

A color gradient is a type of color map which contains several points with a specified color. All space between those points contains color data that is an interpolation of the neighboring points. The Unity engine provides a one-dimensional color gradient both as an API and as a GUI element for custom editors. In figure 4.8a the API of the gradient is visualized. For this project, a two-dimensional color gradient is required. Thus a new color gradient is implemented into the tool. This new system provides an API and a GUI interface for creating two-dimensional gradients. This GUI is visualized in figure 4.8b. The gradient has two derivations, one uses a single color for every point of the gradient and the other uses a texture for every point. The average color of that texture is then used to generate the gradient. This is used for the creation of the Unity terrain data to assign ground textures to the planet's surface.



(a) Unity's 1D gradient UI



(b) The gradient2D UI

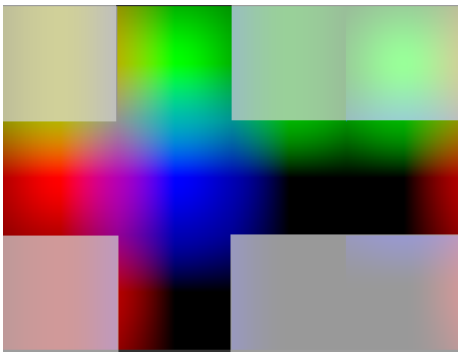
Presets

The presets are predefined planets that can be selected in the GUI. The presents are saved files stored inside the tool. As the save files require certain paths to the textures, an additional preset texture map is created in the asset folder to which the textures are copied from the tool source files. The preset options are visualized in figure 4.3b.

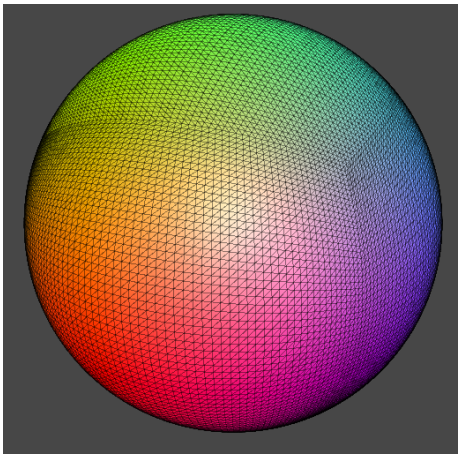
4.4.2 Mesh and texture matching

In order to match the procedural changes applied to the mesh with the procedural changes applied to the material, a method of unifying those two targets is implemented. The procedural generation algorithm requires a three-dimensional position as input to calculate the height and color. As will be explained in chapter 5.3.1, a mesh is defined by an array of three-dimensional positions. This allows the procedural generation algorithm to calculate an output for every position entry in the mesh definition. To generate textures using the procedural generation system, the same three-dimensional position data is required. To create this data, a texture that emulates the mesh is introduced. This process is described in section 5.4.1. This methodology requires that for every mesh used by the procedural generation a corresponding texture is provided as well. The tool uses two types of base meshes: a quad mesh and a unit cube mesh. The resulting texture contains a description of the mesh surface by storing the XYZ values in the RGB[10] channels of the texture. The mesh in figure 4.9b is represented by the texture in figure 4.9a. This mesh and texture were both first defined as a unit cube and were inflated to represent a spherical shape. As the up vector in Unity is defined as $(0, 1, 0)$ one can clearly see the upper part of figure 4.9a being dominated by green which indicates this represents the top part of the sphere. The branch mesh in figure 4.9d shows how the modified mesh can still be defined by the quad texture in figure 4.9c.

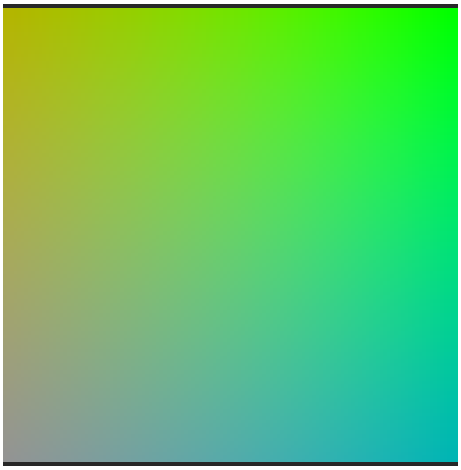
[10] RGB stands for the red, green, and blue color contained in a pixel.



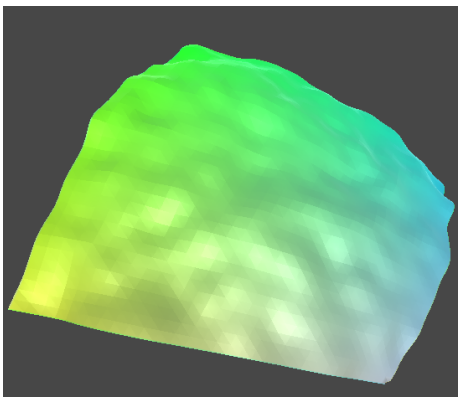
(a) Base texture of a cube



(b) A sphere with the base texture wrapped around



(c) Base texture of a branch



(d) Branch mesh with applied base texture

Figure 4.9: Base texture examples

4.4.3 Procedural mesh

Once the base mesh is generated, the procedural algorithm applies additional changes to the mesh. The first modification to the mesh is increasing the mesh size to the radius of the planet. The second modification to the mesh is the addition of the height map. This height map is not a saved texture but is generated and directly used by the mesh modifier. A more in-depth explanation is given in section 5.3;

4.4.4 Procedural material

Once the base texture is generated, the procedural generation process generates multiple textures per material. Depending on the target an ocean material, a land material, or a combined material is generated. These all have their own color and effect textures. Effect textures are textures used by the material to activate an additional part of the standard shader. This is explained further in section 5.9.2. In order to generate the color and effect textures, first, the data textures are generated. These

textures contain data that is used in multiple procedural calculations. All textures have the same size as the base texture. The generation process of the textures is described in section 5.9.2.

4.4.5 Data textures

A data texture is a texture used for storing data per pixel. These textures are not used by the material, but are used by the procedural generation as input values for the color and effect textures.

Height texture

The height texture stores a single float value for every pixel. The value indicates a positive or negative height value.

Heat texture

The heat texture stores a single float value for every pixel. The value indicates the simulated temperature at every pixel position.

Humidity texture

The humidity texture stores a single float value for every pixel. The value indicates the simulated humidity at every pixel position.

4.4.6 Color textures

The color textures contain the actual color data of the planet's surface. For full-size planets, the surface describes the ocean as well as the land with one mesh and thus one material. A mesh can have multiple materials but generating an ocean in texture space is less complex than assigning two materials to one procedural generated mesh. In the branch system, a separate branch for land and for oceans is defined. This results in dedicated land meshes and dedicated ocean meshes. These two branch types need separate materials.

Land color

The land color texture generates a color texture by sampling the biome gradient. This makes use of the humidity and heat values.

Ocean color

The ocean color texture generates a color texture by sampling the ocean gradient. This makes use of the heat and depth (height) values.

Combined color

The combined texture switches between the previous two systems depending on the height value of the specified pixel.

4.4.7 Effect textures

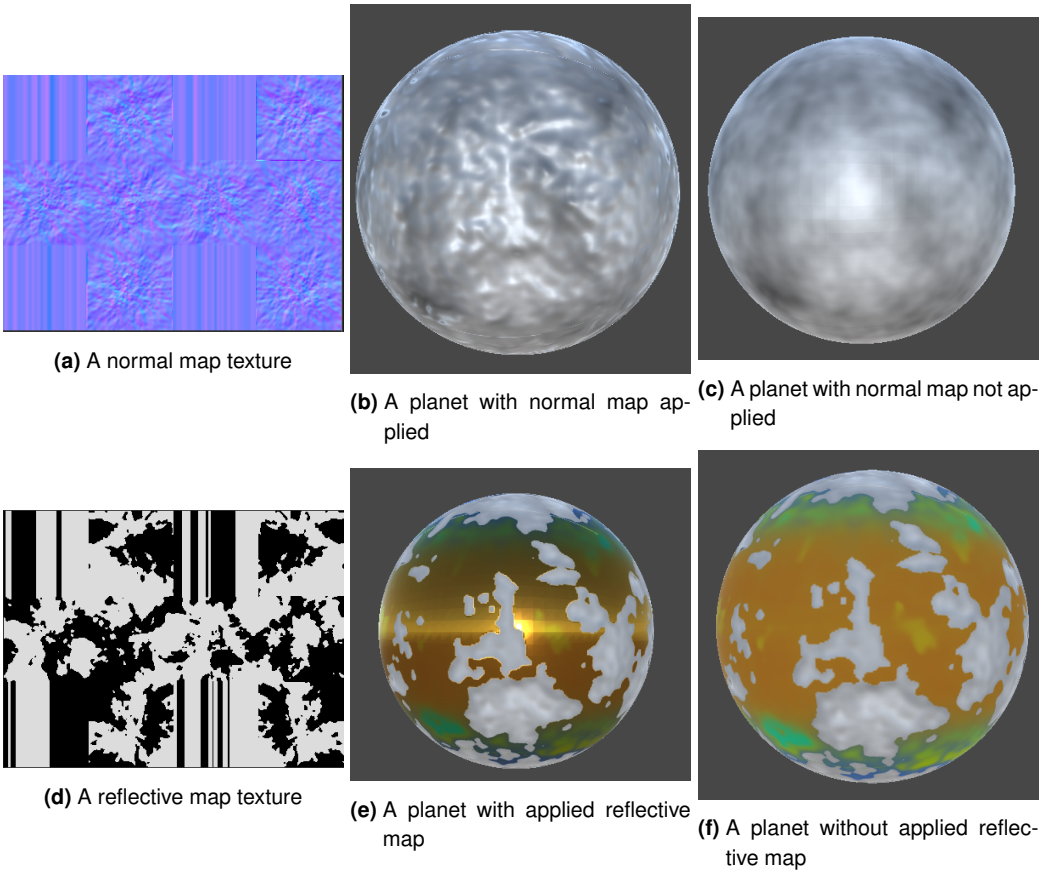
The standard shader used by Unity's materials is a collection of multiple shaders combined. Additional effects can be activated by feeding a certain value or texture to the material. The planet materials make use of normal textures and reflective textures.

Normal map

The normal map defines an override of the surface normals. These normals are used by the engine to predict the light bounce trajectory. By altering the normals on a flat surface, the illusion of an uneven surface is created. In figure 4.10a the normal map texture is visualized. It is recognizable by its blue tint. The difference between a planet with a normal map as seen in figure 4.10b and a planet without a normal map as seen in figure 4.10c is clearly visible. Although it looks like the surface of the two planets is different, it is the same mesh with the same color texture applied.

Reflective map

The reflective map defines the reflection intensity of the light for every pixel position. The reflection intensity defines the amount of light that is distorted after the encounter with the surface. High reflection intensity acts as a mirror while a low reflection intensity acts as a matte look. In figure 4.10d the reflective map texture is visualized. Because the reflective map is used to highlight the ocean the texture has clear white and black zones. This is not always the case, if a material fades between reflective and non-reflective this is also possible by using gray values. The difference between a planet with a reflective map as seen in figure 4.10e and a planet without a reflective map as seen in figure 4.10f is clearly visible.



(a) A normal map texture

(b) A planet with normal map applied

(c) A planet with normal map not applied

(d) A reflective map texture

(e) A planet with applied reflective map

(f) A planet without applied reflective map

Figure 4.10: Effect texture examples

4.4.8 Applied to the Unity terrain system

A Unity terrain requires a terrain data object. This object contains all the data to describe the terrain surface, similar to how the procedural data describes the entire surface. The difference is that the terrain does not use procedural generation and as a result, the terrain data requires data arrays instead of procedural properties. The generation process is similar to the material system. The terrain starts by generating a base texture that defines the area covered by the terrain. This base texture is used to generate a height map, heatmap, and humidity map texture. The terrain data does not require textures but uses data arrays instead, thus the height map texture is converted to a data array and by using the heatmap and humidity map, a color data array is defined as well. The foliage and tree array of the planet data is pushed to the terrain data and a temporary algorithm places the trees and foliage onto the surface.

The procedural generation of the terrain was still in development at the end of the thesis period. As a result, the humidity and heat values are not used by the vegetation placement system and no variation in the placement is introduced. This is visible in figure 4.11. Another result of the unfinished development of the terrain generation is the texture to array conversion and the vegetation placement are both implemented using the CPU instead of the GPU. This decreases the speed of the terrain generation.

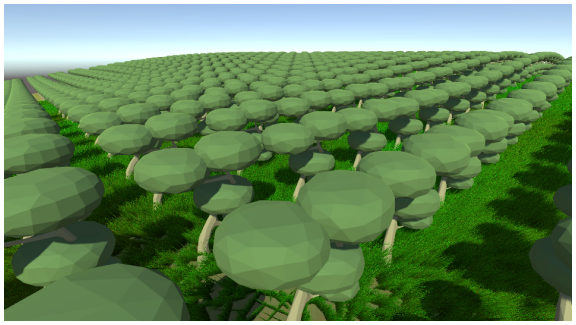


Figure 4.11: Example of regular placement of vegetation

5 IMPLEMENTATION

In this chapter, the implementation of the defined design is discussed.

5.1 Unity Engine

Every game engine has a different approach as to how to expose the vast amount of functionality contained in their environment. The Unity engine does this through a development program called the Unity editor. This editor allows the developer to organize its assets in an internal file structure. Also, the editor contains a 3D environment designer. These environments are called scenes and are similar to levels in classic game terms or to what a scene is in the movie and TV industry. These scenes can be populated by game objects which are objects who exist in three-dimensional space. These objects by themselves have limited functionality. To add functionality to the objects, the concept of components is introduced. A component represents a property of the object, meaning a certain behavior the object contains. An object can have multiple components which can influence each other. The Unity engine contains a wide range of components that cover a large range of aspects of game development. However, every game is unique and as such, it would be impossible to cover every possible behavior as a component in the engine. Because of this, developers can implement their components using C#. This way developers can design a game object with many common game design elements and manage their behavior using a custom component created by themselves. Or create a completely new behavior by also creating a custom component.

5.2 Procedural Data

The procedural data is stored in a JSON encoded save file. The encoding from data object to string is performed using the JSON utility, which is part of the Unity engine. This encoded string is stored as a text file using the Unity asset database. This is listed in figure 5.1. In order to make an object compatible with the JSON utility, the object has to be serializable. This means that every public property must have a primary type or a type that is serializable. In the case of the Procedural data, this requires the Gradient2D and the vegetation references to be serializable. The Gradient2D in turn requires the GradientPoints to be serializable. Because of this, the texture and prefab references stored by the GradientPoints and the vegetation references are stored as the path to the assets in the Asset directory.

The randomization of values is also defined inside the procedural data object. The randomization of the procedural properties is grouped per design phase. The GUI also provides a randomization button for the entire planet, which activates every phase randomizer simultaneously.

```
1  public void SaveData(string name)
2  {
3      string path = $"Assets/PlanetEngineData/{name}-planetData.asset";
4      TextAsset file = new TextAsset(JsonUtility.ToJson(this, true));
5      AssetDatabase.CreateAsset(file, path);
6  }
7
8  public void LoadData(string name)
9  {
10     string path = $"Assets/PlanetEngineData/{name}-planetData.asset";
11     TextAsset file = AssetDatabase.LoadAssetAtPath<TextAsset>(path);
12     if (file == null) return;
13     JsonUtility.FromJsonOverwrite(file.text, this);
14 }
```

Figure 5.1: The saving and loading of the procedural data

5.3 Mesh generation

In the folowing paragraph, a brief explanation of how these structures work is presented followed by the implemented methodology used in this tool.

5.3.1 The definition of a mesh

A mesh is a digital description of a three-dimensional surface of a shape. This is done by storing several arrays containing data describing the shape. A mesh can contain a lot of types of data, but two fundamental arrays are required. The first array is a structured array of three-dimensional vectors. This array is called the vertex array and the three-dimensional vectors are called vertices. The second array is a structured array of integers. This array is called the index array and the integers are called indices. When a mesh uses just these two arrays, it describes a pure geometrical shape. [11]

The shape is defined by a combination of groups of three indices. Every index points to a location in the vertex array where a certain vertex is stored. This implies that every group references vertices. These three points make up a triangle shape, which is the minimal primitive shape to describe a three-dimensional surface. As indices are pointers to the vertex array, two or more triangles can make use of the same vertex. This removes the need to duplicate vertices when two triangles share vertices to create a more complex three-dimensional surface. Describing a three-dimensional shape is done by approximating that shape by stitching many triangles together to create a surface of the desired shape.

For a game engine like Unity, a mesh requires one additional convention to visualize it. The reference of the vertices in a triangle should be consistent in clockwise order, but this can be counter-clockwise for other engines. The Unity engine uses this orientation to assign a direction to the surface. By giving a direction to a surface, a distinction between surfaces facing the screen and surfaces which do not face the screen can be made. This improves the performance by reducing the number of triangles that have to be drawn to the screen as only the ones facing the screen are rendered. In figure 5.2 the vertex and index array of a unit cube are visualized.

Using the vertex and index array, we can only define the shape of an object and assign a single color. To add multiple colors to the shape, a texture is wrapped around the shape. In this wrapping process, an additional array called, the UV array is used. U and V stand for the name of the axis of the normalized texture coordinates. The array contains a two-dimensional vector, called a UV coordinate, for every vertex vector in the vertex array. This UV coordinate indicates the position of the vertex on the texture. As the UV coordinates have the same location in the array as the vertices, a triangle can be cut out from the image using the three UV coordinates at the index positions. Therefore, a sliced texture can be drawn instead of a single color triangle. In figure 5.2 the application of the texture onto the mesh surface is visualized. In this figure, the corresponding UV and vertex index locations are clearly visualized.

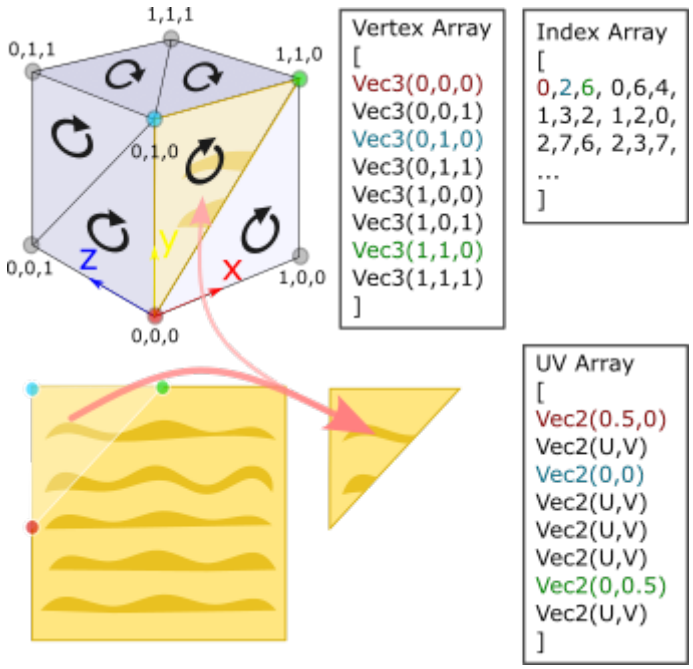


Figure 5.2: An overview of the different mesh arrays

5.3.2 Basic meshes

The entire tool uses two basic meshes to generate all planetary terrain shapes: a unit quad mesh and a unit cube mesh. Although Unity provides these primitives inside the engine, as they are heavily modified by multiple modifiers, full control of the basic shape enables easier implementation of the modifiers. By assigning multiple arrays to the mesh object, the mesh shape is defined.

In figure 5.3, the source code to create a quad mesh inside unity is listed. For the quad mesh, the UV coordinates are easily defined as a quad is a similar size as an image, rectangular. In the tool primitive meshes are generated by predefined arrays embedded in static C# code. Because the tool generates meshes with over 1 million vertices, the data type of the mesh index values has to be overwritten to int32 as the default only allows up to 64 thousand vertices.

The unit cube, however uses a more complex pattern to define the UVs onto the cube. As the cube has 6 rectangular surfaces, a folded-out texture pattern is used instead. As can be seen in

```
static Mesh GenerateUnitQuadMesh()
{
    // Create vertex, index and UV arrays
    List<Vector3> newVertices = new List<Vector3>() {
                new Vector3(-0.5f,0,-0.5f),
                new Vector3(-0.5f,0,0.5f),
                new Vector3(0.5f,0,0.5f),
                new Vector3(0.5f,0,-0.5f)
            };

    List<int> newIndexes = new List<int>(){
                0,1,2, 0,2,3
            };

    List<Vector2> newUV = new List<Vector2>() {
                new Vector2(0,0),
                new Vector2(0,1),
                new Vector2(1,1),
                new Vector2(1,0)
            };

    // Create a Mesh object set to int32 format so mesh can hold more as 64k vertices
    Mesh mesh = new Mesh();
    mesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32;
    // Assign arrays
    mesh.vertices = newVertices.ToArray();
    mesh.triangles = newIndexes.ToArray();
    mesh.uv = newUV.ToArray();
    return mesh;
}
```

Figure 5.3: A static function creating a quad mesh with UV coordinates

figure 5.4, this texture is only used for $6/12$ or $1/2$ of the entire texture surface. This inefficient configuration of the texture is used because it simplifies the implementation of the texture shaders.

5.4 Material generation

Material is an instance of a program that defines how a mesh surface is drawn to the screen. This program itself is called a shader. Unity standard materials make use of a special shader called the standard shader. This shader is not a single program, but instead a collection of different programs that can work together to generate one single output. In order to create a material, these different shaders are required to be enabled. Depending on which textures the procedural generation outputs, the right shaders are activated.

5.4.1 Base texture generation

In order to generate a base texture, the target shape has to be considered. The tool contains three methods of generating a base texture. One generates a unit cube in texture space, the other two generate a quad shape in texture space.

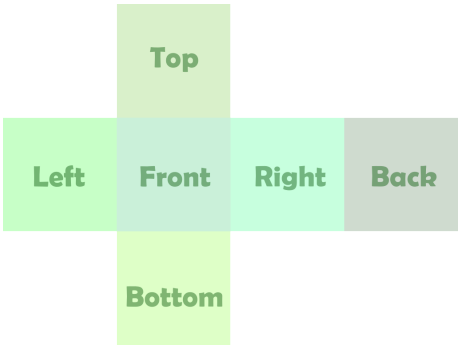


Figure 5.4: The layout of space used by the UVs

Unit cube base texture generation

The unit cube texture contains six zones as visualized in figure 5.4. Every zone is one side of the cube and every side is orthogonal with one of the axes. Therefore, every zone has one coordinate value that will be constant, which leaves two coordinate values that will change with the texture coordinates. For example, every vertex on the top side will be of the format $(x,1,z)$ while the front side will have the format $(x,y,1)$. This allows three-dimensional vectors to be calculated by assigning the constant value directly and by calculating the changing values using the UV coordinates.

Using the layout as visualized in figure 5.4, a mathematical expression for the relations $x(U)$, $y(V)$ and $z(U,V)$ can be formulated. As the X, Y, and Z values are stored in a texture, the positive part of the values can be visualized as the R, G, and B channels respectively. The $x(U)$ relation is defined by equation (5.1) and is visualized by figure 5.5a as the red channel of the texture. The $y(V)$ relation is defined by equation (5.2) and is visualized by figure 5.5b as the green channel of the texture. The $z(U,V)$ relation is defined by equation (5.3) and is visualized by figure 5.5c as the blue channel of the texture. The textures in figure 5.5 are the separated channels from figure 4.9a which is a description of a spherical mesh. This means that the visualized colors are slightly different as the output of $x(U)$, $y(V)$, and $z(U,V)$. An exact description of these textures is $\frac{x(U)}{|\vec{v}|}$, $\frac{y(V)}{|\vec{v}|}$ and $\frac{z(U,V)}{|\vec{v}|}$ in which $\vec{v} = (x(U), y(V), z(U,V))$.

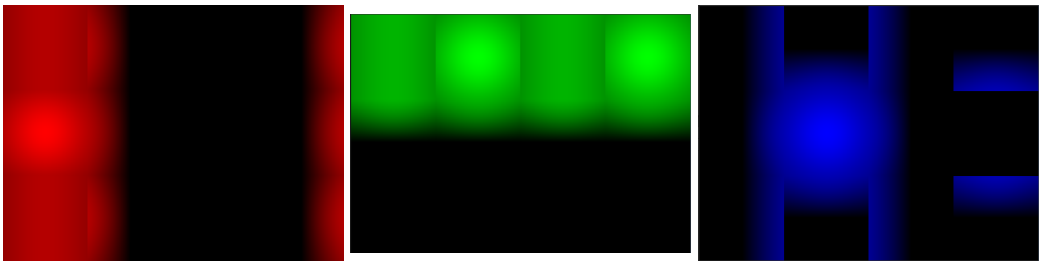
$$x(U) = \begin{cases} 4U-3 & : 3/4 < U \\ 0 & : 1/2 < U \leq 3/4 \\ 2-4U & : 1/4 < U \leq 1/2 \\ 1 & : U \leq 1/4 \end{cases} \tag{5.1}$$

$$y(V) = \begin{cases} 1 & : 2/3 < V \\ 3V-1 & : 1/3 < V \leq 2/3 \\ 0 & : V \leq 1/3 \end{cases} \tag{5.2}$$

$$z(U,V) = \begin{cases} 1 & : (1/4 < U \leq 1/2) \cup (1/3 < V < 1/3) \\ 0 & : (3/4 < U) \cup (1/3 < V < 1/3) \\ z_u(U,V)) & : other \end{cases}$$

$$z_u(U,V) = \begin{cases} z_v(V) & : 3/4 < U \\ 3-4U & : 1/2 < U \leq 3/4 \\ z_v(V) & : 1/4 < U \leq 1/2 \\ 4U & : U \leq 1/4 \end{cases} \tag{5.3}$$

$$z_v(V) = \begin{cases} 3V & : 2/3 < V \\ 3(V-1) & : V \leq 1/3 \end{cases}$$



(a) red channel of base texture **(b)** green channel of base texture **(c)** blue channel of base texture

Figure 5.5: A schematic overview of the branch systems

Quad base texture generation

There are two quad base texture generation algorithms. One generates a unit quad base texture, the other cuts out an area of another base texture.

The unit quad base texture generation has a similar pattern as the unit cube base texture generation with the advantage of only describing one quad and thus one area. This result in the complete use of the texture area. The algorithm requires a direction to orientate the quad. In table 5.1 the coordinate calculation for every direction is defined.

Quad definitions			
Side	Calculation	Side	Calculation
Front:	$(x(U) = 1-U, y(V) = V, z = 1)$	Back:	$(x(U) = U, y(V) = V, z = 0)$
Left:	$(x = 1, y(V) = V, z(U) = U)$	Right:	$(x = 0, y(V) = V, z(U) = 1-U)$
Top:	$(x(U) = U, y = 1, z(V) = V)$	Bottom:	$(x(U) = 1-U, y = 0, z(V) = V)$

Table 5.1: Every cube side and the calculation from texture to mesh space

The cut-out algorithm requires four input vertices. These are defined as top-left, top-right, bottom-left, and bottom-right. These vertices can be sampled from a different base texture or provided by an external source[1]. By interpolating the top-left with the top-right and the bottom-left with the bottom-right using the U value, the new vertex value is calculated by interpolating the results of previous interpolations by the V value.

[1] The terrain uses this base texture system to feeds the corners of the mesh in normalized form.

5.5 In-editor GUI

The GUI in the editor is created using Unity's IMGUI system. By creating a new class that inherits from the Editor class, a GUI inside the editor can be created. This class replaces the default editor of a target component. The GUI of this tool is an override of the preview planet component. In order to create a GUI layout, a set of commands has to be called by the inherited OnInspectorGUI method. The layout is created by calling static methods of the GUI, GUILayout, EditorGUI, and EditorGUILayout classes. By calling these static methods using the Layout classes unity will automatically place the elements in the order they are called. Additional commands to change the sorting order enable GUI design through code. As an example in figure 5.6 a minimal GUI is implemented for a class called PreviewPlanet. Both the example code is listed as the output GUI is visualized.

```
1  [CustomEditor(typeof(PreviewPlanet))]
2  public class ExampleUI : Editor
3  {
4      string text = "";
5
6      public override void OnInspectorGUI()
7      {
8          PreviewPlanet objectTarget = (PreviewPlanet)target;
9          GUILayout.Label($"This is an Example UI of {objectTarget.name}");
10         GUILayout.BeginHorizontal();
11         text = GUILayout.TextField(text);
12         if (GUILayout.Button("Clear"))
13         {
14             text = "";
15         }
16         GUILayout.EndHorizontal();
17     }
18 }
```

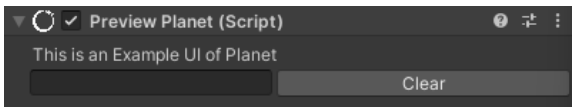


Figure 5.6: Example code of how a GUI can be coded and the result of the example GUI

5.6 Gradient 2D

The 2D color gradient is defined using two struct types: the Gradient2D struct and the GradientPoint struct. The Gradient2D struct describes a single gradient like the ocean gradient or the biome gradient. The Gradient2D contains a collection of GradientPoints that hold position, weight, color, and possibly texture data. Both struct types are serializable. The GradientPoint has a special system for loading and saving textures. All positions are defined in a normalized region and weights have a value between zero and infinity, but this is limited by the GUI itself.

5.6.1 Color interpolation

In order to retrieve a color from the Gradient2D, a two-dimensional position is required. The color interpolation is done by calculating the weighted average of all colored points. This calculation is listed in figure 5.7. The weight is determined by both the colored points' weight value and the distance between the point and the current position. In order to adjust the impact of the distance of the points, a smooth factor is introduced. This value increases or decreases the distance exponentially. This allows the weights to be dominated by the points' defined weight or by the distance of the points toward the target position. The calculation is implemented using shaders, which allows for parallel calculation of multiple pixels when generating the gradient texture.

```
1  // Get an x and y value from the thread id (= unique 3D index for every thread).
2  float x = (float)id.x / width;
3  float y = (float)id.y / height;
4
5  float4 color_sum = 0;
6  float weight_sum = 0;
7
8  // For every defined point in the gradient calculate the weight and color
9  for (int i = 0; i < point_count; i++)
10 {
11     float dist = distance(point_position[i], float2(x, y));
12     dist = pow(dist, smooth); // Apply smoothing based on procedural properties
13     float weight = (dist == 0 ? 1 : 1 / dist) * point_weight[i];
14     color_sum += point_color[i] * weight;
15     weight_sum += weight;
16 }
17 if (weight_sum != 0) color_sum /= weight_sum;
18 color_sum.a = 1; // Set transparency to opague
19 gradient_texture_out[id.xy] = color_sum;
```

Figure 5.7: The shader code for calculating the gradient color.

5.6.2 Using the gradient in texture generation

Color textures use the gradient to define a certain color. This is done by first generating a gradient texture which functions as a lookup table for the texture generation algorithm. The width and height of the texture are used as depth, heat, or humidity axis. So for every (depth, heat) or (heat, humidity) couple, a certain color can be sampled from the texture. The gradient texture is always generated at the same resolution as the target color texture for simplification of the generation algorithm. A schematic overview of how the algorithm goes from base texture to the final colored texture is shown in figure 5.8

5.7 Preview planet

The preview planet uses a single mesh that is generated out of the basic cube mesh which is processed by the procedural algorithm to a sphere with the planet's radius. This is explained further in section 5.9.1. In order to display the height map, the preview planet makes use of the normal

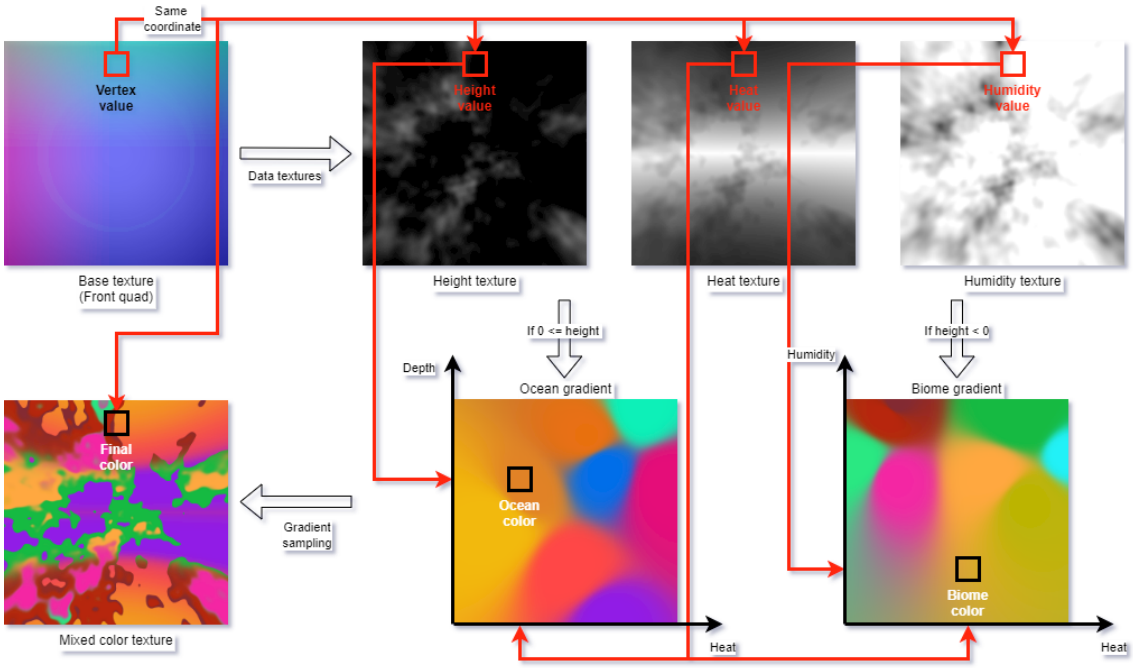


Figure 5.8: A schematic overview of the gradient sampling

textures to create the illusion of an altered sphere surface. As the preview planet displays different types of previews, it uses a custom material creation process that, based on the design phase has a different generation pipeline. At the end of every process, three textures are generated: a color texture, a normal texture, and a reflective texture.

5.8 Run-time planet system

The run-time planet has three systems: the LOD system, the quadtree system, and the terrain system. These three systems all use the same methodology. First, they generate a base mesh and base texture. Then they apply the procedural generation on the base data type to get the right mesh, material, or terrain data.

5.8.1 LOD system

The LOD system uses the built-in Unity LOD component. The component is based on the enabling and disabling of the mesh renderer. The mesh renderer is the component responsible for making a mesh visible. If this is disabled, the mesh is still loaded in memory, but it is not processed by the rendering engine. The mesh renderer also contains a bounds property which exposes the size of a mesh and its location. Using the bounds, the LOD component toggles between mesh renderers based on the projection size onto the screen. [20]

Once the planet is created, three LOD meshes are generated. As seen in table 5.2, every level increases the mesh and texture size. The mesh is generated by subdividing a unit cube base mesh to the right detail level and then it is fed to the procedural generation. The algorithm will apply the

radius and height map to the planet's surface. For the material, a unit cube base texture with the right resolution is fed to the procedural generation which generates a material with textures using the same resolution. This material displays both ocean and land colors.

Level of detail				
Level of detail	Mesh vertex count	Mesh index count	Mesh triangle count	Texture resolution
1	72	144	48	256x192
2	1152	2304	768	512x384
3	18432	36864	12288	1024x768

Table 5.2: Level of detail mesh and texture data

5.8.2 Quad tree system

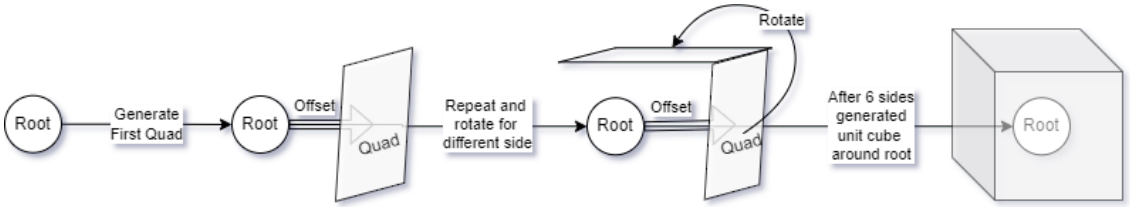
The quadtree system is, together with the procedural algorithm, the most complex part of the tool. The system works as follows. A root object creates 6 root branches. These 6 root branches contain a quad mesh which is rescaled 1x1units and pushed forward 0.5 units in local space. Then every root branch is orientated in a different direction such that the 6 branches collectively create a unit cube. The branches store the bounds of the quad and then apply the procedural algorithm, which modifies the mesh and generates material for that branch. If the planet contains an ocean, the branches will create an ocean branch which is a child object to the branch itself. This ocean branch uses the quad mesh of the parent branch to initiate the procedural algorithm, which modifies the mesh to be an ocean mesh with ocean material. This process is visualized, in figure 5.9a.

The mesh modification for a branch is similar to the single mesh planet. First, the radius of the planet is applied, followed by the application of the height map. An extra step is needed here to recenter the branch around its origin. This is done by getting the mesh bounds and offsetting the mesh so the center of the bounds is in the center of the mesh in local space. The offset is stored and the branch will move to the offset in respect of the planet center. This process is visualized, in figure 5.9b.

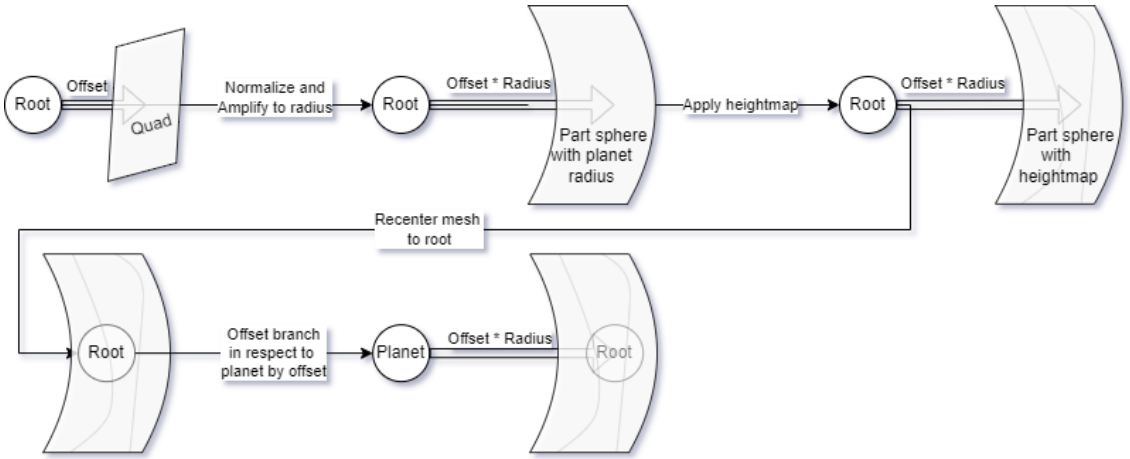
Once the target comes closer to the surface, some branches will trigger and expand. When a branch expands, it will create 4 new child branches and disable its own renderer and that of a child ocean branch if present. Every child branch gets a dedicated zone assigned. These zones indicate which part of the parent branch the current branch is. Using this zone, the branch generates a new base texture based on the base texture of the parent and calculates an offset based on the provided zone. The diagonal of the branch is 1/2th the diagonal of the parent branch. Using this base texture, offset and size the branch has the same initiating parameters as the root branch and the following of the process is exactly the same. As a result, the branch expands in the same way as the root branch expanded. This process is visualized in figure 5.9c.

When the target moves away from a branch, a parent branch will trigger and shrink again. When shrinking, the branch destroys all children except for a possible ocean branch and enables the renderer of itself and the ocean branch.

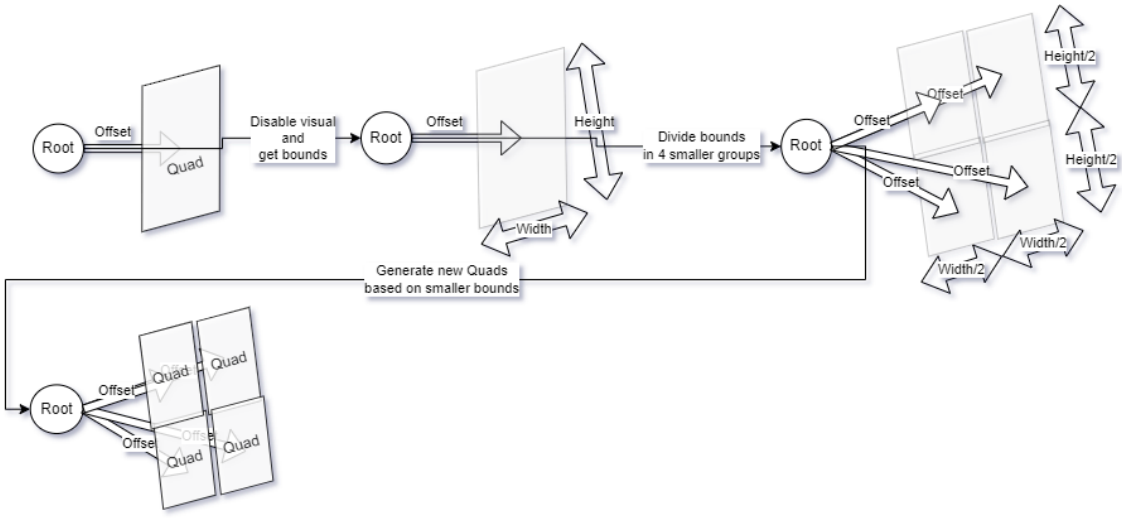
The quadtree system is defined in both the mesh and texture space. Because the algorithm is based on the base mesh and base texture it is independent of the procedural generation algorithm. This allows the procedural system to be changed without breaking the quadtree system.



(a) A schematic overview of the generation of the branch



(b) A schematic overview of the modification of the branch



(c) A schematic overview of the division of the branch

Figure 5.9: A schematic overview of the branch systems

5.9 Procedural algorithm

The procedural generation algorithm is responsible for converting base mesh or base texture in combination with the procedural data into planetary surface mesh, material, or terrain data. The algorithm is optimized by making use of the GPU. This is done by defining the highly parallel procedures into shaders. A schematic overview of the entire procedural generation structure is provided in figure 5.10. The overview shows the flow of data inside the algorithm. Starting from the left, the input data flows to the right until the data is received by the shaders. These shaders use the procedural library which defines all procedural calculations and applies the results to the input data. This modified data is then sent back to the left side and the modified mesh, texture, or terrain data is returned.

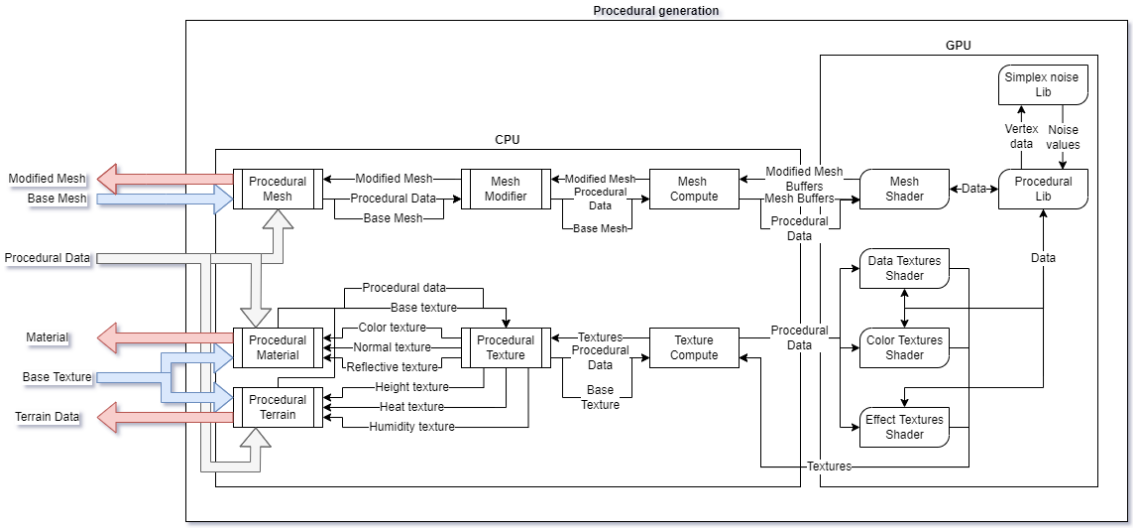


Figure 5.10: The structure of the procedural generation algorithm

5.9.1 Adjusting the mesh

The procedural modification of a base mesh starts with the procedural mesh step. This step defines a series of mesh modifications that shape a base mesh into a planetary surface mesh. For example, the branch meshes will go through the same modifications, except the ocean mesh skips the heightmap modification step. This way one branch has mountains applied to it while the other is perfectly spherical. In figure 5.11, the source code of the quad generation used by the quadtree system is listed. The explanation of the quadtree used new quads which were defined by the size and offset. This definition generates a new quad using those properties.

As seen in figure 5.11 and in figure 5.10, the mesh modifications are performed by the mesh modifier step. This step defines one single type of change to the input mesh. Using a mesh compute interface to communicate with the shaders, the main purpose of the mesh modifier itself is choosing which shader kernel is activated and deciding which data has to be passed to the shader.

The mesh compute step visualized in figure 5.10 is an interface used for interacting with the shaders. The interface manages the memory (de)allocation and alters data types to be compatible with the shader primitive types.

```
1 public static Mesh GetBranchPlaneMesh(float size, Vector3 offset)
2 {
3     Mesh planeMesh = MeshPrimitives.UnitQuad;
4     planeMesh = MeshModifier.NormalizeAndAmplify(planeMesh, size);
5     planeMesh = MeshModifier.Offset(planeMesh, offset);
6     planeMesh = MeshModifier.Subdivide(planeMesh, 5);
7     return planeMesh;
8 }
```

Figure 5.11: Procedural mesh generation of the base quad for the quadtree

The mesh shader contains multiple kernels. These kernels are definitions of a process the GPU can perform. Every modifier has a dedicated kernel. Inside the kernel, the procedural modification is defined. For the mesh shader, four kernels are defined.

The offset modifier

The offset modifier moves the vertices in mesh/local space. This is done by providing an offset vector that is added to every single vertex of the mesh. The formula (5.4) describes the calculation done by the shader.

$$\vec{v_{out}} = \vec{v_{in}} + \vec{p_{offset}} \tag{5.4}$$

The normalize and amplify modifier

The normalize and amplify modifier normalizes all vertices and multiplies the normalized vertex with a given amplitude. The formula (5.6) describes the calculation done by the shader.

$$\vec{v_{out}} = \frac{\vec{v_{in}}}{|v_{in}|} * A \tag{5.5}$$

The height-map modifier

The height-map modifier adjusts the height of vertices using the procedural library shader. This shader provides multiple functions to calculate procedural values, like the height of the terrain. This library uses a third-party MIT-licensed Simplex noise shader to generate the height values.

The branch meshes are centered meshes which, as a result, can not use the normalization of a vertex to provide an orthogonal normal to the surface. Therefore, the vertex data is first converted to be relative to the planet instead of the mesh. This is done by providing a transformation matrix from object to world space. This matrix is generated by the unity transform component. The conversion is defined by equation (5.6a). After this, the vertex is defined in world space. To make it relative to the planet, the planet position is subtracted from the vertex as defined in equation (5.6b). The resulting vertex is used by the procedural library to calculate a height value as defined in equation (5.6c). This height value is then used to create a heightened vertex by multiplying the vertex with a scaled amplitude as defined in (5.6d). To convert the vertex back to mesh space, a second transformation matrix which defines a world-to-object transformation. To convert the vertex back to

world space, first, the world positions should be added. The final vertex calculation is defined by equation (5.6e).

$$\vec{v_{world}} = M_{4x4}^{Obj->World} \times \vec{v_{in}} \tag{5.6a}$$

$$\vec{v_{planet}} = \vec{v_{world}} - \vec{p_{world}} \tag{5.6b}$$

$$h = ProceduralLib(\frac{\vec{v_{planet}}}{|v_{planet}|}) \tag{5.6c}$$

$$\vec{v_{height\_vertex}} = (1 + scaler * h) * \vec{v_{planet}} \tag{5.6d}$$

$$\vec{v_{out}} = M_{4x4}^{World->Obj} \times (\vec{p_{world}} + \vec{v_{height\_vertex}}) \tag{5.6e}$$

The subdivision modifier

The subdivision modifier divides every triangle into four smaller triangles. This doubles the number of vertices and increases the index count four times. As seen in figure 5.12, the algorithm works on a per triangle basis. By interpolating between the vertices of every side of the triangle, three new vertices are generated. By replacing the original triangle with four smaller triangles as defined in figure 5.12, the detail level of the mesh is increased while the surface has the exact same shape.

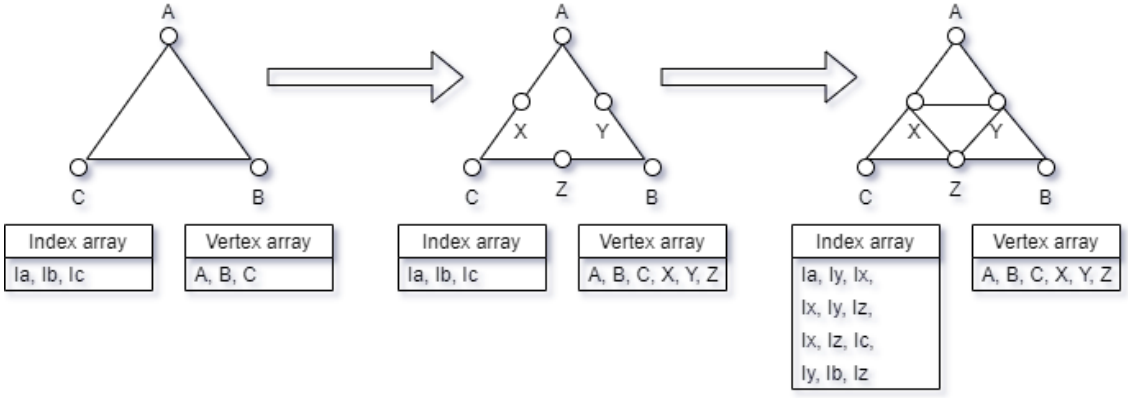


Figure 5.12: A schematic overview of the subdivision modifier

5.9.2 Generating Materials

The procedural material generation uses a similar methodology as the mesh generation system. As seen in figure 5.10. By providing the procedural generation with a base texture, similar steps will be performed where the procedural material step defines which textures are required. The procedural texture step uses the texture compute interface to generate textures using shaders. There are three types of shaders: data shaders, color shaders, and effect shaders. These all contain multiple kernels dedicated to data, color, or effect textures. These textures are defined in section 4.4.4.

Heightmap Texture

The heightmap texture is a data texture and is of the type Rfloat, which means the texture contains only one channel with type float. The value stored in this channel is the direct result of the height calculation performed by the procedural library. The calculation of the height is defined by equation (5.7).

$$height = \frac{1}{d} * \sum_{i=1}^{10} \frac{SimplexNoise(A_{continent\_size} * i * \vec{v_{base\_texture}} + seed)}{i} \quad with \quad d = \sum_{i=1}^{10} \frac{1}{i} \tag{5.7}$$

Heatmap Texture

The heatmap texture is a data texture and is of the type Rfloat. The value stored in this channel is the direct result of the heat calculation performed by the procedural library. The calculation of the heat is defined by equation (5.8).

$$heat = (1 - |y_{vertex}|)^{2*(1-solar\_heat)} * (\frac{1 - height}{2})^{height\_cooling} \tag{5.8}$$

Humidity Texture

The humidity texture is a data texture and is of the type Rfloat. The value stored in this channel is the direct result of the humidity calculation performed by the procedural library. This calculation of the humidity is defined by equation (5.9).

$$humidity = \sqrt{(1 - 2 * height) * humidity\_factor} \tag{5.9}$$

Biome color Texture

The biome color texture is used to display both ground and ocean colors onto the mesh surface. The color is sampled from the biome gradient and the ocean gradient. This is defined by equation (5.10).

$$color = \begin{cases} biome\_gradient\_texture[heat, humidity] & : 0 \leq height \\ ocean\_gradient\_texture[heat, -height] & : height < 0 \end{cases} \tag{5.10}$$

Ground color Texture

The ground color texture is used to display the ground color onto the mesh surface. The color is sampled from the biome gradient. This is defined by equation (5.11).

$$color = biome\_gradient\_texture[heat, humidity] \tag{5.11}$$

Ocean color Texture

The ocean color texture is used to display the ocean color onto the mesh surface. The color is sampled from the ocean gradient. This is defined by equation (5.12).

$$color = ocean\_gradient\_texture[heat, -height] \tag{5.12}$$

Heightmap color Texture

The heightmap color texture is used to display the height map and ocean onto the mesh surface. The color is sampled from the ocean gradient or converted to a gray value using the height value. This is defined by equation (5.13).

$$color = \begin{cases} height & : 0 \leq height \\ ocean\_gradient\_texture[heat, -height] & : height < 0 \end{cases} \tag{5.13}$$

Heatmap color Texture

The Heatmap color texture is used to display the heat onto the mesh surface. The color is generated using the individual color channels of the color vector. This is defined by equation (5.14).

$$color = \begin{cases} (2 * heat, 2 * heat, 1 - 2 * heat) & : 0 \leq heat \\ (1, 1 - 2 * (heat - 0.5), 0) & : heat < 0 \end{cases} \tag{5.14}$$

Humidity color Texture

The Humidity color texture is used to display the humidity on the mesh surface. The color is generated using the blue color channel. This is defined by equation (5.15).

$$color = (0, 0, humidity) \tag{5.15}$$

Normal Texture

The Normal texture is an effect texture. It is used to alter the normals used by the light calculations onto the mesh surface. These normals are vectors that are orthogonal to the surface. The vectors are defined with the Z as the up axis which is different from the Unity convention which defines the Y axis as up. The vectors are defined as starting from $(0.5, 0.5, 0.5)$ which allows the vectors to point in all directions using a 0 to 1 value. The up vector thus is defined as $(0.5, 0.5, 1)$. By

changing the X and Y values based on the height difference with the neighboring pixels, a slightly tilted vector can be calculated. This is defined in equation (5.16c). [21]

$$\Delta x = ((heightTexture[x-1,y] - (heightTexture[x+1,y]) * height\_difference + 1)/2 \quad (5.16a)$$
$$\Delta y = ((heightTexture[x,y+1] - (heightTexture[x,y-1]) * height\_difference + 1)/2 \quad (5.16b)$$
$$\vec{N} = (\Delta x, \Delta y, 1, 1) \quad (5.16c)$$

Reflective Texture

The reflective texture is an effect texture. It is used to alter the reflectiveness used by the light calculations onto the mesh surface. The assignment of the reflectiveness value is defined by equation (5.17).

$$value = \begin{cases} 0 & : 0 \leq height \\ reflectiveness & : height < 0 \end{cases} \quad (5.17)$$

5.9.3 Generating terrain data

In order to assign a shape to the terrain, the terrain requires a terrain data object. This object contains multiple arrays that define different properties of the terrain. The procedural terrain system was not entirely finished within the scope of the thesis. As a result and as seen in figure 5.10, the system uses the procedural texture step to generate procedural properties. The terrain receives three types of data: height data, color data, and vegetation data.

In order to generate texture data that corresponds with the size of the terrain, a base texture is required. The texture is created by defining a point at every corner of the terrain. These corner points are normalized and used by the cut-out base texture generation algorithm to create a base texture that is defined between those four points.

Generating height data

The terrain data defines an internal heightmap by a two-dimensional float array. By using the procedural texture generation and the base texture, a height map in texture form can be generated. By iterating over every pixel and assigning the Rfloat value to the array, this can be converted to a float array.

Generating color data

The terrain data defines the terrain ground color by interpolating between multiple textures. The data requires two arrays to display color onto the terrain. The first array is the terrain layer array. Here, every texture is stored in a separate terrain layer. This is done by iterating over all the points in the biome gradient and by creating a separate layer for every point. The second array is the alpha

array. This array is a three-dimensional float array that defines a coloring map. This coloring map defines, for every position in the height map, an alpha value for every terrain layer. Using this array, the terrain uses a weighted interpolation between all the textures. The alpha values are calculated using the same equation as the ground color texture, which is equation (5.11). The calculation is performed by the CPU instead of the GPU as a result of the unfinished system. The heat and humidity values are read from the data textures which are generated using the base texture.

Generating vegetation data

To assign vegetation to the terrain, a similar system is used as for the texture allocation. First, a prototype array is assigned, which contains a reference to the different types of foliage textures or tree models. This array is created by iterating over the foliage and tree array of the procedural data and assigning them to the terrain prototype array.

To assign foliage, a two-dimensional int array is defined for every foliage prototype. For every position in the array, the number of foliage instances that are placed on the corresponding terrain area is defined.

To assign tree instances, an array with tree instances is defined. These instances all have a reference to which prototype they belong and a position defined in a normalized area.

6 FUTURE WORK

The main focus of this thesis was the design of a new planetary terrain system. As a result, this thesis describes a base system that can be expanded on in the future. Yet as mentioned in the design and implementation chapters, the tool itself is not completely finished. In this chapter improvements to the current tool and future research are proposed.

6.1 Future improvements

In order to finish the first functional version of the tool, the procedural generation of the terrain should be further implemented. Instead of linear placement of trees and foliage, a procedural system should be implemented, which uses heat and humidity in combination with the terrain topology to generate natural-looking forests and grass fields. This in combination with water on terrain level would make the tool a usable planet terrain system.

As mentioned in the design chapter, cube textures only use half of the texture space. This can be improved by using a texture method that uses the entire texture space. To additionally reduce memory usage, the generation of data textures can be avoided by calculating the data value when required. For example, instead of using a heat and humidity texture, the base texture can be used to recalculate the heat and humidity when required.

Planets are still limited by size because of the floating point error[1]. By implementing a floating point origin system and adjusting the generation algorithms to cope with precision issues much larger scale planets are possible.

The branching system uses a hierarchy of game objects to implement the quadtree. An improvement to this system would be to implement the quadtree into a single mesh. This allows the quadtree to be defined fully in mesh space and this allows for stitching algorithms to avoid cracks[2] into the surface. This makes the system more data-oriented which allows the GPU to handle a greater part of the algorithm.

A cloud and atmosphere system would increase the immersion of the generated planets. These systems can hook into the procedural generation to use the climate data. Additional procedural properties can be exposed to the user so the cloud and atmosphere are adjustable by the tool as well.

[1] The floating point error is a precision problem that arises from the fact that float decimals reduce precision as the value becomes higher. If the precision is crucial information is lost and as a result glitching behavior occurs.

[2] When two meshes with different LOD are next to eachother, the surface of the highter LOD can contain height variations the lower LOD does not posses. This results in small gaps in the surface which are called cracks. Stitching is a way of closing those gaps.

6.2 Future research

The proposed tool can be used to implement different procedural algorithms. As visualized in figure 5.10, the procedural generation has well defined inputs (blue) and outputs (red). Different planetary procedural algorithms can be tested by designing a new system that uses the same input/output configuration.

This thesis uses GPU-based algorithms for improving performance. This decision is made based on theoretical knowledge. As a GPU is highly parallel in nature, performing highly parallel algorithms should theoretically be more performant on a GPU. Using this tool empirical analysis of both CPU and GPU-based procedural generation algorithms can be made, similar to the work performed by Vitacion Ryan and Liu Li. 2.1

In the Orbis tool support for runtime alterations of the terrain is provided as discussed in section 2.2.2. Implementing this feature on top of the current design introduces a new research goal. Expanding the planetary terrain design to keep track of changes to the terrain requires a system that works in combination with the procedural generation algorithm.

The current procedural algorithm contains a basic climate system and vegetation placing capabilities. This can be further expanded by implementing additional features like terrain erosion, river generation, dynamic climates, tectonic simulated continents, and so on. All these features can be exposed by the GUI.

7 CONCLUSION

The goal of this thesis was to create a planetary terrain system that handles highly detailed terrain in combination with a procedural generation system that handles multiple environmental situations. Based on this goal, a set of challenges was defined which would shape the design of the proposed tool.

A decision to create the tool as a Unity package for the Unity engine was made. This resulted in the project being developed using the C# language. This also allowed the tool to be distributed as a GitHub repository using the following url to the final implementation of the thesis tool. `https://github.com/DouweRavers/ThesisProject-ThePlanetEngine`

The highly detailed terrain introduced challenges regarding visualization and data configuration. As a result, the tool is provided with three systems to cover different detail requirements: a level of detail system that covers low detail requirements, a quadtree system to cover medium detail requirements, and a built-in Unity terrain system for high detail requirements. In order to make these systems compatible with the procedural generation algorithm, a methodology of generating base shapes in mesh and texture format is proposed. These basic shapes go through the same procedural transformations both in mesh space and in texture space. This prevents mismatching between the mesh surface and surface texture. Also, this allows for a single procedural definition for both texture and mesh generation.

The procedural generation algorithm is a parameterized system that makes use of GPU processing to address performance challenges. The algorithm is designed to be fully deterministic as the configuration of the planet should remain predictable. The generation process also provides multiple environmental situations as was stated in the goal. These situations are allocated by a climate simulation over the surface of the planet.

The deterministic procedural properties are exposed by using a GUI inside the Unity editor. This allows a user to design a planet by altering its properties. External textures and models can be used to assign ground textures, foliage textures, and tree models. The changes made to the procedural properties are previewed using a simplified planet inside the editor. This preview enables the planet to display the final result or highlight current changes.

BIBLIOGRAPHY

[1] (2013). Unite 13 lecture: How hard can rocket science be anyway?

[2] Ackroyd, L. (2021). Space engineers: Everything you need to know about planets.

[3] Developments, F. (2022). Explore.

[4] Digital, D. (2019). Estimated media revenue worldwide in 2020, by category.

[5] d'Oliveira, R. B. D. and Jr., A. L. A. (2018). Procedural planetary multi-resolution terrain generation for games. *CoRR*, abs/1803.04612.

[6] Games, B. (2022). starfield gameplay reveal.

[7] games, H. (2016). No man's sky press kit.

[8] Hall, S. B. (2020). Empty stadiums and online streaming: how coronavirus is affecting the media industry. *weforum*.

[9] Heok, T. K. and Daman, D. (2004). A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.*, pages 70–75.

[10] Imperium, C. (2022). Star citizen.

[11] javidx9 (2018). Code-it-yourself! 3d graphics engine.

[12] Magazine, R. (2016). Pixels and voxels, the long answer.

[13] Smelik, R. M., De Kraker, K. J., Tutenel, T., Bidarra, R., and Groenewegen, S. A. (2009). A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, volume 2009, pages 25–34. sn.

[14] Software, T.-T. I. (2022). Kerbal space program.

[15] space program wiki, K. (2021). *Kerbol System*. Kerbal space program wiki.

[16] Technologies, U. (2022). Performance by default.

[17] technologies, U. (2022). *Terrain*. Unity technologies.

[18] Technologies, U. (2022a). *Unity Manual: Compute shaders*. Unity Technologies.

[19] Technologies, U. (2022b). *Unity Manual: Coroutines*. Unity Technologies.

[20] Technologies, U. (2022c). *Unity Manual: Level of Detail (LOD) for meshes*. Unity Technologies.

[21] Technologies, U. (2022d). *Unity Manual: Normal map (Bump mapping)*. Unity Technologies.

[22] Vitacion, R. J. and Liu, L. (2019). Procedural generation of 3d planetary-scale terrains. In *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 70–77.

[23] Vogt, B. (2016). Procedural terrain generation.

[24] W., D. (2022a). *Orbis docs: introduction*. Orbis terrains.

[25] W., D. (2022b). Unity asset store: Orbis - dots terrains.

[26] Wijman, T. (2020). The worldś 2.7 billion gamers will spend $159.3 billion on games in 2020. the market will surpass $200 billion by 2023. *Newzoo*.

[27] wiki, M. (2022). Minecraft wiki: Terrain features.

[28] Wilkes, C. (2022a). *Space Graphics Docs: Terrain*. Carlos Wilkes.

[29] Wilkes, C. (2022b). Unity asset store: Space graphics toolkit.

FACULTY OF ENGINEERING TECHNOLOGY
CAMPUS GROUP T LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, Belgium
tel. +32 16 30 10 30
fax +32 16 30 10 40
fet.groupt@kuleuven.be
www.iiw.kuleuven.be