BIRKBECK, UNIVERSITY OF LONDON
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

MSc Computer Science

Project Proposal

Game With Procedurally Generated Content in Unity 3D

Supervisor: Keith Mannock
Author: Christian Radev

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text.I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

April 2017

Abstract

This project proposal outlines my background research in the area of graphics, computer games and algorithmic generation of 3D worlds and game objects. The purpose of which is to create a game with primarily procedurally generated content such as fully featured and realistic planets, solar system, characters, textures and more. The latest developments in the field clearly showed the impressive capabilities of procedural generation and its many benefits as compared to ordinary game development. The possibilities it offers are endless and requires much less resources to produce big projects with incredible results, than any other approach. My work so far shows how efficient and productive certain algorithms can be in generating heightmaps which are used to deform spherical mesh to resemble planet surface. This approach has endless variations of planetary detail and can be applied to generating the rest of the game content that is upcoming. It clearly shows that with minimal resources which are myself as a programmer, game engine (Unity 3D) and algorithms one can create a lot of game content which would normally take a big team of artists and game designers. By the end of this project it is expected that a fully featured procedurally generated game will be accomplished. The game itself is first-person role-playing in a fantasy theme, with the goal of exploring planets, gaining experience and advancing your character through quests and interaction with non-player characters. The results will produce a fun playable game with the option of being constantly expanded, simply by code and minimal to none manual content creation.

Contents

List of Figures

Abbreviations

PCG: **P**rocedural **C**ontent **G**eneration

NPC: **N**on - **P**layer **C**haracter

FPRPG: **F**irst **P**erson **R**ole-Playing **G**ame

Chapter 1

Introduction

The definition for procedural content generation, abbreviated as PCG, is 'the practice of creating game content algorithmically with limited or indirect user input' [1] . It is a piece of code that creates new or modifies 3D objects and fills the scene with them, thus creating a world in which the player can exist in. Such content can be, levels, maps, textures, models, weather, weapons, quests, etc. PCG is a different approach as compared to the usual manual creation of content [2]. In the latter, all the work is done by artists and game designers, meaning that the amount and diversity of content is limited by the number of people working and by their creativity. With algorithmic generation, on the other hand, all the repetitive tasks are taken care of by the computer and random diverse content is produced dynamically. One criticism of PCG is that the content can appear very repetitive as well with not having direct artistic control over the output, however this problem can be overcome by combining manual and procedural creation and carefully choosing which aspects of the game will be generated and which will be created by designers.

1.0.1 Benefits of PCG

The benefits of procedural generation are that much more content can be created in much shorter time, the content can be infinitely diverse as each generation can produce results with different seed for variables and it only takes computing power rather than people and creativity. There are well defined algorithms which are being used in the generation of any content available for a game, as it will be discussed later on. In terms of price PCG offers a much cheaper and faster solution, that is why many companies have started employing this method either entirely or in conjunction with manual design [2], [3]. In addition the data stored is much smaller because it only takes storing certain initial values as compared to all game objects as in manual creation.

1.0.2 Project Goals

The purpose of this project is to create a fantasy first-person role-playing game in the game engine Unity 3D, where all or most of the content is generated by code and not produced manually. This engine is free and offers a lot of features and flexibility for game development, where the main programming language that will be used is C#. The game will feature a space environment with different planets in a solar system. Each planet will be as detailed and realistic as possible, with all elements that a planet includes such as diverse landscapes, water, weather, atmosphere, clouds, etc. The player will explore and interact with the environment as well with non-player characters (NPCs), with the aim of completing quests and advancing your character with new abilities and items. The aim is not to implement every single aspect or model in the game myself (such as textures, materials and meshes), as some might be taken from third party libraries, but to develop a full-featured procedurally generated game and be considered as a whole. Moreover learn and gain skills in Unity 3D and C# as well with game development knowledge.

1.0.3 Organisation

This proposal will present my research in the field of PCG and the implementation results obtained to date. It will discuss some of the methods that will be used in developing all aspect of the game discussed previously and what steps I am planning to take in order to achieve it. At this point in time the chapters are organised in the following way:

- Chapter 2: Background Review
- Chapter 3: Current Results
- Chapter 4: Challenges and Schedule
- Chapter 5: Conclusions

1.1 Games Using PCG

Games that included some sort of procedural generation have existed for long time such as The Elder Scrolls II: Daggerfall released by Bethesda Softworks in 1996, which takes place in mostly generated world. One of the most notable games are Spore released by Maxis, Electronic Arts in 2008 where a planet generator is built into their game engine,

and allows creation of infinite diverse planets, which the player can visit and even edit [3].

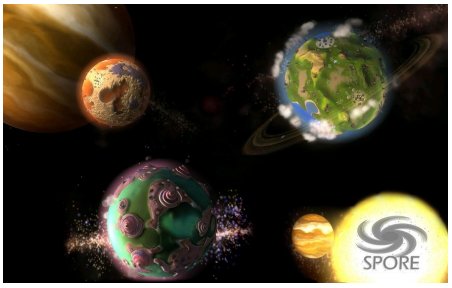

FIGURE 1.1: Spore, planets

FIGURE 1.2: Spore, terrain

And by far the best example and most recent game is No Man Sky released by Hello Games in 2016. The game is fully procedurally generated and features an open universe with 18 quintillion diverse planets, in solar systems and galaxies. The game include all sort of planet details such as ecosystem, flora, fauna and behavior patterns, artificial structures, spaceships and more. This game is incredible in terms of its development and features and the fact that everything is generated by code, makes it the prime motivation behind my project, but it is unrealistic to expect results of such scale.

FIGURE 1.3: No Man Sky, planet

FIGURE 1.4: No Man Sky, terrain

Chapter 2

Background Review

In this project the game that will be created is set in a space environment, which is a solar system with planets. The player inhabits any of this planets and can travel between them. All the other game aspects such as quests and character levelling, takes place on any of the planets. Therefore for this game it is crucial to create realistic life-like planets with different landscapes and detailed features.

2.1 Noise Generation

The most efficient way to create terrain-like structures is by using coherent noise. Such noise is generated by a coherent noise function which is function $f(x)$ such that any two values $f(x_0)$ and $f(x_1)$ are close together in terms of their values when x_0 and x_1 are close together, but do not correlate when they are far apart [4]. Coherent noise has three important properties as described in [5]:

1. 'Passing in the same input value will always return the same output value.
2. A small change in the input value will produce a small change in the output value.
3. A large change in the input value will produce a random change in the output value.'

The input parameter to such functions is called *seed*, because it is a small number that is used by the function to generate its output. Noise functions can be of any dimension and always produce a scalar value. In most cases a float number from -1 to 1. In graphics 2D or 3D noise functions are used to create textures, maps or 3D structures. An example

of what output a 1-dimensional coherent noise function might produce, plotted onto xy-plane is given on Figure 2.1.

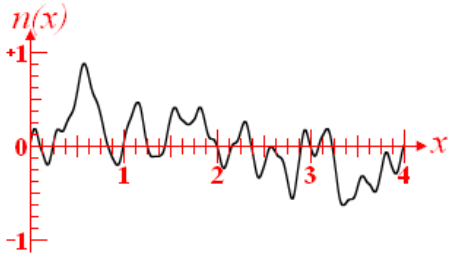

FIGURE 2.1: Coherent noise

One approach to construct a noise function is called *lattice noise* which works by calculating random numbers (lattice points) at integer coordinates. In order to find values at non-integer points the functions needs to be *interpolated* between the lattice points. A *gradient noise* is a variation of lattice, but instead of calculating random values, the function's *gradient* at different coordinates is calculated and the value at integer coordinates is set to 0 and then interpolated so that gradients match up [4] such as on Figure 2.1. The most common type of gradient noise is Perlin Noise, which is a type of coherent noise that is the 'sum of several coherent-noise functions of ever-increasing frequencies and ever-decreasing amplitudes' [5].
For the task of terrain generation, first a 2D texture needs to be generated. This texture is called Heightmap and it stores values from 0 to 1 represented as different shades of white and black where 0 is black and 1 white. These colour values correspond to vertical displacement or height from the ground, where white represents maximum height and black minimum. Figure 2.2 shows what a general Perlin noise heightmap would look like.

FIGURE 2.2: Perlin noise example

As you can see it resembles sine waves projected on 2D surface rather than terrain, this is because certain properties of the noise need to be defined to make it look how we want. These properties are:

- Octave - is one of the coherent noise function in series of such functions which are added together to form the Perlin noise, by default each octave has double the frequency of the previous [5]. The higher the octave the more crowded the image becomes, but also it lowers the amplitude - height.
- Frequency - determines how many changes occur along a unit of length, increasing the frequency increases the number of terrain features [5].
- Persistence - it determines how quickly the amplitude falls for each successive octave [5].
- Seed - it is some initial value depending on which different versions of the noise are generated. Same seed values always generate the same output.

By setting certain values for these properties one achieves heightmaps that resemble the one on Figure 2.3.

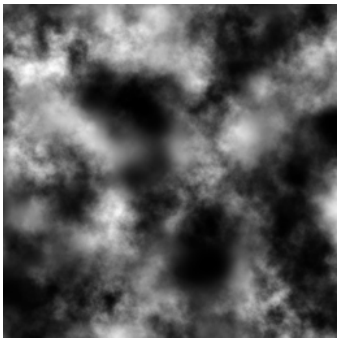

FIGURE 2.3: Heightmap example

An advantage of generating content using coherent noise is that only the seed needs to be stored and then the same content is always generated dynamically. On the other hand with any other approach of game development all of the data needs to be stored which sometimes is a lot and becomes problematic.

2.1.1 Different Noise Functions

Simply using Perlin noise function to generate terrain is not going to suffice. In order to create complex planetary landscape that have distinctive continents with different elevations and sea levels, it is necessary to combine different noise functions. All of these functions are implemented as generators, which given float coordinates they return a float number. There are many ways to combine noise functions such as simple arithmetic operations as addition and multiplication. There is also a constant function which

returns the same value irrespective of input coordinates. The following list describes the types of gradient noise generators according to [4]:

- Patterns - These generators create regular patterns and are used as base functions to be modified by other noise functions.
- Fractal noise - It combined several noise functions with increasing frequency and decreasing amplitudes. Perlin as a subtype of fractal noise [4].
- Voronoi diagrams - these functions don't produce noise but they are still random and produce cell-like patterns.
- Modification - these generators take other noise functions and modify them by applying another function on them such as map them to a curve.
- Displacement - it relies on source too, but it doesn't modify the noise functions, but rather their coordinate inputs. For example it can rotate or stretch noise patterns or add effects such as turbulence or swirls.
- Combination - this generator combines two or more source functions it can apply arithmetic operations to the sources or blend patterns by using a third function as a weight.

2.2 Generating Heightmaps

In Unity 3D libraries LibNoise [5] and CoherentNoise [4] that provide all of these noise generators are used. In my project in order to create more realistic and detailed heightmaps I combined the following functions.

1. Ridge Multifractal generator - it is a form of fractal noise which is used to create rough mountainous terrain Figure 2.4. This will define all the mountains in the map.

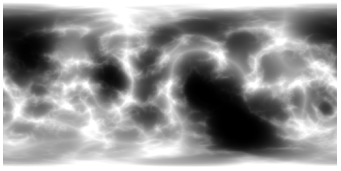

FIGURE 2.4: Ridge generator

2. Billow generator - it is used to define flat regions, because it generates cloud-like patterns as shown on Figure 2.5. However it needs to be scaled down to appear more flat as shown on Figure 2.6.

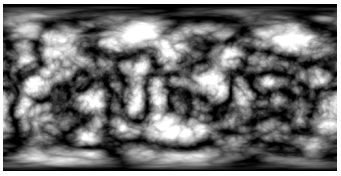

FIGURE 2.5: Billow generator

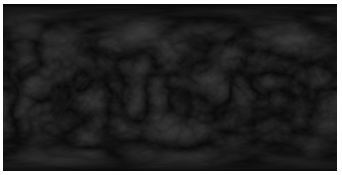

FIGURE 2.6: Scaled down

3. Perlin generator - it is used to define which regions will use the ridge generator and which the billow, thus defining the mountain and flat regions. In this case low values mean mountains and high values - flats Figure 2.7.

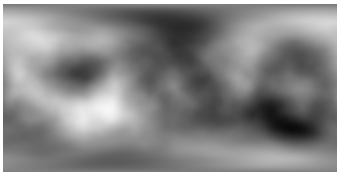

FIGURE 2.7: Perlin generator

4. Combination generator - it brings together all the noises ridge, billow and perlin into the one shown on Figure 2.8

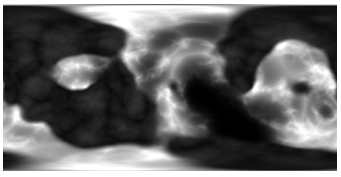

FIGURE 2.8: Combination generator

5. Turbulence - it is used to randomly perturb the values of the input coordinates before sending it to its source function. The final result is given on Figure 2.9

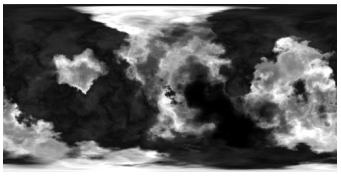

FIGURE 2.9: Turbulence generator

This result is quite good, however to be more detailed it needs to be rendered in high resolution, thus producing the heightmap on Figure 2.10. This map can also be coloured

depending on the height value with different colours for textures such as sand, hills, snow, etc. on Figure 2.11

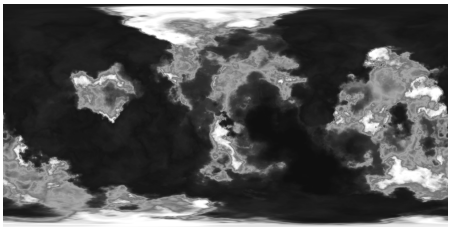

FIGURE 2.10: High definition heightmap

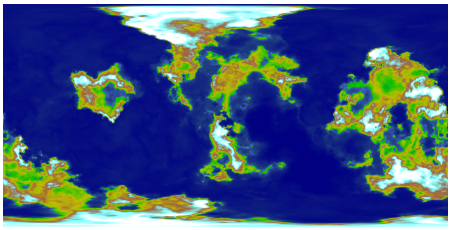

FIGURE 2.11: Same heightmap coloured

It is really important to note at this point that all maps generated so far are produced as a *Spherical Projection* which maps flat xy coordinates into polar (ϕ, θ) hence it appears on the images that the north and south regions are stretched. This stretching is necessary so that when this texture is applied to a sphere it appears completely seamless and wraps the sphere thoroughly. This is key when it comes to applying all heightmaps to spheres to create planetary surface.

Chapter 3

Current Results

At the current stage my game features a planet, the making of which is described below. As well with other features given as a list in the Other Features section.

3.1 Generating Planetary Landscape

Heightmaps were created in the previous chapter, in this one, they are applied to a mesh to create planetary surface. Thus the next step is creating a mesh resembling a planet. Each mesh in Unity is created from points called vertices and connecting three vertices forms a triangle, connecting multiple triangles creates a mesh. The more vertices and triangles there are in a mesh, the smoother and more accurate a shape would look. For a planet it makes sense to use a sphere as a mesh, however as it turns out Unity sphere does not have enough vertices, and also at the poles of that sphere an applied texture loses its seamlessness and gets disturbed. This is because it is implemented as a flat rectangular grid wrapped around a sphere and compressed into a single point at the poles [6]. This approach uses too many triangles at the poles, and ideally a sphere should have evenly distributed triangles.

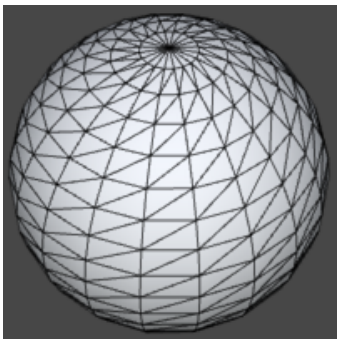

FIGURE 3.1: Sphere side view

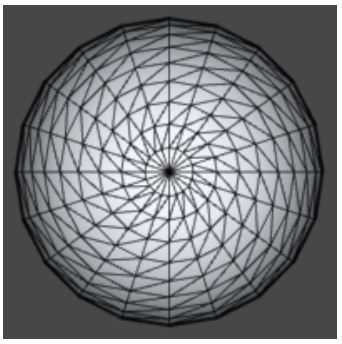

FIGURE 3.2: Sphere top view

3.1.1 Octahedron as a Sphere

From researching different ways to create accurate approximation of a sphere, the most suitable option was to start with a platonic solid and subdivide its faces until it resembles a sphere. An octahedron is a polyhedron having eight faces, twelve edges and six vertices. This mesh is the perfect candidate for a planet, and I use it to apply heightmaps to its surface. The model for the octahedron sphere is already implemented perfectly and taken from [6] as there is no reason to implement it myself, since the project goal is not mesh coding.

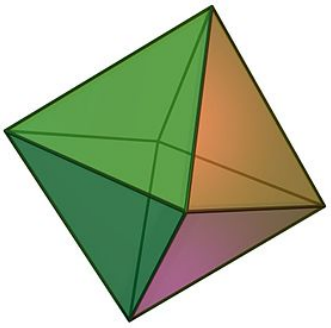

FIGURE 3.3: Octahedron

3.1.2 Why Use Octahedron

The regular Unity sphere does not have enough vertices to accurately reflect the heightmap. Moreover it is not as symmetrical as compared to a platonic solid. A tetrahedron (4 faces), cube (6 faces), octahedron (8 faces), dodecahedron (12 faces), or icosahedron (20 faces) are all approximations of a sphere because their vertices are all the same distance from their centre. If you split their faces into smaller ones, and push the new vertices to the same distance from its centre you end up with better approximations of a sphere proportionate to the number of subdivisions. The more faces you start with the better

distribution of triangles, therefore a tetrahedron is the worst option but the simplest to code and icosahedron the best option, but too complex to implement. Thus to balance between complexity and quality, octahedron falls in the middle and that is why I use it to approximate a sphere [6]. A comparison of a regular sphere and octahedron sphere is shown on Figure 3.4 and Figure 3.5, the octahedron clearly has more vertices.

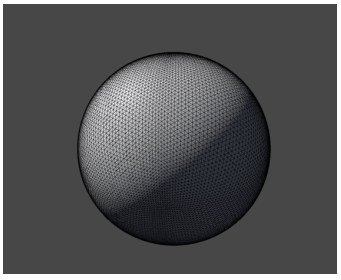

FIGURE 3.4: Octahedron sphere

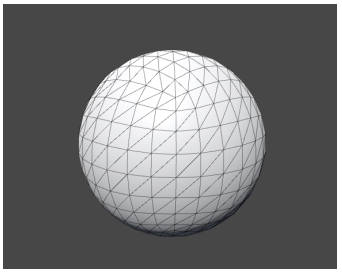

FIGURE 3.5: Regular Unity sphere

3.1.3 Applying the Heightmap

Having the planet mesh and heightmaps we can combine them to get the terrain surface. It is important to note here that simply applying a coherent 3D noise to a mesh only creates wave-like patterns with different frequency and amplitude as shown on Figure 3.6 as opposed to landscapes, this is because only a 2D projection with certain parameters and combination of noises produces realistic terrains, based on my experiments done with LibNoise.

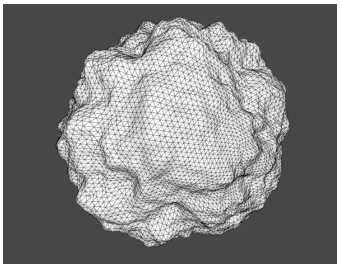

FIGURE 3.6: 3D Perlin Noise

The general algorithm to apply the heigthmap is the following:

1. Iterate over all vertices in the mesh - this way you can displace each vertex according to the map.

2. Having the 2D heightmap, get the grayscale value of each pixel corresponding to the XY- coordinates of each vertex (also called UV) - The reason to get the grayscale value is so that the output is a float in the [0, 1] range. The reason to

get these values corresponding to the UV mesh coordinates, is so that they match exactly the geometry of the mesh to that of the projection on the heightmap (both spherical).

3. Calculate the radius of the sphere from each vertex on the surface of the mesh - The reason for this is so that each vertex is displaced according to the spherical projection on the map as opposed to xyz displacement which leads to complete mismatch.

4. If the pixel value is smaller than certain ground value (experimentally chosen as 0.11), then subtract it from the radius - This allows for different elevation levels and lows to be visible as opposed to flat ground with only heights.

5. Else add the pixel value to the radius value - this adds heights from flat ground upwards.

6. Multiply the position of each vertex by the new calculated radius, thus deforming its height - by calculating the radius and multiplying it by the current vertex coordinates they become displaced according to the pixel value of the map.

7. Apply the new set of vertices to the mesh, thus replicate the heightmap into 3D terrain - applying the newly calculated coordinates of each vertex of the sphere creates a terrain reflecting the heightmap.

To elaborate on point 3. this part is key to make each vertex go either towards or outwards from the centre of the sphere to form highs and lows. This was a big challenge when first figuring out the algorithm, as I did not realise I need to use radius as opposed to xyz coordinates. The radius is calculated as follows:

$$v = vertex;$$

$$r = \sqrt{(v.x)^2 + (x.y)^2 + (v.z)^2}$$

If I only multiplied (or added) the pixel value by each vertex coordinates, I would get the inaccurate output as shown on Figure 3.7 and Figure 3.8:

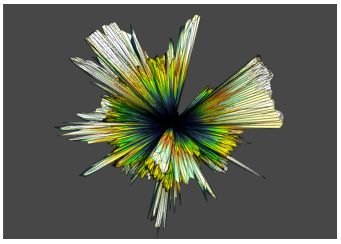

FIGURE 3.7: Multiplying xyz pixel value

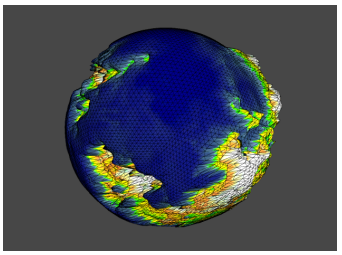

FIGURE 3.8: Adding xyz pixel value

By using the radius and applying the algorithm with the given heightmap from Figure 2.11, and using the coloured map as a material for the mesh too, the planet surface now resembles a landscape Figure 3.9

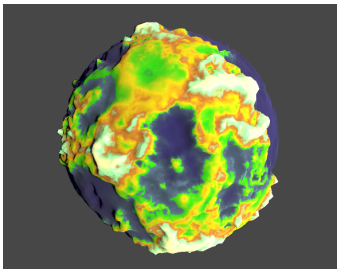

FIGURE 3.9: Planetary Terrain

There are clearly defined water regions, flat regions and mountains of different heights. So far the results are satisfactory, however the planet could be much more detailed, for example it could have rivers. The amount of details and diversity depends entirely on the generated heightmap. The challenges in generating highly detailed maps will be described in the Challenges section.

3.2 Water, Clouds and Atmosphere

Having a planet with terrain is only part of the whole package, in order to make it more realistic it needs adding seas and oceans, clouds and atmosphere. The way to implement water and atmosphere in Unity is by using Shaders, which are used to calculate rendering effects on the graphics hardware. There is a special language that is used to create shaders called Cg. Shaders are applied to materials and can create realistic rendering of different real-life materials. They are not considered as part of the PCG process in making this game, therefore either ready assets will be used taken from third parties, or my own shaders, the creation of which will be described in the Project Report. At this point the ones used are modified versions of default Unity shaders. The planet with

added water and atmosphere materials is shown on Figure 3.10 and with added clouds on Figure 3.11

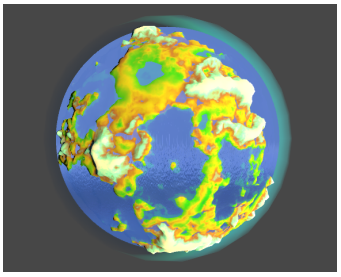

FIGURE 3.10: Water and atmosphere

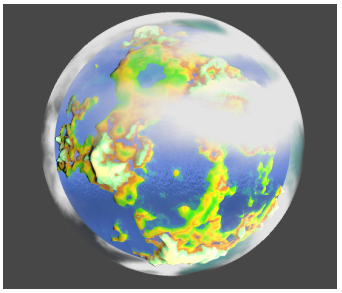

FIGURE 3.11: With added clouds

3.3 Other Features

The following list describes the rest of the features implemented to date and the overall result.

- Sun and Skybox - a realistic looking Sun is created that creates a day scene when shining at planet and a night scene when sets behind. In addition a skybox with glittering stars is used to be visible at night.
- Spherical Gravity - the default Unity gravity only works for flat terrains, in order to have spherical gravity one needs to apply downward force determined by the difference between the player position and the world position and align the rotation with the planet centre [7].
- First-Person Character Controller - a custom character controller is added that allows the player to move on the planet smoothly as well with jump.

The overall result for the player walking on the planet during day/night is shown on Figure 3.12 and Figure 3.13

FIGURE 3.12: Day scene

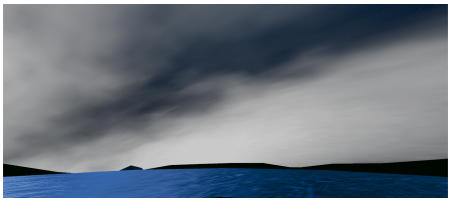

FIGURE 3.13: Night scene

Chapter 4

Challenges and Schedule

4.0.1 Challenges

This chapter describes the challenges that I face at the moment in terms of producing more content, and how I am planning to overcome them. The following list describes each one:

- Generation Performance - simply generating heightmaps takes a big amount of CPU power. In many cases if the details are too much Unity just crashes. In order to solve this I need to implement multithreading.
- Detailed Heightmaps - producing highly detailed and realistic maps would take a lot of coding and combining noise modules. This is where a lot of my focus will be when developing the game further. This issue is related to the first point.
- Threading in Unity - the default threading capabilities in Unity are fairly limited, and don't allow a lot of usability. In order to solve this I will have to use .NET threading features.
- Realistic Shaders - creating realistic atmospheric scattering or water is in the field of shaders, and it's quite complex. As I am planning to learn Unity's shaders, I will attempt to create as good as possible.
- Generating Miscellaneous - these include characters, vegetation, rocks and so on. It is a challenge creating these type of objects by code alone, what is more realistic is having ready models but using code to populate the world with them.
- Texturing - applying textures to the generated landscape and other models, is something I am currently exploring. This feature should be available as the project progresses.

4.0.2 Schedule

My plan to proceed with the project and which features to implement next, alongside with approximated dates for completion, is presented in the following table:

Date	Target
Jun 15 - Jun 21	Apply Dynamic Textures - apply textures to all the generated content such as landscapes. These textures could be either generated themselves or taken from open source libraries.
Jun 22 - Jul 2	Create detailed heightmaps - generate detailed realistic maps
Jul 3 - Jul 7	Adding miscellaneous - create or acquire models of vegetation, rocks, critters, etc. and populate the planet with them.
Jul 7 - Jul 10	Improve Shaders - add more realistic materials for every aspect of the planet.
Jul 11 - Jul 17	Create paths and artificial structures clustering them in cities and populate the planet.
Jul 18 - Jul 25	Adding NPCs - create or obtain NPCs and procedurally spread them in cities across the planet.
Jul 26 - Jul 28	Add more planets - create the solar system with realistic planet orbits revolving around the sun. Connect each planets with teleports for the player to travel among.
Jul 29 - Aug 3	Create GUI - create RPG style gui for the player to use and inspect items, abilities, experience etc.
Aug 4 - Aug 7	Enable NPC Interaction - create enemies for the player to kill and quest givers for quests. Implement leveling system.
Aug 7 - Aug 10	Create basic storyline - create quests that lead to other quests in other cities and planets.
Aug 11 - Aug 17	Create combat system - enable the character to fight NPCs either using melee weapons or by using abilities and spells.
Aug 18 - Aug 21	Add items - generate items such as weapons, armor, potions, etc. Which enemies drop and the player can use.
Aug 22 - Aug 26	Updates and Bug Fixes - add any additional features that need adding. Also fix any bugs or improve current feeatures before finishing and exporting the standalone game.
Aug 27 - 17 Sep	Write Report

Table 4.1: Timeline Table

Chapter 5

Conclusions

As described in previous sections the objective is to create a full featured standalone game, which employs primarily procedural generation for creation of content. It is unrealistic to aim for a big project like No Man Sky or Spore, instead create an interesting playable game with as many features as time allows. So far I have presented my background research in generating noise to create heightmaps. In addition describe how I achieved my current results of deforming the surface of a spherical mesh to resemble terrain, using the heightmaps. Moreover outline the extra features added to the planet such as water, atmosphere, clouds and character controller. The challenges encountered on the way and the dated plan for solutions was presented in the final chapter.
This project will be a good learning experience for me, not only because of gaining skills and experience in Unity 3D and C# but also good understanding of computer graphics and game development as a whole, which I find very interesting and plan to explore further either professionally or as a hobby.

Appendix A

An Appendix

In my original attempt and experimentation to create terrain and planet, I tried using the default Unity terrain utility. The terrain produced looked nice and I could generate smaller chunks of terrain and seamlessly connect them together thus improving performance, moreover it was easy to texture it [8]. Since these terrain chunks are flat, the idea was to arrange them in a sphere and then rotate them outward from the centre to form the planet surface. However as it turned out Unity does not allow rotation on terrain objects, thus this approach failed as illustrated on Figure A.1 and Figure A.1

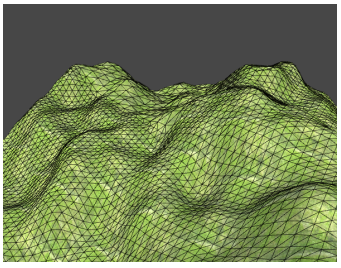

FIGURE A.1: Terrain chunk

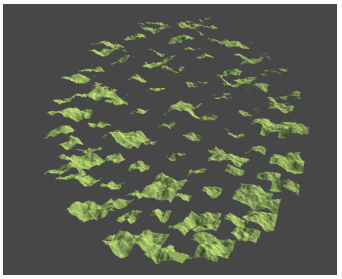

FIGURE A.2: Chunks sphere

Another thing I experimented with are fractal shapes as shown on Figure A.3 and similar objects might make it in the final game.

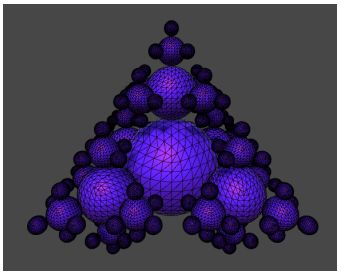

FIGURE A.3: Fractal shape

Bibliography

[1] Julian Togelius, Noor Shaker, and Mark J. Nelson. Introduction. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 1–15. Springer, 2016. URL http://pcgbook.com/wp-content/uploads/chapter01.pdf.

[2] Eoghan Carpenter. Introduction. In *Procedural Generation of Large Scale Gameworlds*, number 12, pages 1–15. September 2011. URL https://www.scss.tcd.ie/postgraduate/msciet/current/Dissertations/1011/CarpenterEoghan.pdf.

[3] Twan De Graaf. Procedural planets into detail. *Procedural Planets Into Detail*, (5): 1–5, May 2013. URL http://www.twandegraaf.nl/Art/Documents/Procedural%20planets%20into%20detail,%20Twan%20de%20Graaf%202012.pdf.

[4] Chaos Cult Games. Introduction. In *Coherent Noise*, number 7, pages 1–45. 21-Dec-2014. URL http://chaoscultgames.com/products/CN/CoherentNoiseManual.pdf.

[5] Jason Bevins. Libnoise: What is coherent noise? URL http://libnoise.sourceforge.net/coherentnoise/index.html. Online; accessed 10-April-2017.

[6] Jasper Flick. Creating an octahedron sphere in unity, Oct 27, 2014. URL http://www.binpress.com/tutorial/creating-an-octahedron-sphere/162. Online; accessed 10-April-2017.

[7] Ben Welsh. How to create spherical gravity in unity. URL https://benwelshscript.wordpress.com/2015/07/21/how-to-create-spherical-gravity-in-unity/. Online; accessed 10-April-2017.

[8] Code-Phi.com. Infinite terrain generation in unity 3d part 1. URL http://code-phi.com/infinite-terrain-generation-in-unity-3d/. Online; accessed 10-April-2017.