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 height maps 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 accomplish. 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. 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. 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.

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.

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 Work Plan
- 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.

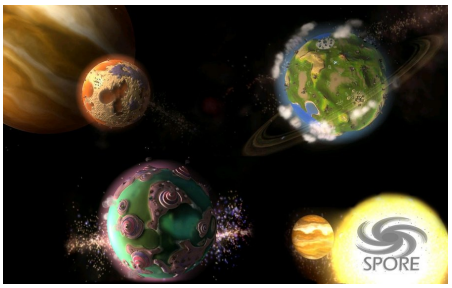

FIGURE 1.1: Spore, planets

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 to achieve similar results.

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.

2.0.1 Noise Generation

The most efficient way to create terrain-like structures is by using coherent noise. Such noise is generated by coherent noise functions and has three important properties as described in **??**

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."

Noise functions can be of any dimension and always produced 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. It is important to note that coherent noise is not random, but it is a pseudo-random smooth noise, meaning that the output values will be bound by certain mathematical procedure and change with accordance to its neighbouring values as opposed to random unrelated values. The following figures compare 1-dimensional coherent noise of frequency 2, with a non-coherent noise (both taken from **??**:

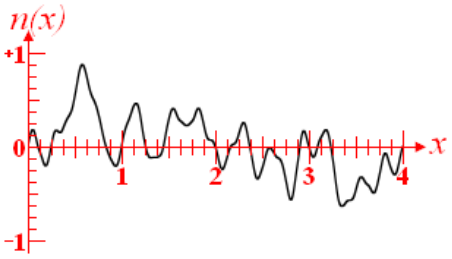

FIGURE 2.1: Coherent noise

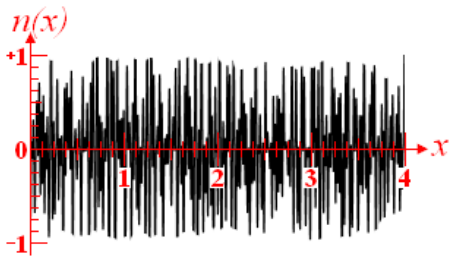

FIGURE 2.2: Non-coherent noise

The most common type of coherent 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 **??** 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 2.3 shows what a general Perlin noise heightmap would look like.

FIGURE 2.3: 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:

- Octaves is one of the coherent noise function in series of such function which are added together to form the Perlin noise, by default each octave has double the frequency of the previous. The higher the octave the more crowded the image becomes, but also it lowers the amplitude - height.

- Frequency it determines how many changes occur along a unit of length, increasing the frequency increases the number of terrain features.
- Persistence it determines how quickly the amplitude falls for each successive octave.
- Seed - it is some initial value depending on which different versions of the noise are generated

By setting certain values for these properties one achieves heightmaps that resemble the one on Figure 2.4.

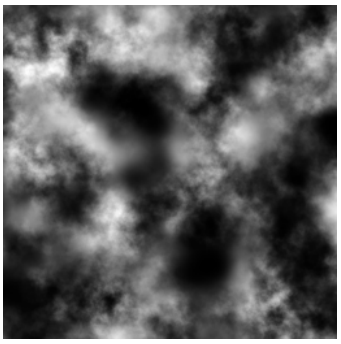

FIGURE 2.4: Heightmap example

In Unity 3D a library that provides noise functions is used for my project - called LibNoise **??** There are many modules and functions that provide different types of noise functions, such as fractal noise and billow noises. There are functions used to combine noises or change certain properties. By combining them and changing these properties one could create heightmaps resembling with different types of landscapes, as realistic as real life, an example of such heightmap is given on Figure 2.5 and instead of a grayscale image, we could colour the map depending on the value with different colours for textures such as sand, flats, snow, etc. on Figure 2.6

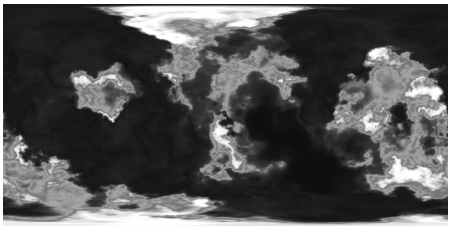

FIGURE 2.5: High definition heightmap

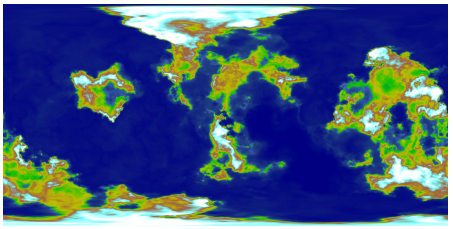

FIGURE 2.6: Same heightmap coloured

The functionalities of LibNoise and the different types of noises will be discussed in detail in the Project Report.

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 subsection.

3.0.1 Generating Planetary Landscape

At this point having the heightmaps it is time to apply them to a mesh so it deforms the surface and produces terrain. The next step is creating the mesh since the goal is to make planets. Each mesh in Unity is created from points called vertices and connecting three vertices forms triangle, connecting multiple triangles creates a mesh. The more vertices and triangles there are in a mesh, the more smooth and accurate a shape would look. For a planet it makes sense to use a sphere as a mesh, however as it turns out Unitys sphere doesnt have enough vertices, and also at the poles of a sphere an applied texture loses its seamlessness and gets disturbed. From researching different ways to create accurate approximation of a planet, the most suitable option was to use an Octahedron and create many subdivisions of it until it resembles a sphere. This approach allows for spheres with many times bigger number of vertices as compared to ordinary sphere. The model for the octahedron sphere is already implemented perfectly and taken from **??**s there is no reason to implement it myself, since the project goal is not mesh coding. A comparison of a sphere and octahedron sphere is shown on Figure 3.1 and Figure 3.2, the octahedron clearly has more vertices.

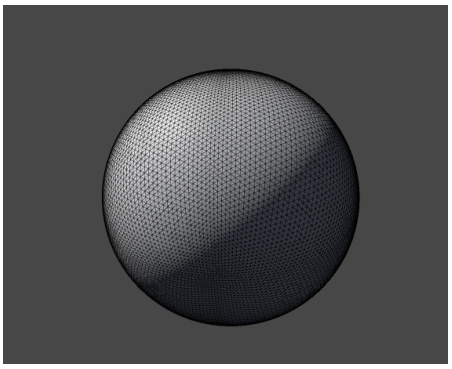

FIGURE 3.1: Octahedron sphere

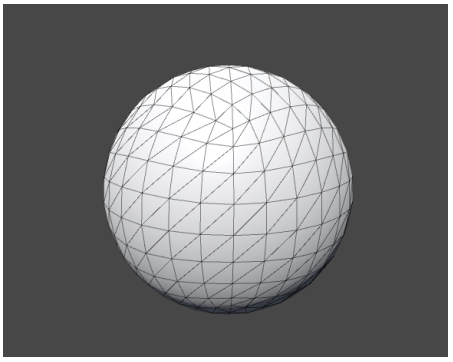

FIGURE 3.2: Regular Unity sphere

Having the planet mesh and the heightmaps, it is time to combine them and create the planetary 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 3.3 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.

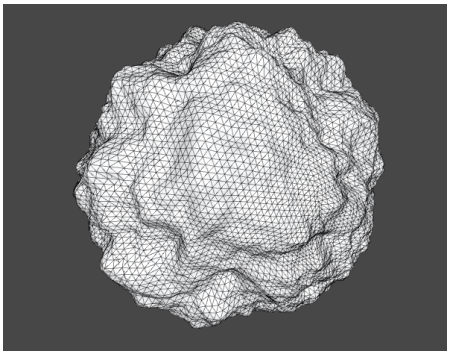

FIGURE 3.3: 3D Perlin Noise

The general algorithm to apply the heigthmap is the following:

- Iterate over all vertices in the mesh
- Having the 2D heightmap, get the grayscale value of each pixel corresponding to the XY- coordinates of each vertex (also called UV). It returns values from 0 to 1.
- Calculate the radius of the sphere from each vertex on the surface of the mesh.
- If the pixel value is smaller than certain ground value, then subtract it from the radius.
- Else add the pixel value to the radius value.

– Multiply the position of each vertex by the new calculated radius, thus deforming its height.

– Apply the new set of vertices to the mesh, thus replicated the heightmap into 3D terrain.

By applying this procedure with the given heightmap from Figure 2.6, and using the coloured map as a material for the mesh too, the planet surface now resembles a landscape Figure 3.4

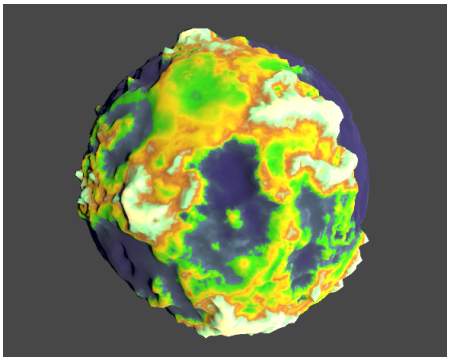

FIGURE 3.4: Planetary Terrain

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.

3.0.2 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 3.5 and with added clouds on Figure 3.6

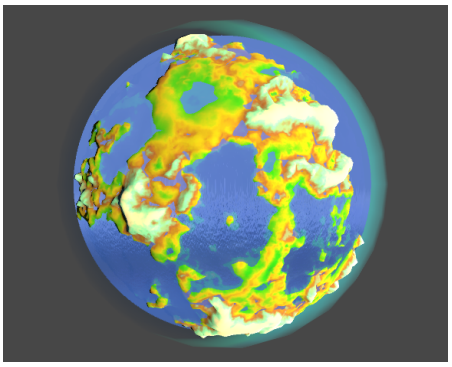

FIGURE 3.5: Water and atmosphere

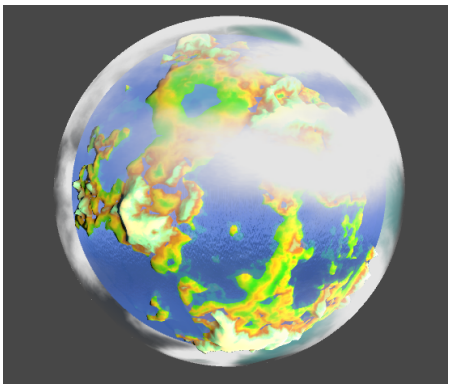

FIGURE 3.6: With added clouds

3.0.3 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.
- 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 show on Figure 3.7 and Figure 3.8

FIGURE 3.7: Day scene

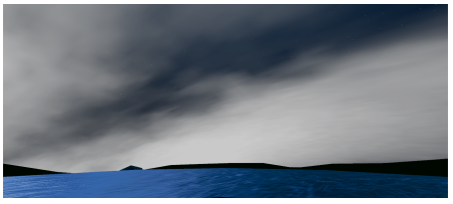

FIGURE 3.8: Night scene

Chapter 4

Challenges and Work Plan

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 Work Plan

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

Appendix A

An Appendix

Bibliography