Abstract

This report explores each of the stages involved in creating a system that randomly generates planetary systems. Each problem is deconstructed and explained in the context of how I chose to solve the problem. The chapters of this report discuss a different area of procedural generation in relation to random planets. The first chapter considers real world physics formulae and how they can be used to produce more realistic planets. The other chapters are focused on various aspects of 3D graphics and game design such as noise generation, texturing and 3D modelling. Certain key concerns of game design are considered throughout these chapters, such as performance and graphical style.
I would like to thank my project supervisor Marie Redmond for her early advice on how to approach the development of an interactive experience and ensuring that I had a concrete plan for the desired user experience. This advice helped to shape my vision for the final product. I would like to thank Ken Perlin for his work in graphics and animation. His work and his stance on open-source code have influenced almost every computer graphics product in development today, including my own. I would also like to thank the countless users of the Unity Answers forum for their wealth of knowledge on all aspects of game development within Unity. Finally I would like to thank my friends and family for their continued suggestions on how to develop and improve my project.
# Table of Contents

Abstract ................................................................................................................. 1
Acknowledgements ................................................................................................. 2
Introduction .............................................................................................................. 5
Background .............................................................................................................. 6
  Motivation ............................................................................................................. 6
  Related Work ........................................................................................................ 6
  Spore ..................................................................................................................... 6
  Planetary Annihilation .......................................................................................... 7
Planet Properties ..................................................................................................... 9
  Variables ............................................................................................................... 9
  Calculating Physical Properties ............................................................................ 9
Random Terrain ...................................................................................................... 13
  Terrain Generation for Two Dimensions ............................................................... 13
  Generating 3D Terrain ........................................................................................ 14
  Mapping 3D Noise to a Sphere ............................................................................. 15
  Noise for Gas Giants ............................................................................................ 16
Texturing .................................................................................................................. 17
  Colour Gradients ................................................................................................. 17
  The Sea ............................................................................................................... 19
  The Atmosphere .................................................................................................. 20
  The Moons .......................................................................................................... 21
  The Stars .............................................................................................................. 22
Planet Mesh ............................................................................................................ 23
  Mesh Properties .................................................................................................. 23
  Moulding the Mesh .............................................................................................. 25
User Experience ....................................................................................................... 26
  Camera Controls ................................................................................................. 26
  User Interface ...................................................................................................... 26
  Simulation ............................................................................................................ 27
Evaluation ................................................................................................................. 28
  Conclusion .......................................................................................................... 28
  Future Work ......................................................................................................... 29
  Learning Outcomes ............................................................................................. 31
Appendix .................................................................................................................. 32
Introduction

The aim of this project was to create an educational videogame in the *Unity* game development environment. My goal was to create a game with a graphical style that could maintain the interest of a younger user. Simultaneously I wanted the game to educate them about celestial bodies and inspire an interest in further study of any topic related to astronomy and cosmology. The game will generate imaginary planets that are influenced by realistic physics. The user will be able to observe the visual appearance of these planets. The player can also look at the physical properties of these planets to understand how these properties have affected the appearance of the planet. For example planets that are spawned closer to their star will be hotter than those that are much further away. The immediate appearance of the planet should reflect this physical property.

This report explains each of the challenges that I encountered while attempting to design a system that can produce these random planets. Each chapter relates to a different area of game development in this context.

**Chapter 2 Background** – This chapter looks at current state-of-the-art planet generation in two commercial videogames. I discuss the merits of these games and which of their features best fits the goals for my own project.

**Chapter 3 Planet Properties** - In this chapter I discusses is how closely these random solar systems obey the laws of physics and how some physics formulae can be altered to allow these planets to be presented in a more engaging way.

**Chapter 4 Random Terrain** – Here I focus on techniques for generating random terrain. The problem is initially treated in a two dimensional scenario to explain some of the core aspects of terrain generation. The issue is then brought in to three dimensions in order to generate terrain for the spherical surface of a planet.

**Chapter 5 Texturing** - This chapter deals with creating terrain textures based on a height map. Of particular interest in this chapter is discussing how to ensure that textures will always appear varied and eye-catching and how the height of a terrain map affects the texture. This chapter also deals with creating textures for a planet’s sea and clouds, and also creating textures for moons and stars.

**Chapter 6 Planet Mesh** – Here I discuss what kind of 3D mesh should be used. The mesh has two requirements; it must be easily mapped for texturing and should be structured in such a way that it can be deformed smoothly.

**Chapter 7 User Experience** - This chapter deals with some of the miscellaneous features of a space exploration game that make it more satisfying for the player.
Background

Motivation

The motivation behind this project was to create a system that can randomly generate planets that can be used in an educational tool. I want to inspire an intrigue into the study of astronomy and cosmology in a younger demographic. I intend for this project to be able to generate planets that are tied to real-world properties of planets while being highly varied and eye-catching to a younger audience. While the planets created possess properties similar to those of the real world I wanted to take some creative liberty to make them more presentable on screen. Two major ways I do this is by making planets orbit much closer together and with a more presentable scale. I also ensure that planets spawn with a rich variety of colours. If someone were to take a picture of our solar system using an average consumer camera with all of our planets in frame each planet would be smaller than a single pixel. The planets in my project have a significantly smaller scale so that they can all be seen in a single frame with ease. Similarly, almost all real world rock planets have a dark, dull colour palette. The planets in this project, however, have distinct colours.

Related Work

Spore

A video game that took a similar approach to presenting planets in an eye-catching way is Spore [1]. It was developed by Maxis and published by Electronic Arts in 2008. In Spore the player assumes the role of a species evolving through its various stages of life. It starts from a cellular organism and evolves all the way up to a spacefaring empire. At this final stage of the game the player can explore a nearly infinite number of random planets. The developers of Spore published a short paper on their approach to random planet generation which is freely available online [2]. These planets have two key properties: atmosphere density and temperature. These properties affect the look of the planet and its ability to house life. Cold planets are covered in ice while hot planets are spew pools of lava on their surface. Planets with a thick atmosphere are densely covered in clouds while those with no atmosphere are barren rock planets. Pictured below are some examples of planets with these varying properties as well as multiple examples of habitable planets.
In *Spore* these two planet properties are entirely random and independent of the rest of the planet’s properties. They can be easily changed by the player if they wish by spending in-game resources. However in the real world the temperature and atmosphere of a planet are dependent upon other properties of the planet and its star. My aim with this project is to make these two key properties depend on the other properties of the planet. Moreover if a player wanted to alter these key properties they would have to do so by altering various aspects of the planet. For example if the player wishes to heat up the planet they could do so in a variety of ways. They could:

- Increase the brightness of the star
- Move the planet closer to the star
- Try to increase the density of the atmosphere and therefore the greenhouse effect on the planet.

These choices would have other effects on the planet that the player would have to account for and observe. Where *Spore* aimed to teach its players about hypothetical forms of life on other planets I aim to teach my users about the various kinds of planets that could exist in our universe and how the physical properties of the planet affect it.

**Planetary Annihilation**

There is a game currently in development that has a stronger focus on altering the physical properties of planets. The name of this game is *Planetary Annihilation* [3]. It is being developed by *Uber Entertainment* and was crowd funded on the website *Kickstarter* in September of 2012 [4]. It is a real-time-strategy game where multiple players compete to gain control of an entire solar system. Each player starts on a separate planet. These planets are randomly generated with key properties such as size, temperature and atmosphere. A key feature of the game is that players can alter planets and even redirect small planets and asteroids to use them as a weapon and send them on a collision course with an opponent’s planet. Players can even create their own man-made planets for the same purpose. Player’s need to be aware of a planet’s mass and
orbital velocity to determine how they can safely alter its course. Players must also be vigilant of the temperature and atmosphere of each planet to ensure that their troops can safely survive on that planet. Below is an early screenshot of Planetary Annihilation showing a starting planet in the centre with a moon to the right and a man-made planet to the upper-left. Also shown is the orbital path of these bodies and the route that ships are taking between the planet and its moons.

This game improves on Spore’s shortcomings by simulating the physics of the planets in more detail. However it falls behind by offering only a handful of planet types. The developers are constraining the variety of planets because their primary focus is to create a strategy game in which player’s seek out planets for certain resources. Each resource will always be more plentiful on particular types of planets. Players should be able to immediately identify at a glance what kind of resources might be found on the planet they are looking at. My aim is to mix the style of variation found in Spore with some of the physics simulations that take place in Planetary Annihilation.
Planet Properties

Variables

The physics requirement in this game are not too high. Each planet that is generated has a small list of physical properties. Some of these properties are randomised at the very start of the planet’s life. These properties and are treated as ‘primary properties’. The remaining ‘secondary properties’ are calculated based on the primary properties and other properties of the solar system.

The primary properties of a planet are:

1) Size  
2) Density  
3) Spin Speed  
4) Distance from the star

The density of a planet determines whether it is a gas giant or a solid rock planet. Rock planets have several more properties than gas giants such as atmosphere and the height of the sea on the planet.

The secondary properties calculated are:

1) Mass  
2) Orbital velocity  
3) Surface gravity  
4) Atmosphere  
5) Temperature  
6) Sea Height

Calculating Physical Properties

When the game generates a new solar system a star is placed in the centre of the solar system. The first planet that is spawned is placed a short, randomized distance from the star. A random number of moons are given to each planet. The space around the planet that is required by these moons is also taken into account. This is used to ensure that the moons of two different planets will not cross paths. Each subsequent planet is spawned another small random distance from the previous planet along with all of its moons. When a planet first spawns its primary properties are randomly generated.

The size of a planet ranges from 0.8 to 4.0. This is an arbitrary scale used in the game. Density ranges from 0.5 to 8.0. These units are in grams per cubic centimetre (g/cm³). It is treated as the mean density of the entire planet. A density below 3.0 g/cm³ will result in the planet being a gas giant in this simulation. I based this threshold on the mean density of planets in our own solar system. Jupiter is the most dense gas giant in our solar system and has a mean density of 1.3 g/cm³. Mars is considered to have quite a low density for a rock planet with a density of 4 g/cm³.
Based on these planets I felt that a density of 3g/cm³ was a reasonable threshold to use. Spin speed is randomly assigned in a range of 0.3 to 3.0 degrees per second.

From these properties the game calculates the secondary properties. The mass is calculated by using the density and size of the planet. The size variable represents the radius of the planet, so this must be converted to volume. This volume can then be multiplied by the density to find the mass. The formula for the mass is:

\[ \text{mass} = \text{density} \times \left( \frac{4}{3} \pi r^3 \right) \]

The angular velocity of a planet's orbit is found using the following formula:

\[ \omega = \sqrt{\frac{G(M + m)}{d^3}} \]

\(M\) and \(m\) are the masses of the star and planet respectively. The distance between the two bodies is represented as \(d\). The variable \(G\) is the gravitational constant. In the real world this value of this constant is \(6.67 \times 10^{-11}\). In this game however I used a value of 20. This allows planets to orbit much faster. This is essential in this game so that users can observe the gradual movement of the planets in real time.

The gravity at the surface of the planet is calculated to be offered as information to the user. It also affects the thickness of the atmosphere on the planet.

\[ g = \frac{GM}{r^2} \]

Where \(G\) was 20 for the orbital velocity in this case I use a value of 0.2. This constant yields values similar to that of real world values with planets having a gravity of just one or two digits.

The atmosphere that a planet might be able to hold is not an easily defined value. I decided to create my own function to calculate the atmosphere on a planet. This function considers two things. First, the stronger the gravity of a planet the better it will be able to retain a layer of gas around it. Second, even if a planet has a strong gravitational pull that does not guarantee that it will hold an atmosphere. Therefore the function is multiplied by a random value ranging from 0.1 to 1.0. The function is:

\[ \text{atmosphere} = \sqrt[4]{g} \times \text{rand}(0.1, 1.0) \]
The graph below represents the function without the random element. It should give a sense of maximum atmosphere a planet can have based on its surface gravity. An atmosphere of 1 is considered to be the same as the atmosphere on Earth.

\[
\text{Graph for } \sqrt[4]{g}
\]

The temperature of the planet is calculated based on three things: the distance from the star, the size of the star and the atmosphere. The atmosphere affects the temperature due to the greenhouse effect. The greenhouse effect of an atmosphere is difficult to equate as it depends upon the chemical composition of the atmosphere. For this project the greenhouse effect is applied quite simply by just adding 0.5 to the atmosphere and multiplying the temperature by this number. Here is the function:

\[
temperature = (\text{atmosphere} + 0.5) \sqrt[4]{\frac{\text{sunSize}}{10\pi\sigma d^2}}
\]

This function is closely based on the real world formula for the effective temperature of a body. This function uses the constant \( \sigma \), the Stefan-Boltzmann constant. This constant is concerned with the heat radiated from a black body and is equal to \(5.67 \times 10^{-8}\). The real world effective temperature uses a value of 16 whereas I have used 10. The real function uses the fourth root of the fraction where I have only used the squared root, this yields more desirable results for my game. This results of this function are treated as degrees kelvin.

While multiplying this by the atmosphere is a very simplified method it works quite well for this project. It offers a large variety of possibilities where some planets further away from a star have
a slightly higher temperature than some planets closer to the star. This is in fact observable in our own solar system. Venus has the thickest atmosphere of any planet in our solar system. Despite Mercury being closer to the sun Venus is in fact hotter due to its thick atmosphere.

Finally, the sea height of the planet is dependent upon the temperature and the atmosphere. Once again this is a function that I have created myself without a strict basis in real life physics. The aim of this function is to have a sea only appear on planets that are not too hot. A hot planet will have its water evaporate and colder planets will be able to retain a frozen sea. The chance of a planet having a sea in the first place depends on the thickness of the atmosphere. Here is the function for calculating the height of a planet’s sea:

\[
Sea \, Height = atmosphere \times \left( 0.5 \times \left(1 - \frac{\text{temperature} - 300}{400}\right) \right)^3
\]
Random Terrain

Terrain Generation for Two Dimensions

One of the most popular methods for generating random terrain textures is by combining multiple textures of randomly generated noise at different scales. These noise maps are generated by filling a two dimensional array with random values. This noise is transferred to a texture at a desired scale, with the borders between noise values being blended. This is commonly known as ‘salt and pepper noise’. Combining several maps at different scales creates a composite noise map called ‘fractal noise’. Figure 1 below is a demonstration of how 6 noise maps can be combined to make a fractal noise map.

![Fractal Noise Maps](http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)

This map is used to represent the height of the terrain at each point. We can turn this into a believable terrain map by applying different colours to certain heights on the map. In the case of this project black pixels have a value of 0.0 and white pixels have a value of 1.0. A simple way to create a texture map would be to take all the pixels with a value less than 0.5 and turn them blue. All other pixels are turned green. This will create a map of green islands in a blue sea, similar to our own Earth. Figure 2 below is an example of how that might look:
This map goes one step further by shading the map based on the height. Mountains are made slightly brighter towards their peak. The sea is made darker in its deeper areas. Maintaining this shading helps to give texture maps a high level of contrast. Particularly at coastlines. For this project I wanted to guarantee that there would be a large amount of contrast and ensure that users could easily distinguish what is supposed to be land and what is supposed to be the sea. This is easy to recognise when looking at maps using an Earth-like colour palette but it requires greater care when it comes to randomly generated colour palettes. Areas of sea close to the coastline are always drawn slightly brighter than the coastline and the rest of the sea.

Generating 3D Terrain

In order to generate terrain on a spherical surface we need to generate noise in three dimensions. Moving into three dimensions for texture generation presents several issues. For one, mapping a randomly generated texture similar to the one above onto a sphere causes significant warping. My solution to this is to instead consider a three dimensional noise cloud. Within this cloud we can sample noise values along the surface of an imaginary sphere. The texture maps generated in the previous chapter are 128x128 pixels, which is 16,384 total values that need to be computed. We can easily create random noise in three dimension and interpolate in three dimensions if we wish.

However if we were to create a cubic 3D array of noise 128 pixels wide it would require an array of over 2 million values. My aim is to generate maps that are an even higher resolution than this so that I can create high fidelity planets. Generating a full cubic map is a very wasteful approach because only a small percent of the values will be used. The solution to this is to only generate this 3D noise along the surface of the sphere. This doesn’t entirely solve the problem because all of the initial random values near the sphere’s surface must still be stored. At high resolutions and with a large number of different noise scales this will still require that a large amount of memory.

An alternative way of generating noise is a method known as ‘Simplex Nosie’. This a noise generation algorithm developed by Ken Perlin. His code for this noise generation algorithm is freely available on his website [5]. It is an improvement of his previous noise generation algorithm that he called ‘Perlin Noise’ which Perlin discussed in a paper he published in 2002 [6]. This method of noise generation utilises gradients at each integer grid point similar to that of method discussed in the previous chapter. These random gradients are

http://www.angelcode.com/dev/perlin/perlin.html
created from a hash of the integer grid points. This hash is fed into a permutations table of just 256 different integers. This table contains every integer from 0 to 256 in just a single location. This 256 long table is the only memory requirement for this noise generation method. As well as having a fraction of the memory requirements, the noise can be sampled at any decimal position within a set of integer grid points. Moreover it offers the ability to resample the noise at a higher resolution at a later time. This means that the game could initially generate the planets at a low resolution. As the game is running higher resolution texture can be generated as needed. If implemented appropriately this could allow the game to have a reduced loading time on start-up and allow for infinitely detailed planets that are rendered procedurally.

**Mapping 3D Noise to a Sphere**

The planets in this game are set up to receive 2D textures that are then wrapped onto the surface of the sphere. This requires that the noise is sampled in 3D space and then mapped to a 2D texture. This texture will be a rectangle with a width twice as long as its height. This shape will be preferable because the texture’s width will be mapped to the circumference of the sphere, and therefore the height will only map half the circumference from pole to pole. My approach to this is to loop through each pixel on the texture. Each pixel point has the coordinates U and V. These UV coordinates are then mapped to their 3D position on a sphere in the form of XYZ. Finally these XYZ coordinates are fed into the Simplex Noise generation function. Figure 4 is an illustration of how these texture pixels are mapped to the surface of a sphere.

![Mapping 3D Noise to a Sphere](image)

The functions for finding the X, Y and Z coordinates are [7]:

\[
 x = \cos \varphi \cdot \sin \theta \\
 y = \sin \varphi \cdot \sin \theta \\
 z = \cos \theta
\]

Where:

\[
 \varphi = \frac{U \cdot \pi}{\text{textureWidth}} \\
 \theta = \frac{V \cdot \pi}{\text{textureHeight}}
\]
These values $\varphi$ and $\theta$ are the polar and azimuth angles, the angular measurements used in spherical coordinate systems. Using the same method of combining several noise maps of difference scales as seen in the previous chapter yields similar results. The difference now is that the texture appears warped towards the top and bottom. This counter-acts the warping that results from mapping this rectangular texture to a sphere.

This noise will be a random float with a range that is very difficult to predict. For the purpose of this project I need the noise to lie within a range of 0 to 1. Rather than simply clamping the noise within this range I want to scale it. This will ensure that none of the height is lost and that the full spectrum of height will be used.

**Noise for Gas Giants**

The noise map for gas giants needs to be calculated slightly different. Gas giants can be seen to have waves and stripes of different colours running through them. These ripples are caused by the spinning of the planet. We can recreate this effect by applying one extra calculation to our noise before inserting it into the height map. If we feed the fractal noise value into a Sine function we can create noisy sine waves. This new adapted noise function returns:

\[
\text{return: } \sin(n + (f \cdot z))
\]

Where: $n$ is the noise value, $f$ is the frequency of the waves, $z$ is the height of this noise position.

While this noise map can longer be correctly called a 'height map', it does offer a noise map that will work well for a gas giant texture.
Texturing

Colour Gradients

After generating a random height map, the next step is to start giving the planet some colour. My aim with this stage of planet generation is to colour the planet according to a colour gradient. Each point on this gradient will map linearly to a height value. For example, if the colour gradient were to have 10 unique values then a height between 0 and 0.1 will be coloured according the value in position 0 of the gradient. Heights between 0.9 and 1.0 will be mapped to position 9 in the gradient. For this project the planet gradient contains 100 colours, but the program is set up to accommodate a gradient of any size.

The colour gradient system chooses a small number of positions in the gradient and applies an initial colour to these positions. These initial colours are referred to as ‘exemplars’. After the exemplars have been set the program then steps through each empty point in the gradient and interpolates the colour at that points based on the two nearest points at which a colour exists. For a gradient that is $n$ colour values long the program will start at position 0 and move up to position $n-1$, computing the colour at each point. Because the loop is moving upwards through the gradient, we know that the colour value at the previous step will always be present. Therefore the program only needs to seek out the nearest colour value higher up in the gradient. When this colour is found, its position must be noted. Once the distance between the previous exemplar and the next highest exemplar has been found, the colour value can be linearly interpolated.

Example:

Two exemplars lie at: $i = 4$, $j = 28$
Position to be computed: $k = 5$
Distance: $d = j - k = 23$
Ratio: $r = \frac{d}{d+1} = 0.95833$

Colour at $k$: $colour(k) = \text{Lerp}(colour(j), colour(i), r)$

This makes use of Unity’s built in ‘Lerp’ function which can be used to linearly interpolate colours. This function interpolates the Red Green and Blue channels separately using this ratio. Here are some examples of colour gradients generated by this algorithm. Each gradient is 32 values long and contain 3 exemplars. These exemplars are shown on the upper half of the gradient.

![Image of colour gradients](image6.png)
Note that each gradient has an exemplar at the first and last position of the gradient. This is a rule that I enforce in this function. This rule is required for the function to be able to compute the gradient at each point. This is because the program will always look for the next highest exemplar in the gradient which would be impossible in certain cases beyond the final randomly place exemplar. Given that these two values will always be filled the interpolation loop only needs to calculate the values at positions 1 to n-2 in a given gradient.

The range of colours that are randomly generated are limited to follow a small set of rules that I have imposed. These rules are put in place in an effort to make gradients yield more interesting results in their random terrains. The first rule is that the colour of the land must never be too similar to the sea. Colours are initially chosen with ‘Hue’, ‘Saturation’ and ‘Value’ levels. This is a convention that is used by many development environments and graphics programs. Hue can have a positive value from 0 to 360 and is treated as an angle. This is demonstrated by Figure 7.

When an exemplar colour for the sea is randomly chosen, none the land exemplars can have a Hue that is ±60 degrees close to this sea Hue. For example, if the Hue of the sea on a planet is 0 then the land colours must all have a Hue between 60 and 300. This guarantees that the land and sea are always two very distinctly coloured sections that can easily be recognised by the user.

The Saturation and Value of each exemplar are also carefully controlled. The Saturation refers to the ‘colourfulness’ or strength of a colour. Value refers to the brightness or darkness of a colour. As a general rule, the lowest areas of a planet’s height map should always be coloured slightly darker and therefore have a lower Value. On the other hand, mountain tops should have bright, white peaks with a low saturation. This creates a simple form of shading on a planet’s texture. It also offers to quickly communicate information about the planet. The user should be able to distinguish peaks and valleys at a quick glance. If the player had a hard time distinguishing the shape of the planet then it could make for a poor experience. Since the target audience for this program is a younger demographic I felt that it was important to place an emphasis on the way in which planets are shaded and coloured. This system ensures that planets will always have a wide spectrum of colour and contrast.

Unity only handles colour with Red, Green and Blue channels. However I need to use Hue, Saturation and Value in order to control colour as discussed above. When the HSV exemplars have been generated I need to convert them to RGB. Figure 8 demonstrates the relationship between HSV and RGB. To begin, let’s assume that Saturation = 1 and Value = 1. For each Hue, only two channels from RGB are affected, these are treated as the ‘primary channels’ of the conversion function. The two primary

---

http://www.johnpaulcaponigro.com/blog/5688
the-temperature-of-color-warm-or-cool

http://en.wikipedia.org/wiki/HSL_and_HSV
channels can be found by determining what sector of the Hue scale we are in. To do this, we divide the Hue by 60 and round down. This will give us an integer from 0 to 5. Each of these is mapped to the grey sections seen Figure 8. For example, if we have a Hue of 200 (light blue) then 200/60 rounded down will give us sector 3. The diagram indicates that in this sector blue and green are the primary channels. Red is not a primary because it has a value of zero throughout all of sector 3. As Value is reduced the two primary channels decrease linearly. Reducing Saturation will increase the value in the third channel. Once the corresponding sector has been found we can compute our values and insert them in to the appropriate RGB channels.

The Sea

As previously discussed, the land colour and sea colour are always kept slightly different from one another. The colour of both the land and the sea are saved within the colour gradient. When it comes to applying the sea colour to the gradient, the lower section of the gradient is reserved for the sea colour, while the rest is left for the land. Another trick to maintain visual distinction is to alter the shade of the sea close to the shore. The sea close to the shore is shaded a slightly brighter colour, which can be seen in this screenshot.

The most important difference between the texture of the land and the sea is the use of a ‘Specularity Map’. In computer graphics Specularity is a method of giving a surface ‘glossiness’ or reflection. While a specular surface will not render a detailed reflection of its surroundings it will create a highlight on the surface as a result of any near-by lights. Our own Earth behaves very similarly. Our oceans are highly reflective, while our land doesn’t reflect any visible light whatsoever. By applying a specular texture map appropriately and making use of Unity’s specular shader we can recreate this same reflective property. The specular map is created at the same time as the terrain texture. This texture has white pixels anywhere that has a high specularity and black pixels for anywhere with no specularity. The screenshot above demonstrates this visual effect. The land is quite a dark colour whereas the sea is a bright white colour as a results of the reflection of the sun.
The Atmosphere

The atmosphere that a planet possesses also needs to be made immediately apparent by using visual effects. The clouds around a planet are created by applying a texture to a second sphere that surrounds the planet. This cloud layer is a smooth sphere that is scaled so that it will always be slightly higher than the highest terrain of the planet. In order to save computing time the clouds are generated using the same height map that affects the terrain. The texture generated is the same size as the terrain and specular texture. For each pixel, the colour is set to a light grey and the transparency is calculated based on the noise value at that position and the atmosphere property that was calculated at the start. The transparency is referred to as the 'alpha' channel of the pixel. Here is the function that is used to calculate these alpha channel values. Also included is a graph demonstrating the kind of values that result from this function.

\[
\text{cloudPixel} = \text{noise} \frac{1}{\text{atm}} - \frac{0.3}{\text{atm}}
\]

\[x = (1^n(1,z))-(0.3/z) \text{ from } z = 0 \text{ to } 3 \]
The Moons

A planet's moons are generated in a very similar way to rock planets. Each moon is given a handful of random primary properties. Moons are carefully spawned in such a way that no two moons will be too close to one another. Moons generate random textures using the same function that planets use. With moons however, there can be no sea or atmosphere on them due to their small size and mass. Therefore there is no reason to calculate a temperature, atmosphere or sea height for them. Their colour gradients are generated with 3 random colours. Because there is no sea, the range of possible hues is not restricted. These colours do, however, follow one of the other traits of planet textures. The colour of the lowest parts of a moon will be darker than the peaks of the moon. The orbit speed of a moon is calculated the same as it was for the planet. *Unity* offers a system for easily implementing object hierarchy. Each moon can have its parent assigned. Any transformations that are applied to the planet will also affect the moon. This makes it very simple to create a system in which a moon orbits a planet smoothly, while this planet is orbiting its star. Moons are generated with a lower texture resolution than planets. They are only generated for visual purposes and do not affect the behaviour of their parent planet.
Stars are the first object to be placed in a scene and randomly generated. They have a fixed texture size and can only have a colour within a certain pre-determined range. These colours are used to indicate the age of the star. As observed in astronomy, the colour of a star can give an indication as to what stage in its life-cycle it is. At the beginning of a star’s life it burns bright and blue, as it ages it cools and turns from white to yellow to red [8].

In order to constrain stars to this range of colours I created a colour chart in Photoshop. Each star has a random age which will be used to pick a location along the x-axis in the colour chart. The y-axis of this chart is the colour gradient for the star’s texture at this point. The colour at the top of the star’s colour gradient is applied to the light emitted by the star. *Unity* provides a lighting system for basic lighting of a scene. The intensity and range of the light emitted are the same for all stars.

The random noise generated by the star is done in the same way as the planets. The difference with the star is that it combines a smaller number of noise scales. Stars only generate larger scale noise maps and add all of these together. The additional smaller noise scales add grainier levels of detail to the noise map which helps for generating terrain. The result is a smoother texture with noticeable curved tendrils running through it.
Planet Mesh

Mesh Properties

Initially I was mapping the generated textures to a UV sphere. A UV sphere is constructed from a series of latitudes and longitudes similar to those seen on a map of the Earth. This allows for a direct relationship between the texture map and the polygon mesh of the planet. If the texture were to be the same dimensions as the sphere’s vertices then there would be a direct mapping between each pixel and its corresponding vertex. The figure below demonstrates how such a rectangular texture maps to a UV sphere.

While a UV sphere makes it simple for texture mapping it is not ideal for deforming the mesh. This is because the density of vertices at the top and bottom of the sphere is much higher than the vertex density along the middle of the sphere.

I opted to use a mesh that has a more even distribution of vertices. A Shape with a perfect distribution of points is known as a ‘platonic solid’. There are only five shapes that meet this criteria. The icosahedron has the highest number of faces of all the platonic solids with 20 triangular faces. In order to achieve a sphere with a higher number of faces I subdivided the faces of an icosahedron into several smaller triangles and ensure that all of these vertices are the same distance from the centre of the sphere. Further subdivision and smoothing creates a higher detail sphere with almost perfect distribution of points. After three subdivisions the icosahedron will be smoothed to a sphere with 1280 faces. I used the modelling program Blender to create this mesh. It offers a tool to create an icosahedron and subdivide it as needed.
The texture mapping for an icosahedron is considerably less neat compared to a UV sphere. It was still possible to achieve the same result by creating a handmade texture wrapper in Blender. Figure 12 illustrates how a texture is mapped to this icosahedron. This image is essentially a view of the sphere when it is unfolded and warped to fit a square. The vertices on the very top and bottom are stretched considerably more than the other vertices which creates a slight graphical glitch at the top and bottom of the rendered planets.

Now the planet is set up to be ideal for shaping the mesh based on the terrain. Below is a comparison of deforming an icosahedron and a UV sphere. The icosahedron produces much smoother results. A significant amount of detail is lost along the UV sphere’s equator.
Moulding the Mesh

With this improved mesh the next challenge is to morph the mesh based upon the random height map. This is done by looping through each vertex of the mesh. For each vertex we can get the position of the vertex in XYZ coordinates. The centre of the mesh is positioned at: (0,0,0). This coordinate can be mapped to its UV position on the height map. This is done by essentially working backwards from the texture mapping process. When mapping the texture, we took the UV position of the height map and found the corresponding XYZ coordinate. We work backwards from this by using the following functions [7]:

\[ U = \frac{\varphi}{2\pi} \quad V = \frac{\pi - \theta}{\pi} \]

Where:

\[ \varphi = \tan^{-1} \frac{Y}{X} \quad \theta = \cos^{-1} \frac{Z}{r} \]

The radius of the sphere is represented by \( r \) in the function above. The U and V coordinates then need to be rounded to the nearest int. This UV position points to the height of the planet at this XYZ position on the mesh. Because the centre of the sphere is at (0,0,0) a vertex can be translated away from the centre by simply multiplying its XYZ coordinate by a single scalar. Therefore each vertex can be transformed by multiplying its position by the height of the planet at that position. The only precaution that needs to be taken at this stage is to account for the height of the sea. If the height at a particular vertex is less than the height of the sea then it should be transformed by the height of the sea and not the height of the terrain. This will ensure that there will be no point lower on the planet than the sea. If this precaution was not taken then the sea would be bumpy and displace just like the land. The sea needs to be kept smooth to maintain its visual appearance.
User Experience

Camera Controls

The camera in this game is set up to give the player the feeling of being another orbiting object within the solar system they are exploring. The camera orbits a central point and has a feeling of momentum when used. Clicking and dragging with the left mouse button will orbit the camera around the point that it is focused on. Dragging with the right mouse button will pan the camera around the scene, repositioning the focus position of the camera.Scrolling the mouse-wheel zooms the camera in and out. All of these three functions are controlled by acceleration and deceleration modifiers that give the movement of the camera a degree of weight. An input from the user will be either -1 or +1 depending on its direction. The acceleration function is:

\[ \text{output} = f + \frac{i \times a \times \Delta t}{\text{timescale}} \]

Where:
- \( f \) is the current value of the variable being accelerated.
- \( i \) is the input value
- \( a \) is the acceleration
- \( \Delta t \) is the time since the last frame
- Timescale is the speed that time is passing.

The function divides by the timescale so that the camera moves with the same speed regardless of what time speed the player has set the game to.

The camera is restricted within a certain boundary. It cannot tilt through more than an angle of 90 degrees. This means that the player can look at a top down perspective or a side-on perspective or any angle in between. The player cannot, however, tilt below this and look at the solar system from the bottom. This helps to ensure that the player will not lose track of what direction is ‘up’ within the solar system.

User Interface

My aim with the user interface was to keep it clean and minimal. When the player is exploring the solar system they can click on each planet to get some information on that planet. However I did not want to flood the screen with all of the information available for that planet. Instead, the player is given four of the most important properties of the planet; the size, the distance from the star, the temperature and the atmosphere of that planet. This information is given on a small ‘info card’ that will hover beside the planet it pertains to.
The info card also gives the planet a brief description. For example that planet in the screenshot above is described as a cold planet. Other types include hot planets, habitable planets and barren planets. Habitable planets are those with a reasonable temperature and atmosphere and barren planets have a survivable temperature but no atmosphere.

When the player clicks the ‘Zoom’ button the camera focuses on the planet and the info card is expanded and moved to the side of the screen. This larger info card offers the user all of the information available for that planet. Users can view properties such as the length of the planet’s day and year relative to Earth’s, the mass, gravity and number of moons.

Simulation

The simulation in this game is quite minimal. The orbit speed of all of the planets and moons are calculated at the start. These speeds are calculated as the angular velocity of the orbiting body. Each of these bodies is then rotate around its parent body. Because the speed of these rotations is calculated using real-world physics, the planets can be seen to orbit realistically. Planets that are much further away from the star will take a far longer time to complete one full orbit.

In the top left of the screen are buttons for controlling the passage of time. Players can speed up time to 1000x the default speed and watch the planets and moons orbit at a hugely increased speed.
Evaluation

Conclusion

This program generates planets with a style very similar to what I aimed to achieve. They are highly unique and pleasing to look at, similar to those in the game *Spore*. Terrain appears varied and interesting on almost every rock planet that is generated. Small planets have only a handful of peaks and valleys while larger planets have a vast expanse of terrain features. The morphing of the mesh helps a great deal in demonstrating these features.

The physical properties of planets are emulated well. Temperature values are calculated in degrees Kelvin and the atmosphere of a planet seems to vary anywhere from 0 to 3x the atmosphere on Earth. All of these properties very closely relate to planets in our own solar system. The properties of each planet affect the atmosphere and sea of planets in a way that always makes sense.

The time taken to generate a solar system is also reasonably short. Taking measures such as scaling the planet’s texture based on the size of the planet ensure that there is never an excessive level of detail on smaller planets. Because of the improvements in efficiency made by Simplex Noise the time to generate the noise maps is much faster than it would be with my other potential options, and the memory footprint is also kept to a minimum.

While not all of the interface utilises the graphical elements that I wanted to use, the most important parts do. The info cards are clean and do not occupy more space than they need to. While the buttons for controlling time and generating a new solar system do not make use of this graphical style, they still occupy very little room on the screen. If I had more time I perhaps would have found a way to give the user greater control over time in the interface.
The graphical quality achieved is very close to my initial vision. Specular maps for the sea and semi-transparent cloud layers serve to create a visually engaging experience. This is further helped by some of Unity’s built in graphics features such as colourfull local illumination and shading.

One of my main objectives with this project was to make these random solar systems interactive in some way. I had hoped to implement a feature to allow users to change the properties of planets and observe what impact this had on the planet. For example, moving the planet closer to the star could increase temperature and lower the sea levels. Unity also offers simple tools to collision detection. I had hoped to allow users to potentially cause planets and moons to collide with on another if players were not vigilant enough. Unfortunately I did not have enough time to implement these interactive features.

I also hoped to have noise and texture generation functions run on their own threads. This would allow for several benefits. Primarily this would significantly reduce the load time for powerful, multi-core processors. Secondly, this would allow me to generate higher resolution textures more easily in the background while the game is running. This feature would have paired very well with the interactive features. When a user makes a change to a planet, the new textures could be generated in the background without affecting the frame-rate or performance of the game in any noticeable way. It would also have allowed me to generate a much larger number of texture resolutions. The texture could initially be rendered at a drastically lower resolution. This would allow levels to be loaded in less than 10 seconds in most cases. The higher resolution textures could then be streamed in seamlessly as the player moves closer to a planet.

Finally I had hoped to offer a greater incentive for players to explore and learn. Initially I envisioned a system where players would have to spend points to alter planets and explore new solar systems. These points would be earned by calculating missing values on the info cards of certain planets. Because the planets do not follow real world physics 100% accurately, I could think of no way of implanting this feature without essentially teaching the user formulae that are slightly spurious in some way.

Future Work

This project is essentially the framework for a more complex game. This system of random planet generation could be applied to any video game set in a random space-scape. I plan on exploring the possibility of creating a real-time strategy game similar to Planetary Annihilation, or an exploration game where players must fly from one planet to another. Both of these would require that I can display sections of the planet at a higher resolution when a player gets closer to
the planet. For example, in an exploration game, the player might be walking on the surface of the planet. This surface should be rendered with a high level of detail. However the terrain on the other side of the planet should not be rendered with this level of detail. It should not be rendered at all. Therefore the most important feature that needs to be implemented is the ability to infinitely scale the detail of planets and do so in an intelligent and procedural way.

I also want to add a greater range of possible objects that could be spawned. For example planets with an asteroid belt, black holes and stars that have died and no longer emit a significant amount of light. I also want to present players not only with a look at a solar system, but also an entire galaxy. A map of random stars would be spawned in an enormous cloud. Players could move from one star to another, and bookmark stars that they may wish to return to. This is a feature present in the game Spore.

![Screenshot of Spore's Galaxy](attachment:image)

Finally, I want to investigate making solar systems change over time and make them more reactive and animated. Textures generate for the star and clouds of planets are static texture images. I want to make them animate and change over time. This could be done by implementing Simplex Noise in four dimensions. The fourth dimension will allow me to animate the noise in a smooth way. This will work especially well for the texture on the star. For the clouds on planets I will perhaps look in to switching from a flat texture to a particle system that generate 3D clouds along a planet’s surface. The solar system could be made more ‘reactive’ by implementing more in-depth physics. Currently all planets and moons orbit in perfect circles. In the real world however, all objects orbit in elliptical shape. To make the physics more complex and realistic I could support collisions of planets that are moved, or even allow solar systems to spawn two planets with elliptical orbits that overlap with one another. I would not only have to account for collisions, but also the gravitational attraction between two planets even when they are moving close to one another.
Learning Outcomes

This project has been an exciting and inspiring experience for me. This is one of the first large projects I have taken on that was entirely my own invention. While I entered this project with prior knowledge and interest in astronomy I have discovered so much more about the physics of our solar system.

The tools provided in Unity have given me dozens of opportunities to learn about the various challenges encountered in developing interactive games. I have come to realise the importance of optimisation and performance analysis of code. I made use of some of Unity’s ‘Pro’ features such as analysis of memory used and load times. This encouraged me to seek alternate methods to solving a problem more efficiently. This led me to my discovery of Perlin Noise.

This has been my key area of learning in this project. I spent several months exploring various forms of procedural noise generation. I have developed a fascination for utilising noise to produce random objects with a degree of coherency and predictability to them.

I have also attained other useful skills essential for game development. For example learning about 2D texture generation and mapping to a sphere and random morphing of a 3D mesh. I also developed a simple algorithm for colour interpolation.

Lastly I learned the value of being mindful of the user experience at all times. Not only was I trying to solve the problem of randomly generating planets, but I ensured that it was at all times simple and fun to use for a new user.
Appendix

The final build of the project is an executable file on the attached CD called RandomSolarSystems.exe

The Unity project including all the scripts and assets I created are in the 'Unity Project' folder. The code is located in `\Unity Project\ Assets\ Scripts`
Bibliography


