Script

* Hello, my name is Adam, and this is my presentation on the progress I have made with portfolio final project.
* As part of this presentation, I shall cover a basic description of the project, and its context within the wider research field and its importance. Then I will summarize the research I have completed thus far and explain some of the core concepts and findings I have found within this process. Afterwards I will then show my progress in the implementation and demonstrate the current functionality. Finally, I will outline my plan for completion of both the implementation and dissertation.
* This section will explain the project, the reasons I have chosen it, and why it is important to study.
* In the gaming industry, specifically within the sci-fi genre, there is an increasing tendency to use a large amount of procedurally generated content. This approach allows for developers to create vast universes and worlds for the player to explore, either as a core gameplay concept, or as an aid in game development so that developers can focus more on gameplay features instead of world building. 3 examples shown above are some examples of procedural generation, specifically planet generation. On the left is Elite Dangerous by frontier developments, on the top right is no man’s sky by hello games, and on the bottom right is space engineers from keen software house. All 3 of these games, as well as many more, have inspired my project to focus on this idea of procedural planet generation
* For this project, I plan on combining all the benefits of these games into one simple product, eliminating all of the negatives as well. For example, although the scale of Elite dangerous is incredibly realistic, so is their planets. Which is good in a simulation context, but if you want to create interesting, diverse worlds to keep the player interested, this is less ideal. No man’s sky on the other get this diversity very well, and all the planets are noticeably different from each other, but they suffer from the issues of being on a much smaller scale. Additionally, even though space engineers suffer from a much more restrictive generation system and their planets are much smaller, the one feature they have over both these other games is transitioning from space to the planet. Both Elite and no man’s sky have a noticeable transition from space to their planets and vice versa, whereas space engineers handle this transition much better, and the experience is almost seamless.
* As a result of this first initial look at current works within the industry, I would like my final product to be a Unity package that allows for the creation of procedurally generated planets, at a user defined size, diversity through biomes.
* For this section, I will summarize some of the major theorists and algorithms that I explored during the ongoing research process for this project.
* There were 5 main areas of research that I wanted to investigate as part of this project. These include, Sphere Generation, Terrain Generation, level of detail systems, Atmospheric rendering, and floating-point precision errors. I will now break down all these individual areas of study withing the next few slides of this presentation.
* The first major area of investigation was the best way to generate a sphere mesh. This was crucial as planets, at their core, are spheres with extra lumps. Although some planets and moons do exist that do no not follow this spherical shape, such as the moon of mars Phobos, for the sake of this project’s scope, only spheres will be investigated. Initial investigation into the subject showed there were 4 well established ways to create the vertices of a sphere. These were the UV sphere, the normalized cube, the spherified cube and the icosahedron. These methods are the most documented algorithms I found during my research, and examples of each of these methods can be seen in the picture here. Although these methods were good, they all suffered from a less than perfect point distribution, and this would be important to lower the chance of visual artefacts and allow for a more consent level of detail. As such my research took me to what is known as the Fibonacci sphere. This algorithm uses the Fibonacci sphere to generate the most evenly distributed points in all the methods researched. This method would have been perfect if it weren't down to some major downsides. This method is much more computationally and mathematically complex to generate this sphere. The main issue arrives from trying to triangulate the generated points, as the algorithm generates the points in a nonlinear way, making it too complex for my understanding to triangulate. Additionally, again due to this non grid structure, it would be incredibly hard to subdivide this sphere into different terrain chunks, which is important for level of detail which will be discussed later. As such the method for implementation that I have chosen is the spherified cube, as it had good enough point distribution, whilst also being much easier to triangulate and break into chunks.
* Another area I briefly researched into was Marching Cubes. This algorithm uses voxels to create the shapes of the terrain, instead of having to manually specify the vertices of every triangle of the terrain. This was investigated, as in theory it would have allowed for deformable terrain, as well as much more advanced terrain features such as caves and overhangs. There was even a paper by Zackary P. T. Sin and Peter H. F. Ng, on a method for implementing this algorithm for a spherical space, for use within planet generation. Unfortunately, during initial testing, this algorithm proved to be far to slow for runtime operation, at least at the scale that I wanted to achieve. In a future iteration of this project, I would like to have another attempt at implementing this using the GPU to aide in computation.
* The next topic I conducted research into was different methods and algorithms of how to generate terrain, either on a traditional plane, or on a sphere. The first well established way of generating a terrain that I researched, was the use of heightmaps and noise to generate the terrain. The most basic form of this is to use a form of noise called coherent noise, such as Perlin or simplex noise. This involves taking the vertices of a plane, then with the heightmap generated using the noise algorithm, moving the Y position of all the points in the plane. One downside of using just one layer of noise to generate terrain, is that you end up with results as seen above, demonstrated in a paper in Michelic. This example was generated using a single layer of Perlin noise, and as you can see it does not exactly look like realistic terrain you would expect to see in the real world
* Even though these issues can be helped by adding additional layers of noise, during my research I found an interesting approach to use multiple layer noise as a base for the terrain and biomes to be created from. A paper written by Fischer et al. called auto biomes, outlines a multiple stage approach using a multistage process to generate diverse terrains. The process starts with a base layer of noise forming the basic terrain, then applying some weather simulations to define this terrain into different biomes, then applying additional biome specific noise to these different biomes and refining them based of that biome’s characteristics. This is the approach I will be taking for my implementation, as it offers the diversity and control, I need for the product I would like. One additional mention that would benefit the realism of the terrain, was discussed by Sean Murray in his 2017 GDC talk, where he talked about using a machine learning model to use real world terrain to make the generated terrain more realistic.
* A crucial part of this project would be the ability to alter the meshes detail, depending on the distance the camera is from the model. This is because of the size of the planets, if full detail was used for the entire planet all the time, not only would unity’s mesh system not allow it, but most consumer computers would probably not have enough RAM or VRAM to store all the required vertex data. As such several different options were investigated for a level of detail system, to alleviate this computational strain. The most well documented method of doing this on large terrain data sets, is utilizing a quadtree. This process involves using a high resolution heightmap at the highest possible detail level, then further simplifying the mesh further and further out you go, using the quadtree to specify which points to use in the heightmap. This, in addition to using some additional culling methods, such as view frustum culling, should allow for sufficient render performance of these planets.
* Another aspect of planets in game that makes them much more immersive and realistic, is the addition of atmospheres and skies to the planet. From the multiple sources that I have researched, including works by Schafhitzel et al and Elek, all pointed towards a very similar methodology for achieving a realistic in planet and outer planet atmosphere experience. This algorithm works by first precomputing complex volumetric scattering equations into lookup tables, which are then fetched by the fragment shader when they are needed. The two images on the right of the screen are this method as I have described, whereas the images on the left are a much older method, that only considers a single scattering of light. Due to this algorithms’ multiple mentions across many works, and its claims of speed by elek, this will be the method that I shall attempt to implement.
* One additional thing that required small additional research, that is issues that arise with floating point numbers. Due to the scale that I intend to create, floating point number are not precise enough. Around 1000km is where floats lose their precision, and with earth being 6371 km, there would be some issues that would occur, such as camera jitter, and other game objects not being in the expected position. One method, as described in the 2013 Unite conference by the Kerbal Space program developers, showed that moving the camera and objects between different spaces and scales was how they overcame this issue. Additional research discussed Sean O'Neil discussed a similar method, via scaling the planetary bodies and keeping them close to the player. As such, a combination of scaling and moving the game world around the player camera will be used to avoid any issues involving floating point inaccuracies
* This section will show what I have worked on thus far, regarding implementation as well as current work completed on the dissertation side of things
* The first thing that I wanted to implement, was the generation of a sphere, to give myself a good base for the rest of my work. The above algorithm, inspired by one written by Sebastian Lague, creates the face of a normalized cube, given the normal of that face and its resolution. This is a good starting point, but soon (or if currently implemented say that) this will be converted to use a spherified cube, as research indicated that it would be more appropriate to this project and have less issues with surface normal. The whole concept of this algorithm is to iterate through all the points on the face, and then apply a transformation to them to convert them from the unit cube to the unit sphere. As such, to convert to a spherified cube, only the point on Unit Sphere would be changed. After the points and triangles are generated, these are then fed to unity’s mesh class to be converted to something that can be rendered
* As you can see I have a very lumpy looking shape, this is the result of a simplex noise library that I have applied to the vertices of the unit sphere I am generating. This was a very basic demo just to show that the simplex noise script I was using was functional, and the way I was altering the vertices by multiplying their normal was working as intended.
  + Then just to show that the sphere generation is working as intended, I have removed any noise I have added to show the sphere, as well as how the points are distributed using a spherified cube.
  + Finally, I am demonstrating the first step in my implementation to remove any floating-point precision errors. For this, I have bounded the camera to a 1000 by 1000-unit box and am moving the planet whenever the camera reaches the edges of its bounding box.
* This section will Include my plans for continuing project, and what will be undertaken in the following weeks.
* On screen is my initial Gannt chart that I submitted as part of my proposal. However due to some initial problems experienced whilst trying to start building the implantation within my own engine, this has put me a few weeks behind schedule. However, I am still very much on track to complete everything on schedule. As stated before, I have been able to generate and triangulate the points of the sphere, but contrary to what this Gannt chart says I am currently working on both the Level of detail system as well as the Basic terrain generation, as their implementation is more intertwined than initially planned
* On Screen now is my plan for completion for the rest of the implementation. As you can see, my plan still concluded before my deadline of the 24th of April, when I would like my implementation to be complete, and my dissertation to be in a first draft state. If I can complete all these items ahead of schedule, I will also conduct research into adding oceans to my planets, and well as investigating other small refinements and features that I could add.