[Analysis 2](#_Toc96782930)

[Problem Identification 2](#_Toc96782931)

[Stakeholders 2](#_Toc96782932)

[Interview 2](#_Toc96782933)

[Research 3](#_Toc96782934)

[Requirements 6](#_Toc96782935)

[Functional Requirements 6](#_Toc96782936)

[Non-Functional Requirements 7](#_Toc96782937)

[Success Criteria 7](#_Toc96782938)

[Design 8](#_Toc96782939)

[User Interface Design 8](#_Toc96782940)

[Main Menu 8](#_Toc96782941)

[Generate Terrain Screen 11](#_Toc96782942)

[Generate Heightmap 14](#_Toc96782943)

[Load Heightmap 16](#_Toc96782944)

[Decomposition 19](#_Toc96782945)

[Algorithms 24](#_Toc96782946)

[Indexing of Data 24](#_Toc96782947)

[Calculating Vertex & Face Normals 25](#_Toc96782948)

[Creating Faces 26](#_Toc96782949)

[Creating Vertices 28](#_Toc96782950)

[Creating Heightmap Image 29](#_Toc96782951)

[Generating Noise (Perlin Noise) 30](#_Toc96782952)

[Inputs & Outputs 31](#_Toc96782953)

[Testing 33](#_Toc96782954)

[Development & Testing 34](#_Toc96782955)

[1st Prototype – Editing the File 34](#_Toc96782956)

[2nd Prototype – Perlin Noise 36](#_Toc96782957)

[3rd Prototype – Displaying the Terrain 38](#_Toc96782958)

[4th Prototype: Combining all Functions 39](#_Toc96782959)

[5th Prototype: Generating Flat Surfaces 39](#_Toc96782960)

[Prototype 6: Adding Normal Generation 43](#_Toc96782961)

[Evaluation 43](#_Toc96782962)

[Criteria Met 43](#_Toc96782963)

[Usability 43](#_Toc96782964)

[Issues 43](#_Toc96782965)

[Final Code 43](#_Toc96782966)

# Analysis

## Problem Identification

3D terrain is frequently used in 3D games and other projects (such as creating a fantasy world with lore behind it), however for smaller developers/hobbyists this is usually done by hand, which is very time consuming process, therefore the proposed software will allow for a small environment (terrain/planet) to be generated, with each time being a new, unique piece of land, intended for use in prototyping or worldbuilding.

Current solutions exist and are commercially available (such as World Creator and Instant Terra), however many are pricey (which may be an issue for many hobbyists who do not want to spend several hundred pounds for a piece of software), with many features which could overwhelm someone/not be necessary for prototyping purposes.

The proposed software would be used by small developers looking to prototype certain aspects of their game in a world (to see how it looks and interacts), as well as authors who want to create a fleshed out feeling world. On top of this, hobbyists would also find it useful for creating worlds to develop their games in.

The proposed system will manipulate a mesh, using several types of noise (i.e. Perlin and Worley) with controllable parameters to control things such as mountain quantity, terrain roughness and steepness, which will then be able to be viewed quickly, then exported as a Wavefront .obj file, which is able to be used in numerous programs, such as Blender and Unity.

The success will be measured as if the terrain is unique, the file is able to be exported, and the generation has controllable parameters.

### Stakeholders

## Interview

For my project, I interviewed a game developer who frequently makes small scale demos to test out certain features. They shall be referred to as Programmer A from hence forth.

Programmer A often makes small scale simulations and models in order to test out certain features for larger projects. Due to this, it is not feasible for them to create a map/piece of terrain by hand for each project, as this would be very time consuming. This makes them a good stakeholder as they would directly benefit from the tool, and their advice will likely be made with understanding of potential limitations.

**Question 1:** Have you used any 3D terrain generators before?

**Programmer A:** I haven’t used terrain generators, however I have used several character model generators. They are generally to complex for my use case, however.

**Question 2:** So would you prefer a simple program with fewer parameters?

**Programmer A:** That would be better for me, then I can create the terrain very quickly for my prototyping. Or I could maybe load in heightmap.

**Question 3:** Which programs do you intend to use the models in?

**Programmer A:** I use a few game engines, so Unreal (UE4), Unity and Godot. I use obj files for this currently, so that might be a good format to output in. I could also load it in Blender then to edit it further!

**Question 4:** Would it be beneficial if the terrain was coloured in some way?

**Programmer A:** That would be useful but if it was not I still wouldn’t be bothered. It may be interesting to add colours based off of height and steepness of the terrain in the future though.

**Question 5:** In terms of the size of the generated terrain, would it be useful to generate different sizes?

**Programmer A:** That would be nice, a chunk based system like in Minecraft would be interesting to use, but having different shapes, like 3x3 or 16x9, would certainly be useful.

**Question 6:** Would having different noises used in the generation be useful?

**Programmer A:** In certain situations that may be a useful feature to have access to, however for me that is not too much of a concern due to the use case I have.

**Question 7:** Is a preview of the terrain useful?

**Programmer A:** It would be useful, as otherwise I would have to navigate to where the file is stored and open it with 3D Viewer.

**Question 8:** And finally, would you prefer a web-based client or a program that runs on your machine?

**Programmer A:** Having a web based client would be really useful if I am working on a machine that is not so powerful, though it is not a necessity. It would be really interesting to see, however!

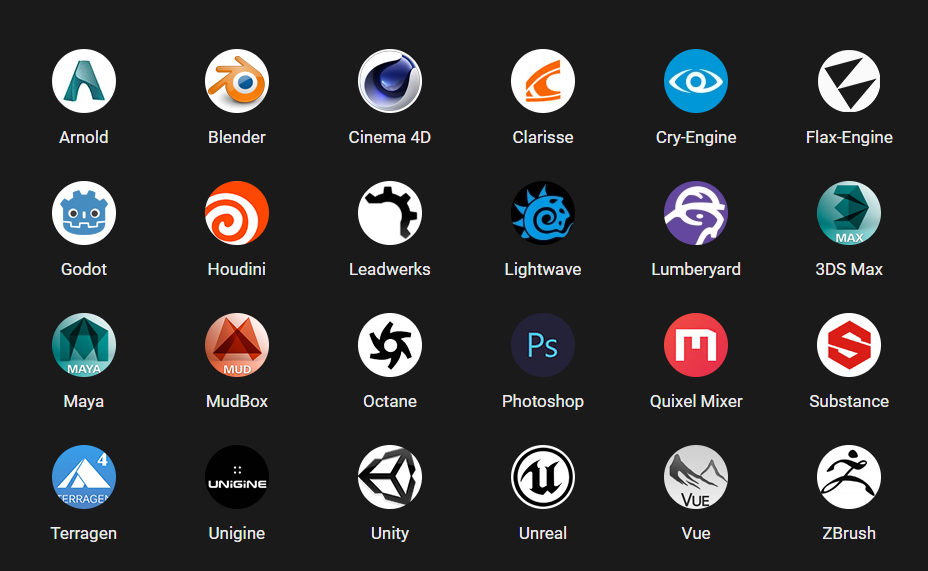
## Research

Currently, there are multiple solutions on the market, such as World Creator ([world-creator.com](https://www.world-creator.com/)) and the Low Poly Terrain Editor plugin for blender ([Low Poly Terrain Editor - Blender Market](https://blendermarket.com/products/low-poly-terrain-editor)). Both are paid for tools regardless of the use case. For non-commercial uses however, World Machine ([world-machine.com](http://world-machine.com)) is a free alternative which can be used, but can also be used in industrial settings using the paid license.

**Example 1: World Creator**

Overview:

World Creator allows for the real-time editing of terrain, which utilizes the GPU in order to increase the speed at which the terrain modifications can be completed and rendered. It is used by companies such as Blizzard and Ubisoft. As well as the creation of landscape, it also allows them to be painted in diverse colours and have trees and other vegetation added. It is compatible with many game engines (see photo).



1 Taken from world-creator.com

Pros:

Editing can be done in real time, this makes it easier for the user to see how the variables that they are changing effect the overall terrain

Can be exported in .obj and .fbx file formats

Allows for manual editing of terrain

Uses GPU acceleration

Cons:

Costs - $289 for a fully featured Windows license

The program is crash prone (according to reviews of the software on Steam)

Has a very steep learning curve, making it hard to create good looking terrain for beginners to the software

**Example 2: Blender Low Poly Terrain Editor Plugin**

Overview:

This is a tool for the 3D modelling software Blender that allows low poly terrain to be edited. It has a selection of 5 stylised procedural textures that can be applied, in order to make terrain from different biomes. It also comes with low poly models that can be added to the scene in order to help bring it to life, as well as shaders for water and a skybox.

Pros:

Cost - $10 for the editor, or $39 for the editor and all other content

It is very fast to edit low poly, isometric and voxel terrains

Allows any model to be converted into these styles

Can be used to apply procedural textures of magma, snow, creeper, sand and terrain

Allows roads to easily be added to the terrain with editable paths

Allows the planting of trees

Enables ‘scattering’ of models over the terrain

Allows for the exporting of all supported file types by Blender (obj, fbx, 3ds, dxf, svg, stl, etc.)

Cons:

Only a small area can be generated and edited

Very slow to process tools such as the scatter tool

Only has the majority of features if the more expensive version is purchased

**Example 3: World Machine**

Overview:

World Machine is a program that is more centered around using algorithms to generate the terrain than to manually sculpt the whole thing by hand. Due to this, there are many customizable parameters with editable curves and modes that can be utilized in order to create the exact terrain wanted, however also allows for painting on top of this. It has been used in numerous games, as well as by independent artists.

Pros:

Edits made can be previewed in real time

Workflow is based on graphs, allowing for 8k resolution terrain generation instantly

Enables the creation of macros to speed up repetitive tasks

Allows changes to be undone and redone if necessary

Allows terrain to be generated from fractal noise

Allows the simulation of nature, such as erosion and water systems

Allows exporting in numerous file types (obj, tiff, OpenEXR) at a variety of resolutions

Can break the terrain down into ‘tiles’ for exports

Can be used for free

Cons:

To unlock all features, it will cost $1999 (Studio Site License)

Licenses only include 1 year of updates

Cheaper versions only support building with 2 cores (slowing down the export process)

|  |  |  |  |
| --- | --- | --- | --- |
|  | World Creator | World Machine | Blender Low Poly Terrain Editor |
| Price | $289 | Free/$119 | $10/$39 |
| Supported Export Filetypes | Obj, raw, png, fbx, bmp, tiff | Obj, tiff, png, OpenEXR | Obj, FBX, 3DS, DXF, SVG, STL, UDIM, USD, VRML, WebM, X3D |
| Allows manual terrain manipulation? | Yes | Yes | Yes |
| Uses parameter based workflow? | No | Yes | No |
| Allows props (trees, vegetation, etc.) to be added? | Yes | Yes | Yes |
| Allows water to be simulated? | Yes | Yes | No |
| Allows infinitely large areas to be generated? | Yes | Yes (dependent on version) | No |
| Allows terrain to be edited and previewed in ‘real time’? | Yes | Yes | No |
| Uses GPU acceleration? | Yes | Yes | Yes (For rendering the final result) |
| Allows the use of fractal noise in terrain generation? | No | Yes | No |
| Allows tiles to be exported? | No | Yes | No |

## Requirements

## Functional Requirements

* Can export .obj files, so that it can be used in most software (such as Unity & Unreal Engine 4), as requested by the stakeholder
* Can create a flat plane, so that it can be seen that the program works
* Must allow for the editing of certain parameters so that the terrain can be best suited for the user and their requirements, as requested by the stakeholder
  + Amplitude
  + Normal generation
  + Weighting of noise
  + Scaling of coordinates inputted to noise
  + Seed used for noise generation
  + Weighting of trigonometric functions
  + Scaling of coordinates inputted to trigonometric functions
  + Offsetting coordinates inputted to trigonometric functions
* Must allow for a ‘chunk-based’ system, allowing terrain to be infinite, similarly to the competitors
* Must allow for ‘chunks’ to be made of assorted sizes so that they can be utilized in any situation, similarly to the competitors (namely World Machine)
* Allows the use of a selection of different noises in order to create different styles of terrain
  + Perlin Noise
  + Worley Noise
  + Stacked Perlin Noise (Fractal Noise)
  + Heightmap
* Allows a preview of the terrain to be viewed so that the user knows what they will be exporting
* Preview allows camera rotation and panning, allowing the user to get a better view of the terrain
* The user should be able to select the location of the export and the file name

### Non-Functional Requirements

* Must have GUI that allows for parameters to be easily changed
  + Some parameters should have sliders and others text input boxes
* The GUI must explain parameters if needed

## Success Criteria

|  |  |
| --- | --- |
| **Criteria** | **How to Evidence** |
| Can export an .obj file | Screenshot and loading in Windows 3D Viewer without error |
| Can create a flat plane | When loaded, the surface is flat and shaded correctly |
| Must allow for editing of specific parameters | Video showing changing of parameters and the effect on the surface |
| Must allow for a ‘chunk-based’ system, allowing terrain to be infinite | Export consecutive ‘chunks’, load into 3D editing software and check they are continuous |
| Must allow for ‘chunks’ to be made of assorted sizes | Export different sized ‘chunks’, load into 3D editing software and check they are different sizes |
| Allows the use of a selection of noises | Video showing changing of noises and the effect on the surface |
| Uses trigonometric function to generate hills | Screenshot of a chunk which is a wavelength long, showing the curve |
| Allows a preview of the terrain to be viewed | Screenshot of viewer being opened and used |
| Preview allows camera rotation and panning | Video of these functions being used |
| User should be able to select the location of the export and the file name | .obj file can be saved to a folder, then opened from said folder |
| Illegal inputs are prevented | Only correct datatype allowed in each input |
| Clear error messages should be supplied | When illegal inputs are entered, they are clearly highlighted |

# Design

## User Interface Design

### Main Menu

In order to design the main menu, I split the screen into 2 types: constants and buttons. The buttons are then further split up into “Load Heightmap”, “Generate Terrain” and “Generate Heightmap”, as these will be all the functions that the software is able to provide. These can be seen in the following picture, with constants and buttons green and orange respectively.

A picture containing text

Description automatically generated

I then prepared a design myself that made sense and would be possible to implement. The design had the constant at the top, and underneath was a 2x2 grid of buttons, with the bottom right slot being empty. If I implement this, I would fill this slot with a “Quit” button or something of the same effect.

After this was prepared, I asked Programmer A to try out the interface with the paper model. Initially, he tried the design and stated: “Whilst it is usable, I find that the empty gap leads to the GUI looking a little odd”. We then spent some time moving different elements around, until we settled on a design that placed all of the elements in a single column, with the buttons being on the bottom. It was determined that a “Quit” button was not necessary, as the window will not be full screen and has both “Minimise” and “Close” icons on the top right (as all Windows programs do). Programmer A suggested the order of “Generate Terrain”, “Generate Heightmap” and finally “Load Heightmap”, as he felt that this order would reflect the frequency of use between the different options. He also recommended that the differentiated colour between static elements and buttons be kept. The layout can be seen in the following image.

Text, letter

Description automatically generated

After creating this layout with Programmer A, I used Figma ([www.figma.com](http://www.figma.com)) to mock up the GUI and change colours to find an interesting yet not too contrasting colour scheme. I chose to maintain the different colours between the buttons and static elements, though I wanted them to be a similar colour so that they were clearly part of a consistent system. This resulted in using two different shades of a bright brown-orange colour on a blue background, creating a clear distinction between each element and the background, as well as between each element type, without creating a menu that puts strain on the user’s eyes.

I also tried adding curved corners to the different elements, allowing me to further differentiate between the buttons and the static elements with more than just colours. This change will allow my program to be more inclusive to people who struggle with colour blindness, as well as helping users understand the use of each element with more ease. Both this change and the colour scheme will be able to be carried forward onto the other screens, in order to maintain the consistent feel of the tool.

After designing in Figma, I then interviewed Programmer A regarding the new design. He was happy with all aspects, bar the colours, saying the GUI was simple and easy to use and understand. However, he found that the colours were too bright, though a good match, and suggested making them “duller of more pastel”. The GUI with the colour changes incorporated can be seen below.

Graphical user interface, text, application

Description automatically generated

After designing in Figma, I used Tkinter Designer ([TestTest4253/GUI-Designer (github.com)](https://github.com/TestTest4253/GUI-Designer)) to transfer the Figma design into Tkinter code that I could use within my project. The end result is as follows.

Graphical user interface, text, application, chat or text message

Description automatically generated

### Generate Terrain Screen

When designing the paper version of this GUI, I employed the same colour coding system as the one used for the main menu, however I also included a yellow colour for text input boxes. The buttons in this design are simply “Back” and “Generate”, however there are many more text input boxes. These are “X Size”, “Y Size”, “X Chunk”, “Y Chunk”, “Seed”, “Bias”, “Smoothing” and “Generate Normals”. These can be seen in the following photo.

Diagram

Description automatically generated with low confidence

Similarly to the process used when designing the main menu, I designed a sample layout for Programmer A to comment on. This layout placed the title (“Generate Terrain”), at the top, with “X Size”, “Y Size”, “X Chunk” and “Y Chunk” inline horizontally below it. The other text entry boxes were organized in a 2x2 grid below, with the “Generate” and “Back” buttons at the very bottom.

I then took my design to Programmer A in order to get him to use the design and comment on the user interface. Initially, he stated: “The menu is very usable as it is and I can easily tell what needs to go where. There do seem to be too many settings though, I can’t see that I would use X Chunk or Y Chunk very often at all”. We then changed the positions around until Programmer A felt that the design was as usable and aesthetically pleasing as could be. The result of this was “Generate Terrain” again being at the top, with “X Size”, “Y Size” and “Seed” on the row below, “Bias” and “Smoothing” on the row below that and finally “Generate Normals”, “Back” and “Generate” on the bottom row. However, Programmer A stated that it would be ideal if the text entry elements would have a background behind them, in order to differentiate between them and the buttons. This layout can be seen below.

A picture containing diagram

Description automatically generated

Text

Description automatically generated

After my meeting with Programmer A, I created a mock-up of the design in Figma ([www.figma.com](http://www.figma.com)). For the colours behind the static elements and buttons, I used the same colours as before. This helped me to create a consistent user experience between every window in the software, which should hopefully make the transition less jarring for the users of the software. However, I also had to decide on the colours to use for text entry fields and the background behind the text entry fields.

In order to maintain the use of colours to symbolise which elements are interactable and which are not, the colour of the text entry fields is the same as that of buttons. However, there is no text on the rectangle (unlike on button elements), instead labels use a small font positioned on top and to the left of the text entry fields. In order to further differentiate between the buttons and text entry fields, I added the change in background that Programmer A suggested. This was in the form of a purple square behind the text entry fields.

After creating this design, I sent it to Programmer A in order to get their feedback. They liked the design, as all elements were of the same colour and curvature of the previous menu. However, he stated that the background for the text entry fields would look better if it was, again, “duller” and “more pastel”, as well as if the rounding of the corners was smoothed out. The GUI with these changes made can be seen below.

Graphical user interface

Description automatically generated

After finalising the design, I exported the GUI to Tkinter using the same tool as last time, Tkinter Designer ([TestTest4253/GUI-Designer (github.com)](https://github.com/TestTest4253/GUI-Designer)). The final GUI can be seen below.

Graphical user interface, text, application

Description automatically generated

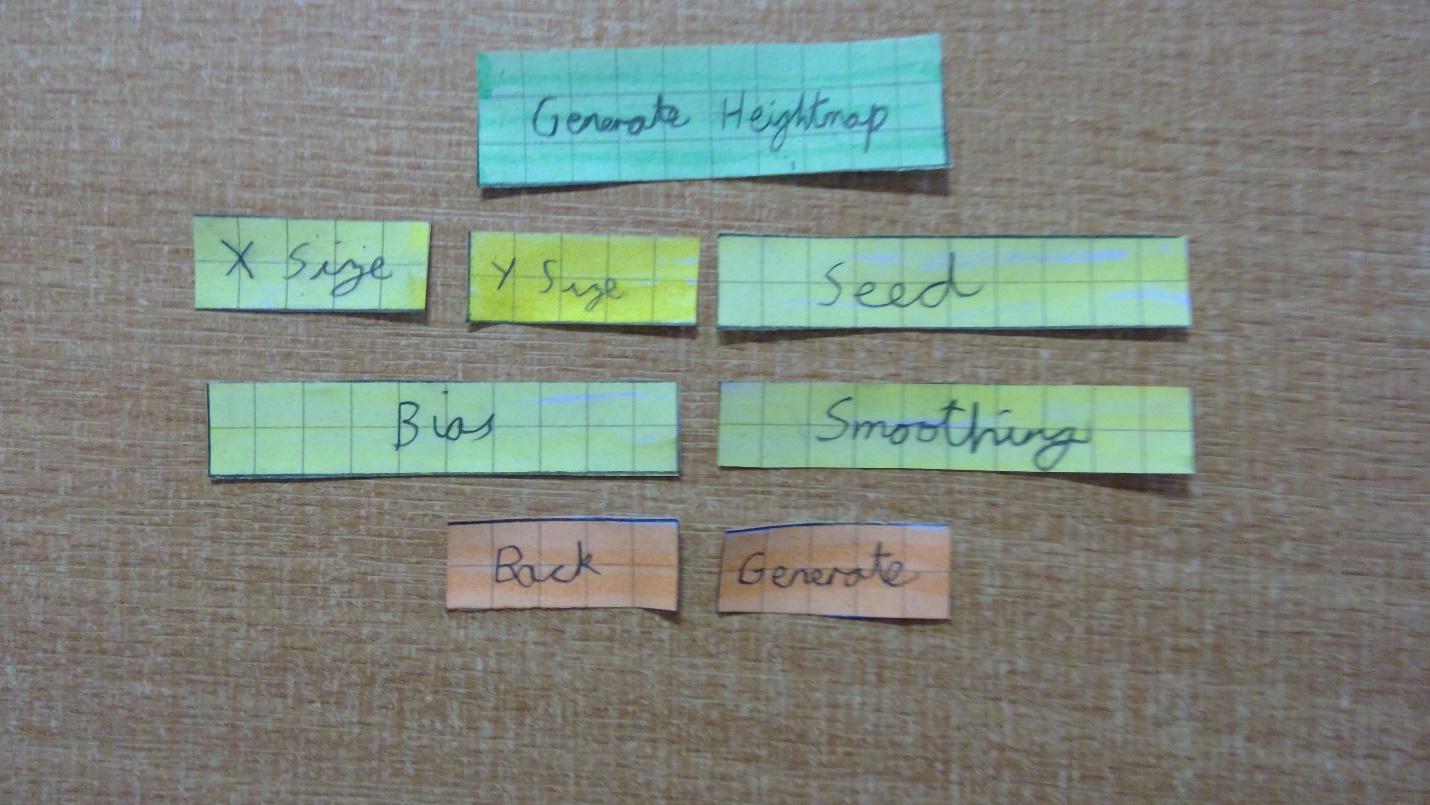
### Generate Heightmap

When designing the paper version of this GUI, I was able to reuse all components of the “Generate Terrain” GUI, with the only new component being a green “Generate Heightmap” constant. The components can be seen pictured below.



After procuring the components, I created a mock design to show Programmer A. This design had “Generate Heightmap” at the top, followed by “X Size” and “Y Size” on the same line, then the other text entry fields in a vertical column below. After the last text entry field, were the “Back” and “Generate” button, both on the same row.

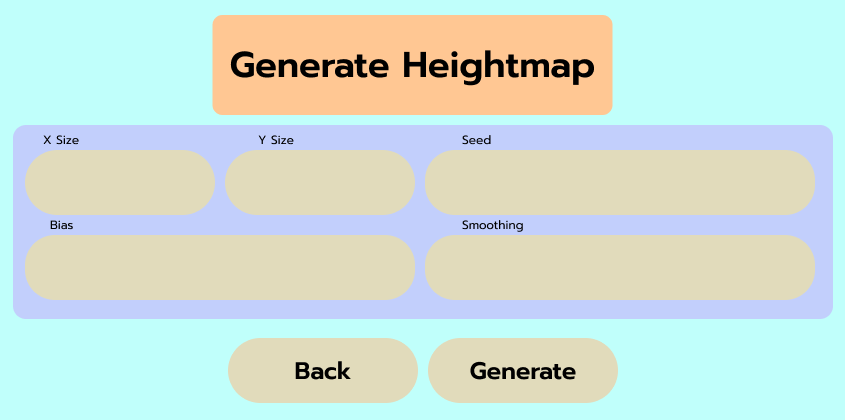
I then took the design to Programmer A for feedback. He said that the design “was very usable”, however it did not mesh well with the “previously established design cues”. In order to make the menu match, we positioned “Generate Heightmap” again at the centre, with “X Size”, “Y Size” and “Seed” all inline on the row below. This was then followed by “Bias” and “Smoothing” on the row below, with “Back” and “Generate” on the final row. The completed layout can be seen below.



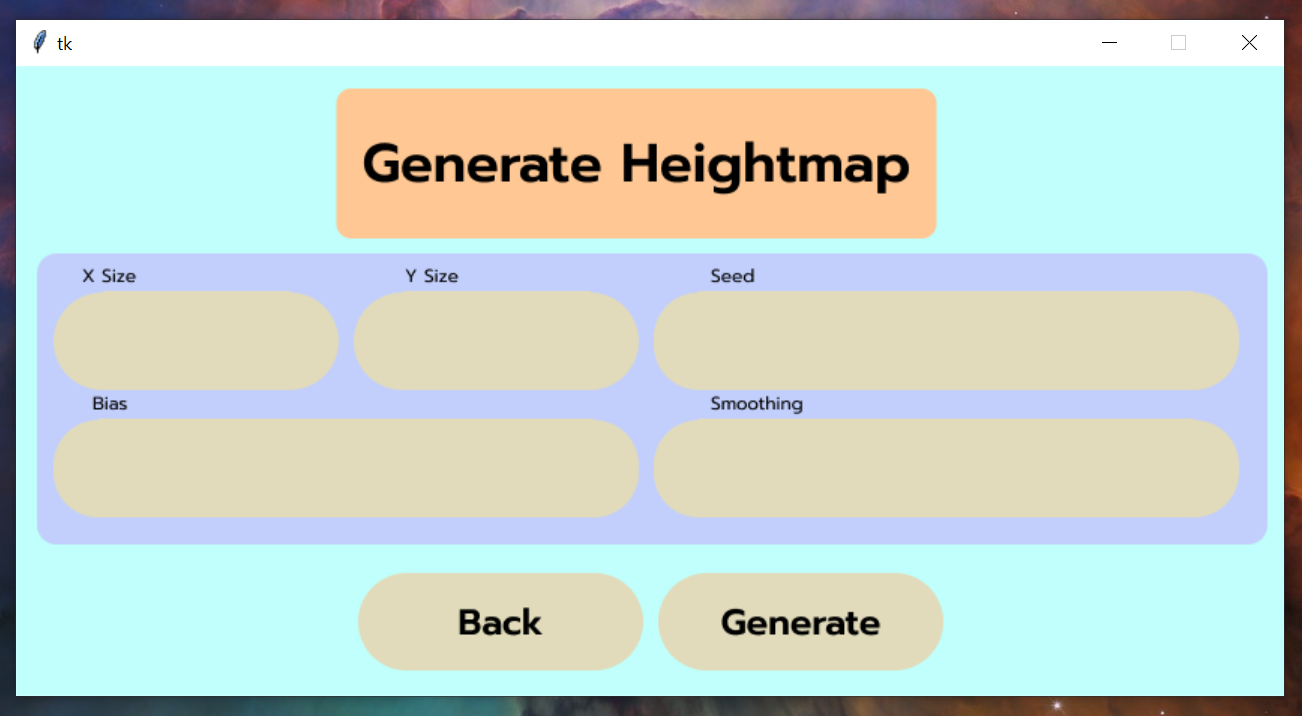
After finalising the layout with Programmer A, I started to implement the design in Figma ([www.figma.com](http://www.figma.com)). I utilized the same colours as with the previous two GUIs, as well as the same fonts. This will help to create a more natural looking experience for the user, as the screen won’t rapidly change colour to be a completely different look.

The shapes of all of the components are also identical, bar the purple background around the text entry fields. This was changed to be a rectangle with rounded corners, as opposed to a rectangle with a “chunk” taken out of one corner.

After finalising the design in Figma, I showed Programmer A. As all the colours were already the same as for the previous GUI, as well as the fonts and shapes, he was fine with it. The completed design can be seen below.



I then again exported the GUI to Tkinter using the same tool as last time, Tkinter Designer ([TestTest4253/GUI-Designer (github.com)](https://github.com/TestTest4253/GUI-Designer)). The final GUI can be seen below.



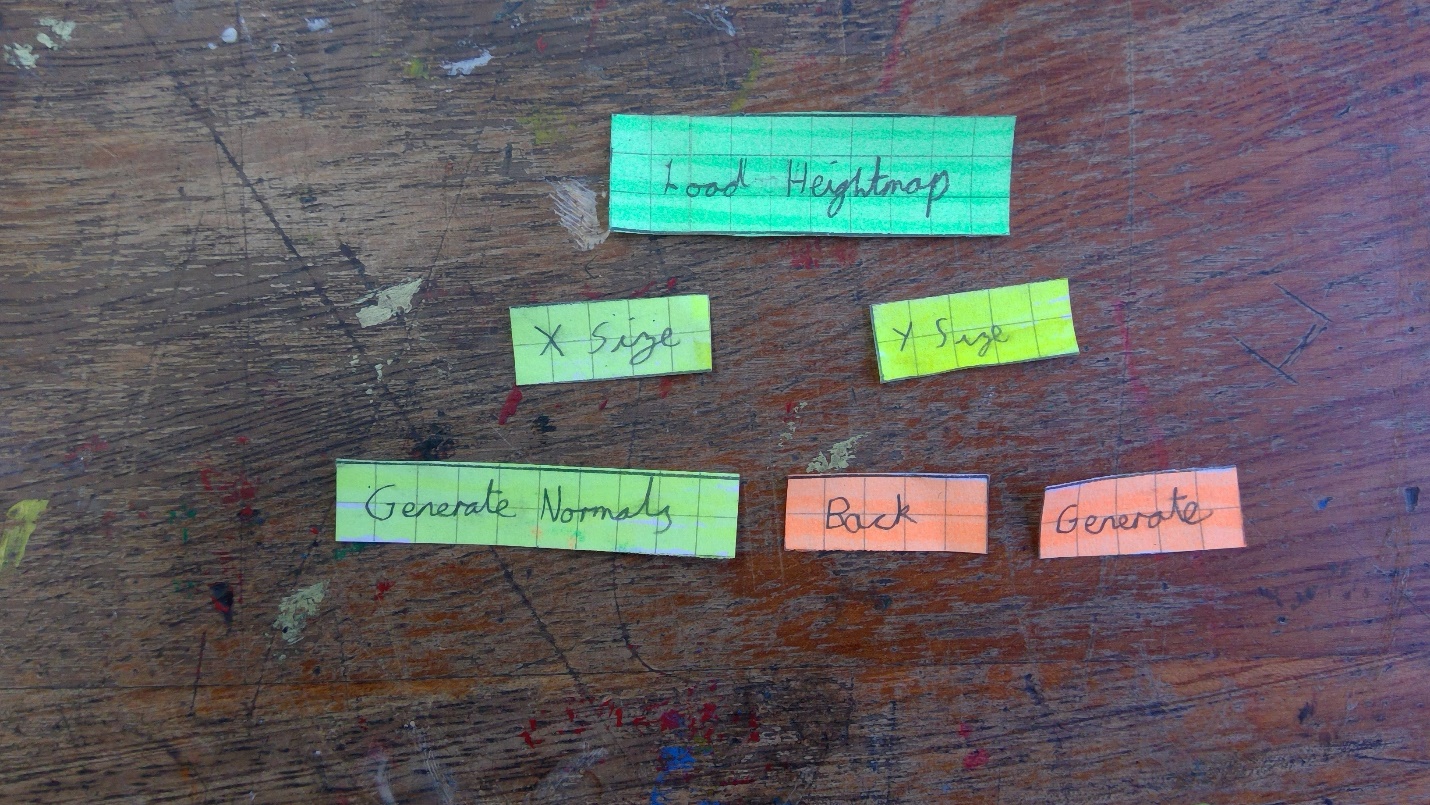
### Load Heightmap

Similarly to Generate Heightmap, Load Heightmap only needs one more component/element, this being the static “Load Heightmap” text. The other components are the “X Size”, “Y Size” and “Generate Normals” text entry fields, as well as the “Back” and “Generate” buttons. The elements can be seen pictured below.



I then produced a design to show to Programmer A, with “Load Heightmap” again being at the top, with “X Size”, “Y Size” and “Generate Normals” in line on the row below. Underneath this the “Back” and “Generate” buttons could be found.

I took this design to Programmer A for feedback. He stated that the design was “ok”, but that it would look better if it was more similar to the design of the Generate Terrain screen, where the buttons are in line with the last of the text entry fields. To do this he suggested increasing the size of the “X Size” and “Y Size” text entry fields to be the same width as all of the other elements. This design (bar the increased width of text entry fields) can be seen below.



After finalising the layout, I recreated it in Figma ([www.figma.com](http://www.figma.com)). I used the same colours as the previous designs, and the same buttons sizes and designs, other than for “X Size” and “Y Size”, which both are increased in the X – Dimension.

The purple background is the same design as that on the Generate Terrain menu, with a rectangular “chunk” taken out of the bottom right. The only difference is that the height of the purple rectangle has been reduced by one element.

After finalising the design, I received Programmer A’s feedback. He felt that the design was very suitable, and fitted with the rest of the screens. This design can be seen below.

Graphical user interface, application

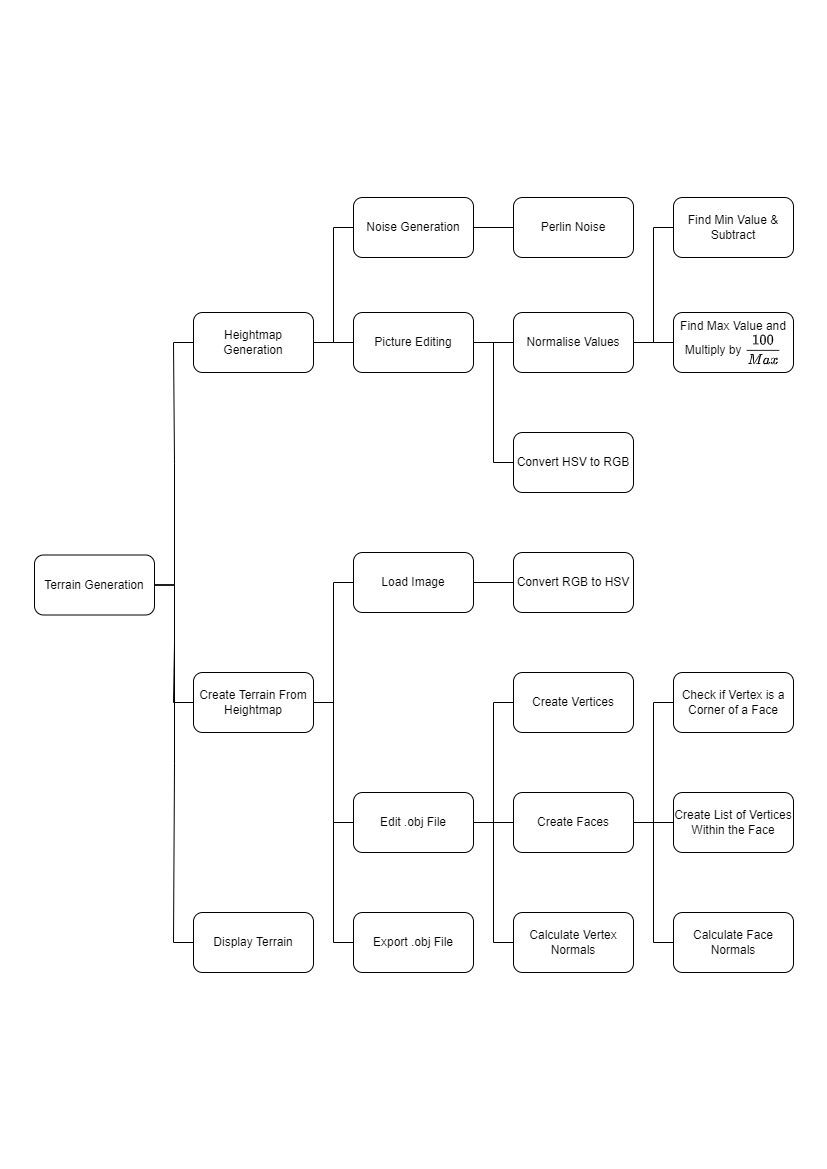
Description automatically generated

I again converted the Figma design to Tkinter using Tkinter Designer (<https://github.com/TestTest4253/GUI-Designer>). The completed GUI can be seen below.

Graphical user interface, text, application, chat or text message

Description automatically generated

## Decomposition



This method of decomposing the problem allows me to easily work on the three main components one at a time.

This will be imperative when I am working on the terrain editing (see “Create Terrain From Heightmap” on the diagram), as having separate heightmap generation will prevent me from accidentally breaking the Perlin Noise, for example.

As the terrain will also be generated from a heightmap, it will be easy to implement the functionality of using a predetermined heightmap, as this image will simply be a parameter that could be passed in. Whilst this is not on the success criteria table, Programmer A (the stakeholder) has requested this feature.

**Features of .obj Files**

The .obj file is very easy to work with, as it is plain text. Each line will start with an identifier, which tells the rendering program if it is a vertex, vertex normal or a face, among other data types. The symbols for the data types I will use are shown below:

V – vertex

Vn – vertex normal

F - face

**On the Separation of Perlin Noise**

Having Perlin Noise as a separate module to everything else will allow me to quickly replace it if required. This will be very useful as for testing I will be able to use a function that will give me a pre-determined output for each coordinate, allowing me to more easily check that my terrain generation from heightmap is working. This will make testing much easier, and will potentially allow me to more easily bugfix the terrain generation components.

In addition to this, a testing function with a predetermined output will make it much easier to test that the image is being edited correctly. This is necessary as the value from the noise (which will represent the height of the terrain) will be used as the input for one of three values (Hue, Saturation & Value in HSV or Red, Green & Blue in RGB) that is given to each pixel in the image. If this is incorrect in the image editing segment, then it may result in colour banding or artefacts.

Colour banding is when there is a noticeable shift of colour in pixels along a line or edge in an image (see image below), resulting in a “band” appearing in the image. This could cause issues in my program, as the terrain generated from the image would also feature this band, in the form of a vertical face that would run along the terrain. This may not be as noticeable due to the low vertex resolution of the terrain that will be generated, however it is ideal if this is not the case, especially if the program is used in order to generate heightmaps.

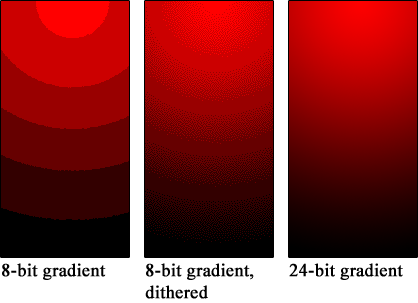


Image from Wikipedia: <https://en.wikipedia.org/wiki/Colour_banding>

Artefacts are also another issue that would negatively effect the production of terrain from an image. These are visual errors in digital graphics, an example from Windows XP can be seen below. Similarly to colour banding, these result in very sudden shifts of colour in pixels adjacent to each other. Due to the nature of terrain, this is not ideal as the terrain should be as smooth as possible, and artefacts could result in columns of terrain, or similar features.

Graphical user interface, text

Description automatically generated

An Artefact on Windows XP, taken from Wikipedia: <https://en.wikipedia.org/wiki/Visual_artifact>

Having Perlin Noise as a separate function also enables the type of noise to be easily changed, by simply replacing the algorithm (assuming that they both have the same outputs and function names). For example, the noise could be swapped out to Worley noise, in order to produce varying terrains including a variety of different features (a comparison of Worley Noise and Perlin Noise can be seen below).

A picture containing leopard

Description automatically generatedA picture containing outdoor, night sky

Description automatically generated

Left: Worley Noise, take from Wikipedia: <https://en.wikipedia.org/wiki/Worley_noise>  
Right: Perlin Noise, generated by my program

The implementation of Perlin Noise I will use is a direct translation of Ken Perlin’s original implementation into Python. The noise will be used within a class (NoiseFactory) which will have getters and a function for octave noise on top of the Perlin Noise. Octave noise is based off of Hugo Elias’ definition of persistence (Source: [Perlin Noise (arendpeter.com)](https://www.arendpeter.com/Perlin_Noise.html)).

**A picture containing shape

Description automatically generated**

NoiseFactory UML Diagram

The functions and variables will be explained within the algorithms section on Perlin Noise, bar list[]. This array allows the values generated by the noise algorithm to be stored, and in turn accessed at any time. This should help to optimise the program, as the noise only has to be generated once, as opposed to other systems that may result in noise being generated for every point multiple times.

**On the Separation of Normalising Values**

When the Perlin Noise algorithm outputs values for all pixels, they will likely be over a wide range of values. These values can include negative numbers, have no negatives, and extend to much higher numbers in certain circumstances. This is not ideal, as if I use a HSV image to store the value for each vertex (where each vertex is represented by one pixel) then I will only have 100 different colours to use, assuming that only the Value data is used, as this is a float between 0 and 100 inclusive.

In order to have the most detail possible in the terrain whilst still maintaining a reasonable file size, I need to make sure that every value in the output is between 0 and 100. In order to do this, I will take the smallest value within the array of values produced by the noise algorithm and subtract it from all values. Then I will divide every value by the maximum value in the array, and multiply by 100, resulting in the values being normalized.

Having this as a separate function will allow me to quickly change this to a larger range, for example 0 – 255. This is useful if I want to be able to store more detail within the image so change to use RGB. Furthermore, it will also allow me to change the normalisation to use other values within the HSV declaration, further increasing the versatility of the program.

**On the Separation of Normal Calculation**

In order to use certain features in .obj wavefront renderers, such as Phong shading, vertex normals are required. These are values that allow the renderer to smoothly interpolate the lighting used, which can result in a model looking rounder than it actually is (see below). Having this as a separate function allows me to disable or enable this feature at will, in order to create low-poly terrain or terrain that appears more detailed.

A picture containing logo

Description automatically generated

Flat shading (left) vs Phong shading(right). Taken from Wikipedia: <https://en.wikipedia.org/wiki/Phong_shading>

## Algorithms

### Indexing of Data

In order to be able to easily access all data, it needs to be stored in a standardized format that is optimised for the procedures that it will perform.

Vertices will be stored in an array of length . Each vertex will itself be an array of length three, containing the data of coordinate and noise value in the form [x, noiseValue, y]. Each vertex array will be have the index of within the vertices array. When the vertices are in the .obj file they will be indexed at their vertices index plus 1.

Faces will be stored in an array of length . Each face will itself be an array of length three, where each element in the face array is a vertex (which subsequently has three values, as described in the above paragraph). When they are in the .obj file, they will be indexed at their faces index plus one.

Face normals will be stored in an array of length . Each face normal will itself be an array of length three, where each element is the X component, Y component and Z component respectively. These will not be contained within the obj file, but instead used for creation of vertex normals.

Vertex normals will be stored in an array of length . Each vertex normal will itself be an array of length three, where each element is the X component, Y component and Z component respectively. When they are in the .obj file, they will be indexed by their index in the vertex normals list plus one.

### Calculating Vertex & Face Normals

In order to calculate the vertex normals, the face normals first have to be calculated. In order to use this, we take the coordinates of the vertices that make up the face. We can then calculate the vectors of two edges of the triangle, and take the cross product of these. This can then be turned into a unit vector by dividing each component by the magnitude of the normal. See a worked example below.

**Example:**

Vertex 1: (0, 1, 2)

Vertex 2: (0, 1, 1)

Vertex 3: (0, 0, 1)

Edge 1:

Edge 2:

Face Normal:

After all face normal have been calculated, we can then iterate through every vertex and store all the faces that it is part of. The normals of these are then added together in a vertex normal array, where array[0] is the x coordinate, array[1] y and array[2] z. This then needs to be turned into a unit vector by finding the magnitude and dividing every component by this.

Finally, as we are generating terrain and this will always be on the floor, we need to make sure that the y component (array[1]) is positive. If negative, we need to take every component and multiply by -1. This is the final vertex normal for the point.

**Pseudocode:**

Text

Description automatically generated

Text

Description automatically generated

### Creating Faces

In order to create faces, we first have to find out which vertices have space to be in the lower left corner of two different triangles (see circled corners on diagram below). We can then find out which vertices that are also on available faces, and add an element to a list of faces that contains all 3 of these vertices.

A picture containing indoor, orange

Description automatically generated

The reason we check that each vertex is part of a triangle is because the terrain will be made up of lots of triangles. This allows us to have more detailed and natural looking terrain than if squares were used, whilst also maintaining the low poly style of terrain that is desired.

Finding the other vertices in the face is simple, as all vertices are stored in a one dimensional array, with an index of . This means that all we need to do is take the index of the vertex in the list , and add or subtract a specific number. For the vertex above, we simply subtract , and for the vertex to the right we add 1. We can then return the vertex in that index in , and use this in the face.

This is a much simpler method than the alternative, which involves manually creating the vertices added to the face by creating them from the X and Y coordinates of the first vertex, and then finding the value of the noise at these new coordinates. This could result in rounding errors (depending on the parameters used), which would result in a much more complicated system that occasionally will return errors and break the program.

**Pseudocode**

Text

Description automatically generated

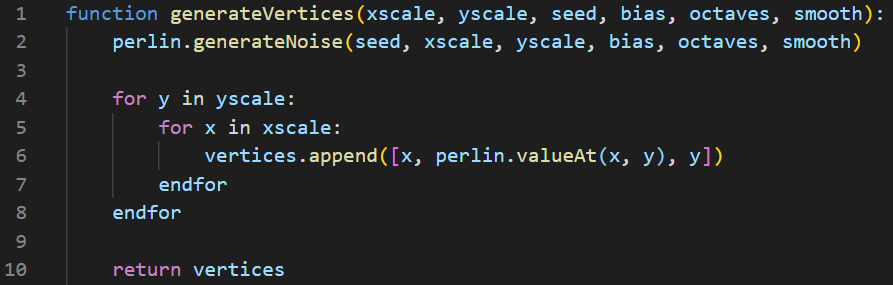
### Creating Vertices

In the pseudocode, I will reference a library “perlin”. This is the noise class that I will use (see Perlin section in Pseudocode), though it can be replaced with a different noise class with the same output.

In order to create a list of all vertices, we first need to calculate the values of the noise at each pixel. Once this is done we can then run through two nested loops and create all necessary vertices. We will only create the vertices that make up the top face, as these are the only vertices that would be seen if the terrain is used in a game, and adding other vertices would slow down both the creation and viewing of terrain, as the number of faces would be slightly over doubled.

When we insert the vertices into the list however, we need to make sure that they are inserted in a way that gives them the correct index, which in this case is . If this is not done correctly, other algorithms will break or have to be much more complicated in order to take into account the different order.

**Pseudocode**



### Creating Heightmap Image

To handle images in my project I will use the Python library Pillow ([python-pillow/Pillow: The friendly PIL fork (Python Imaging Library) (github.com)](https://github.com/python-pillow/Pillow)). This allows me to create, edit and access images. In the pseudocode I shall use functions that it provides, and reference them as pillow.FunctionName().

I will also reference a library “perlin”. This is the noise class that I will use (see Perlin section in Pseudocode), though it can be replaced with a different noise class with the same output.

In order to create the image used as a heightmap, we first need to generate the noise. In this example I will use Perlin Noise, as it is the one I will use, but ideally every type of noise can be made to return in the same manner and with the same indexing. For example, “perlin” could be replaced with “worley” and it would still function in the same way, just creating a different looking heightmap.

After the noise has been calculated, we then need to set the pixel in the image which represents a coordinate to the value of the noise at that coordinate. This can be done by drawing a square at position (X, Y) which has side-lengths of 1. As I am using HSV to store the image in greyscale, I then set the colour of this square to (0, 0, noiseValue).

The HSV image must be converted into RGB before it can be saved. Finally, it can then be saved.

A screenshot of a computer

Description automatically generated with medium confidence

### Generating Noise (Perlin Noise)

In my initial tests, I will use Perlin Noise so the explanation and pseudocode here will be based off of Perlin Noise, however it is interchangeable with another noise type, such as Worley Noise or even random noise. I shall reference the following UML diagram in the explanation.

**A picture containing shape

Description automatically generated**

The permutation array stores all numbers from 0 to 255 inclusive, in a random order. In this case, the order will be the same as Ken Perlin’s original implementation as that was shown to give good results. When the class is initialised, the array p will be populated with the numbers from permutation, with every number being included twice.

When the noise is generated, the algorithm checks which 1 by 1 “grid cell” the point is within. It then calculates the relative position within this grid cell. A vector for each corner of the cell is pseudo-randomly generated (this results in the same grid cell corners always having the same vectors associated with them, assuming that the same seed and permutation table is used). The algorithm next dot products the vector and the displacement gradient. Finally, we interpolate the dot products in order to return the value at that point.

Octave noise allows for smoother output images. In order to create octave noise we need to generate Perlin Noise with a number of different frequencies and amplitudes. In order to scale these correctly, I will use Hugo Elias’ definition of persistence:

In this definition, is the octave being calculated. Persistence is a variable that can be tweaked in order to control the smoothness of the output.

A picture containing calendar

Description automatically generated

A comparison of different values of amplitude, taken from <https://www.arendpeter.com/Perlin_Noise.html>

**Pseudocode** (For Octave Noise)

**A screenshot of a computer screen

Description automatically generated**

## Inputs & Outputs

**Generate Terrain**

The program will take no input files, however the user will be able to input data through a GUI menu to the program, from which the terrain will be generated.

The inputs will be:

* X Scale
* Y Scale
* Seed
* Bias
* Smoothness
* Generate Normals
* Location to save to
* File name to save to

The program will then output a file, which will be a .obj file containing the generated terrain.

**Generate Heightmap**

The program will take no input files, however the user will be able to input data trough a GUI menu to the program, from which the heightmap will be generated from.

The inputs will be:

* X Scale
* Y Scale
* Seed
* Bias
* Smoothing
* Location to save to
* File name to save to

The program will then output a file, which will be a .png file containing the generated terrain.

**Load Heightmap**

The program will take 1 input file, as well as allowing the user to be input data through a GUI menu, and will then use all data collected to generate terrain from the heightmap.

The inputs will be:

* A png heightmap
* X Scale
* Y Scale
* Generate Normals
* Location to save to
* File name to save to

The program will then output a file, this being a .obj file containing the generated terrain specified by the heightmap.

## Testing

**Generate Terrain**

As the data parsed to this will be used differently across programs, I am not able to compare the output to that of different products. However, I am able to perform some tests. These are outlined below.

|  |  |
| --- | --- |
| Task | Success Criteria |
| Integer input to X Scale | No errors |
| Non integer input to X Scale | Returns a warning of incorrect data type |
| Integer input to Y Scale | No errors |
| Non integer input to Y Scale | Returns a warning of incorrect data type |
| Float input to Bias | No errors |
| Non float input to Bias | Returns a warning of incorrect data types |
| Float input to Smoothing | No errors |
| Non float input to Smoothing | Returns a warning of incorrect data types |
| Integer input to Seed | No errors |
| Non integer input to Seed | Returns a warning of incorrect data types |
| Boolean input to Generate Normals | No errors |
| Non Boolean input to Generate Normals | Returns a warning of incorrect data types |

On top of these, if the same inputs are given to the program on two separate occasions, the same terrain should be generated in both situations.

**Generate Heightmap**

As the data parsed to this will be used differently across programs, I am not able to compare the output to that of different products. However, I am able to perform some tests, which are outlined below.

|  |  |
| --- | --- |
| Task | Success Criteria |
| Integer input to X Scale | No errors |
| Non integer input to X Scale | Returns a warning of incorrect data type |
| Integer input to Y Scale | No errors |
| Non integer input to Y Scale | Returns a warning of incorrect data type |
| Float input to Bias | No errors |
| Non float input to Bias | Returns a warning of incorrect data types |
| Float input to Smoothing | No errors |
| Non float input to Smoothing | Returns a warning of incorrect data types |
| Integer input to Seed | No errors |
| Non integer input to Seed | Returns a warning of incorrect data types |

On top of these, if the same inputs are given to the program on two separate occasions, the same terrain should be generated in both situations. The heightmap should also be able to be opened in various other image generation programs.

**Load Heightmap**

The data parsed to this will be used similarly between image generation programs. Due to this, the output of this function can be checked against the output that other, similar programs will return. We can also perform the following tests.

|  |  |
| --- | --- |
| Task | Success Criteria |
| Integer input to X Scale | No errors |
| Non integer input to X Scale | Returns a warning of incorrect data type |
| Integer input to Y Scale | No errors |
| Non integer input to Y Scale | Returns a warning of incorrect data type |
| Boolean input to Generate Normals | No errors |
| Non Boolean input to Generate Normals | Returns a warning of incorrect data types |

On top of these, if the same inputs (both file and data supplied via the GUI menu) are used on two distinct occasions, then the output should be the same on both of these occasions.

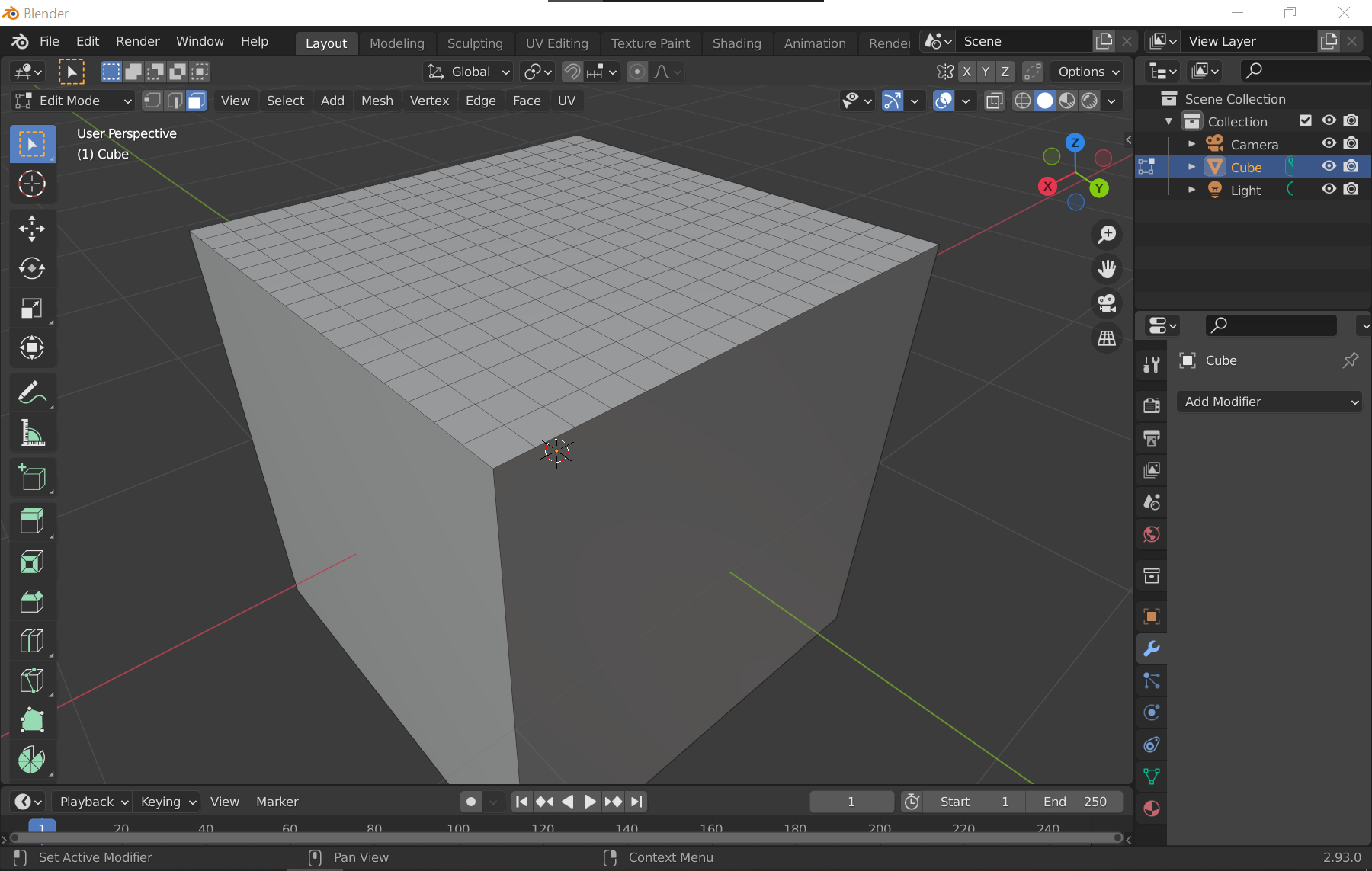
# Development & Testing

### 1st Prototype – Editing the File

For my first prototype I decided that I would initially take an input .obj file of a subdivided cube (ie a cube that had one face with a matrix of vertices), which would allow me to experiment with the .obj file system, and how to edit it.

The first step was to produce the subdivided cube in a 3D modelling program, and for this I chose to use Blender ([blender.org - Home of the Blender project - Free and Open 3D Creation Software](https://www.blender.org/)), as it is free and open source. On top of this, it allows the programming of scripts in Python, which would allow me to quickly prototype certain features if I deemed it necessary.

In order to create the cube, I created a new project within Blender. I then selected just the top face and applied several subdivides, until I was happy with the resulting resolution. This was a 16x16 grid of faces, resulting in a 17x17 grid of vertices that I would be able to manipulate with my program. This cube can be seen imaged below.



After creating the cube, I was then able to export the cube as “test.obj”, which I would then be able to access in my code.

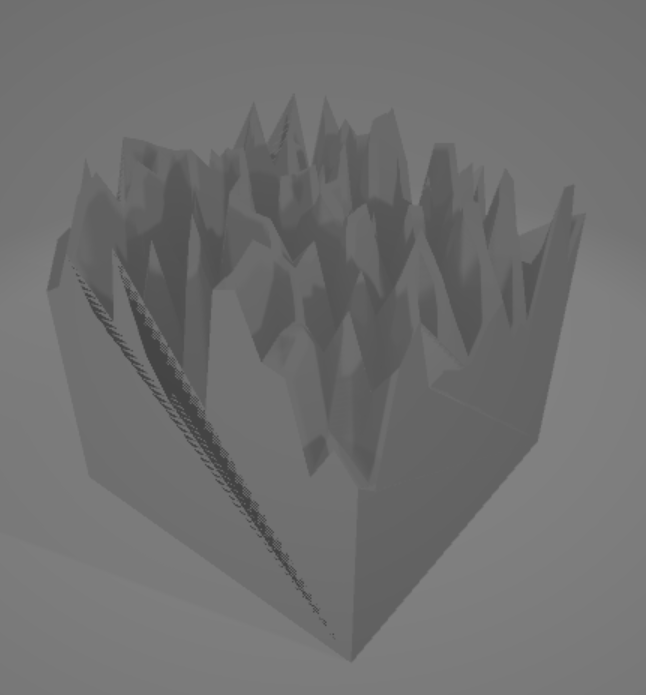
As I would be reading through each line in the original cube, inside my function I open the file, then loop through every line. This is very simple to do, as the reader will create a list, where every new line is a new item, so I can in turn simply use a for loop to iterate through every item.

A picture containing text

Description automatically generated

As my initial test is only to try editing .obj files, I will only be changing the height of vertices on the top face. In order to check if the line being operated on in the loop is representing a vertex, I simply check the first two characters in the line. If they are “v “, then I know that the line is representing a vertex.

The output when viewed in 3D Viewer (default program in Windows 10/11) can be seen below.



The darker bands that can be seen are caused by faces overlapping. This is due to Blender creating triangular faces, which always feature the bottom right corner of the face they are located on as a corner. As the heights of the vertices are randomly chosen without smoothing, there is a chance that a face will pass over another face. After Perlin Noise has been correctly implemented and only the top surface is generated, this will not be an issue as the triangles will not overlap and there will be no side triangles to overlap.

The 1st prototype was very useful, as it developed my understanding of .obj files and how to edit them. This understanding greatly assisted when I came to write code that would generate .obj files of varying sizes automatically.

### 2nd Prototype – Perlin Noise

After I confirmed that the editing of the .obj file was successful, I translated the pseudocode for Perlin Noise found on Wikipedia ([Perlin noise - Wikipedia](https://en.wikipedia.org/wiki/Perlin_noise)) into Python. However, at the time I did not implement the algorithm correctly. This would go unnoticed for several iterations, as the cube that I was using to test it on was not large enough to show the imperfections. The main implementation can be seen below, however.

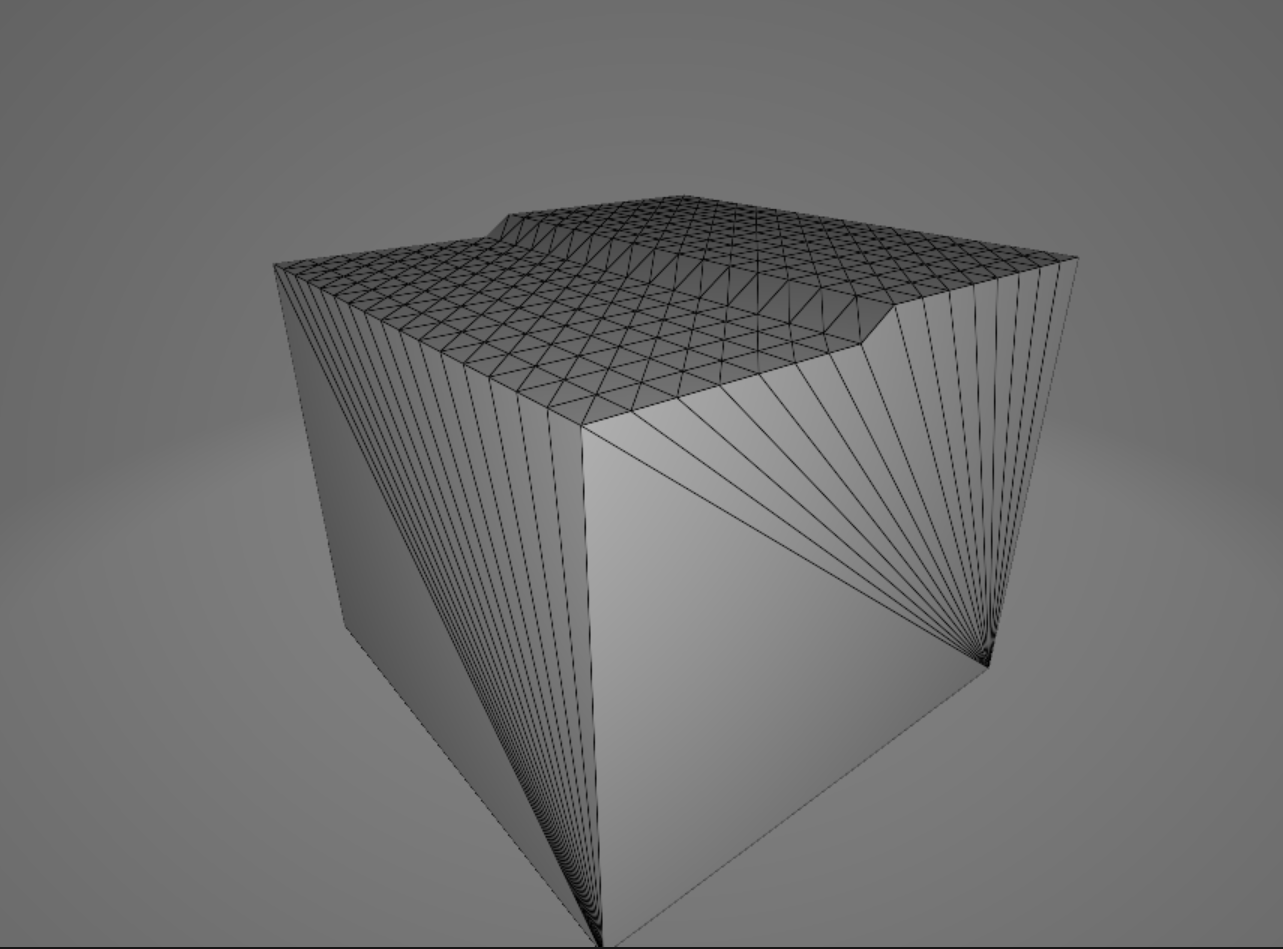
Text

Description automatically generated

This main piece of code was perfectly functional, however the “support functions” that I used in order to generate the randomness in a controllable way differed from Ken Perlin’s implementation. Instead of using a permutation table, I used an equation to generate the different values. The equation is as follows:

This equation would always generate the same value for repeated inputs, however it did not scale well, unlike the permutation table approach. In later prototypes this issue was addressed.

I was then able to change out the function that I was previously using for a call of my noise function. The output from this can be seen below, with triangles shown. This is again shown in 3D Viewer.



Whilst this 2nd prototype did not result in the finished product, or anything close to that, it did help me to further understand the Perlin Noise algorithm, despite the fact that the implementation was wrong, and the resulting noise from it did not look as good as it could do.

### 3rd Prototype – Displaying the Terrain

The next component to work on was the renderer that I would use to display the terrain. Initially, I decided that I would attempt to use PyOpenGL ([PyOpenGL · PyPI](https://pypi.org/project/PyOpenGL/)) to write my own, custom render. However, I quickly realized that this could likely be a project in it’s own right, and began to research alternatives that I could use.

Initially, I tried to use a combination of PyWavefront ([PyWavefront · PyPI](https://pypi.org/project/PyWavefront/0.1.3/)) and Pyglet ([Home — pyglet](http://pyglet.org/)). This seemed to be a good combination, however I quickly found out that the code I would have to use was practically the same as that in PyOpenGL.

After more research, I found PyRender ([Pyrender Documentation — pyrender 0.1.45 documentation](https://pyrender.readthedocs.io/en/latest/)). This library would allow me to easily load .obj files and view them, as well as packing features such as wireframe view (this helped me to verify that the surfaces were correctly shaped when lighting made it hard to see normally) and both face and vertex normal visualisers. After reading through their documentation, I was able to load in my files and show them, with a variety of parameters changed for the best experience.

In order to display my terrain, I disabled backface culling, as this produced some issues with faces not showing, regardless of normals. For debugging purposes, I have keybinds to show a wireframe view, as well as all other features offered by the renderer. The terrain shown in the 2nd Prototype image can be seen in PyRender below, in both standard and wireframe mode.

A picture containing icon

Description automatically generatedDiagram

Description automatically generated

### 4th Prototype: Combining all Functions

As I had successfully implemented all of the basic functions of the program, I now needed a menu that would bring them all together. This menu would have to allow the input of the different parameters that the various components needed, as well as sequence the functions so that they happened in the correct order, this being load original .obj, edit .obj and finally render the terrain.

As this is still an early prototype and missing many of the features that the later versions will have, I decided that a text-based input system would be more fitting than a GUI. The inputs that would be required are output file name, seed, and optionally x-shift and y-shift.

This was easy to implement, and an example use of it can be seen below, with inputs on the left and the output on the right.

Text

Description automatically generatedA picture containing text

Description automatically generated

This prototype was tested by using the same inputs as I would use to the functions individually. The outputs were then compared, and as they were the same it was deemed that the function was working.

### 5th Prototype: Generating Flat Surfaces

After the 4th prototype, the program was limited in the size of terrain that could be generated by the size of predetermined chunks that I manually created in Blender. This also had the effect of requiring every terrain “chunk” to be square, and lead to having many extra faces that weren’t required (ie the sides and bottom of the terrain, as only the top surface is changed). In order to get around this issue, I needed to create the .obj file from scratch.

Initially, this would be done with a flat plane, as adding in the noise afterwards should be easier to implement than adding it from the start.

The method that I used to do this is very similar to the method laid out in the design section, with one small change. This change is that instead of referring to a noise function, I will instead use a constant value, which will in this case be 1. This will also allow me to easily test the generation with random values if I deem it necessary.

I will then also need to implement the face generation algorithm that I also previously laid out. This will not need any changes in order to work in this way. On top of this, I will also be able to implement the normal generation algorithm, and this will also not need any changes to be functional. I will not implement normal generation in this prototype however.

After implementing this, the program initially seemed to be working, as shown in the screenshot below. The screenshot is a flat square generated by my code, viewed inside 3D Viewer.

A picture containing text, stationary, envelope, dune

Description automatically generated

As this worked, I decided to use a random integer between 1 and 3 for the height, in order to check that all of the vertices were connected correctly. However, this resulted in several bars of flat area crossing the surface, whilst the rest of the terrain looked correct. This can be seen below, again as a screenshot taken from 3D Viewer.

A picture containing light, airplane, aircraft

Description automatically generated

In order to find the cause of this bug, I added a statement inside of the .obj file creation, that would print all of the vertices in each face. The small section of this can be seen below. This output showed that the algorithm was trying to connect the points (0, 0), (1, 0) and (1, 15), when in actual fact it should be trying to connect (0, 0), (1, 0) and (1, 1). If this was done, then the algorithm would be correct.

Arrow

Description automatically generated with medium confidence

As faces are added in pairs, with each pair using a different function to find the correct vertex (one needs the vertex “above” and the other needs the vertex to the “right”), I can tell which function is causing the issue. We can take the first two lines of this output, and see that the second vertex in the first line is correct, but the second vertex in the second line is not correct, as (0, 15) is not next to (0, 0). This tells us that the issue lies within the function, which returns the vertex to the right of the one parsed into it.

After this had been found, I noticed that there was an issue with my pseudocode plan. The plan had and returning swapped values, and the correct function for has the wrong equation, as it used an instead of a . The change to can be seen below.



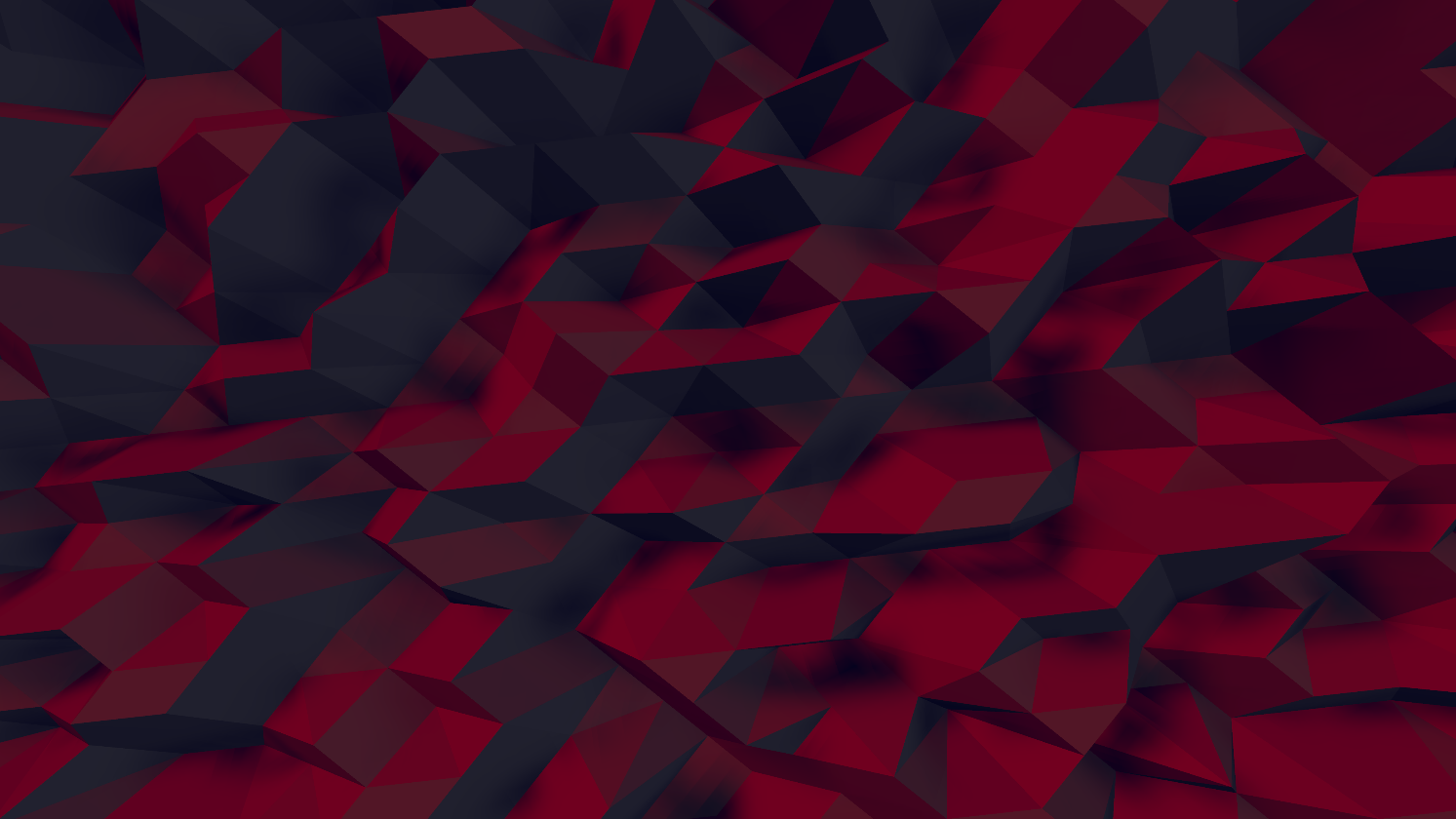
After implementing this change into my code, the bug was fixed. The code no longer had lines crossing the terrain in the output. A screenshot of the file viewed in 3D Viewer can be seen below to verify that it was successful.

A picture containing image

Description automatically generated

Overall, prototype 5 has been very successful, due to the identification and fixing of a bug that may have been harder to fix, if it was found later. An interesting piece of information is that a 16x16 piece of terrain takes 8710 bytes, whereas a 64x64 piece of terrain takes 180526 bytes. This is an increase of around 20 times. As each edge becomes four times it’s original length, I would have expected an increase of 16 times. I am not sure why this is not the case, though I imagine it is due to the amount of faces not being the same as the amount of vertices.

As a side note, this terrain can be used to create interesting looking low-poly backgrounds, as can be seen below.



### Prototype 6: Adding Normal Generation

The next feature that needs implementing is normal generation. This will be done through the implementation of the algorithm put forward in the design segment. I then also had to enable the feature within the other functions, as well as create a function to handle the selection of face normals to parse to the vertex normal generation code.

This function was relatively simple, it just looped through every vertex and found the faces that it was within. It then added the corresponding face normals to a list which was parsed to the function. The vertex normals for every vertex are then returned. The code can be seen below, as there is no pseudocode in this section.

Text

Description automatically generated

I also added this small piece of code to the function in order to calculate the normal for each face.

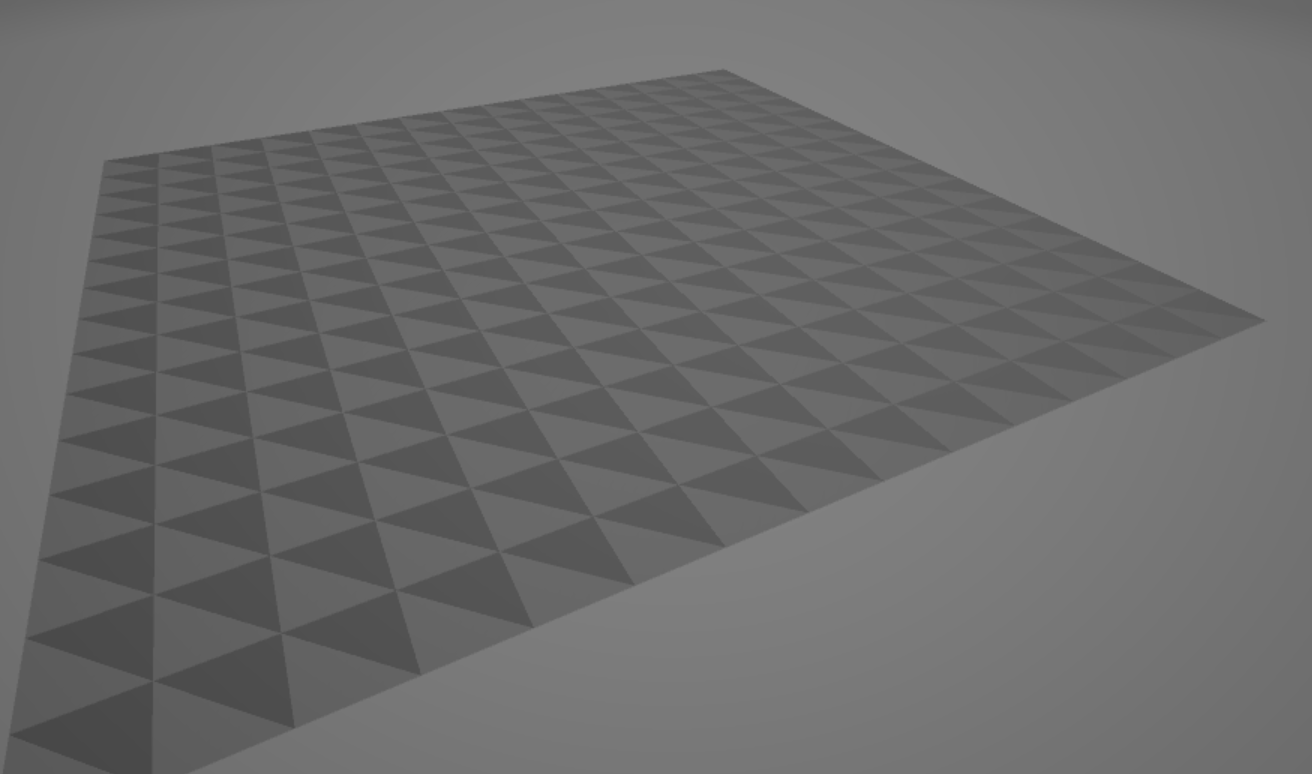
Text

Description automatically generated

Next, I had to add vertex normals to the created file. In order to do this, I created a new Boolean argument, . This was then used to select if any normal (both face and vertex) should be generated, as well as added to the file.

The simplest way to do this involved setting the default contents of a vertex normals list to none. I could then update this if normal generation was desired, and ignore it inside all functions otherwise.

Initially, this code had an issue where faces next to each other would have “flipped” normal. This can most clearly be seen when the terrain is a simple flat plane, as shown below.



When .obj files are viewed, face normals are calculated from the vertex normals of all vertices that make up the face. Due to this, I knew that the faces must therefore be defined with vertices in the wrong order. This made for a very easy change, simply editing the creation of one face. The change can be seen below:

faces.append([i, right, aboveRight]) => faces.append([i, aboveRight, right])

This then fixes all issues. The output (with normals generated) can be seen below, again shown in 3D Viewer.

A close-up of a brain

Description automatically generated with low confidence

This screenshot looks weird, however when changes are gradual, as they will be with Perlin, then the output will look much more natural.

# Evaluation

## Criteria Met

## Usability

## Issues

# Final Code