ASSESSMENT REPORT

CMP305

FRASER BARKER
1600196
1600196@uad.ac.uk

Contents

Features

Controls

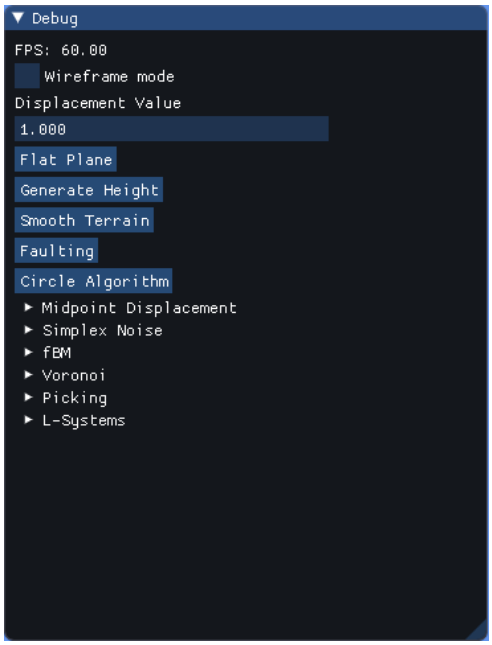
Camera

The camera can be moved around the scene using the following controls.

- W – Move forward.
- S – Move backward.
- A – Move left.
- D – Move right
- Q – Move down.
- E – Move up.
- Spacebar – Toggle camera rotation via mouse/free mouse from screen boundaries.
- Up Arrow – Tilt up.
- Down Arrow – Tilt down.
- Left Arrow – Rotate left.
- Right Arrow – Rotate right.
- Mouse – Used to rotate the camera more freely around the scene. Can be toggled via Spacebar.
- Left Mouse Button – If picking is enabled, this will displace the terrain upwards.
- Right Mouse Button – If picking is enabled, this will displace the terrain downwards.

ImGui

The application has a GUI integrated into which allows the user to alter specific values relating to the terrain in real-time rather than having to map specific functions to certain keys.

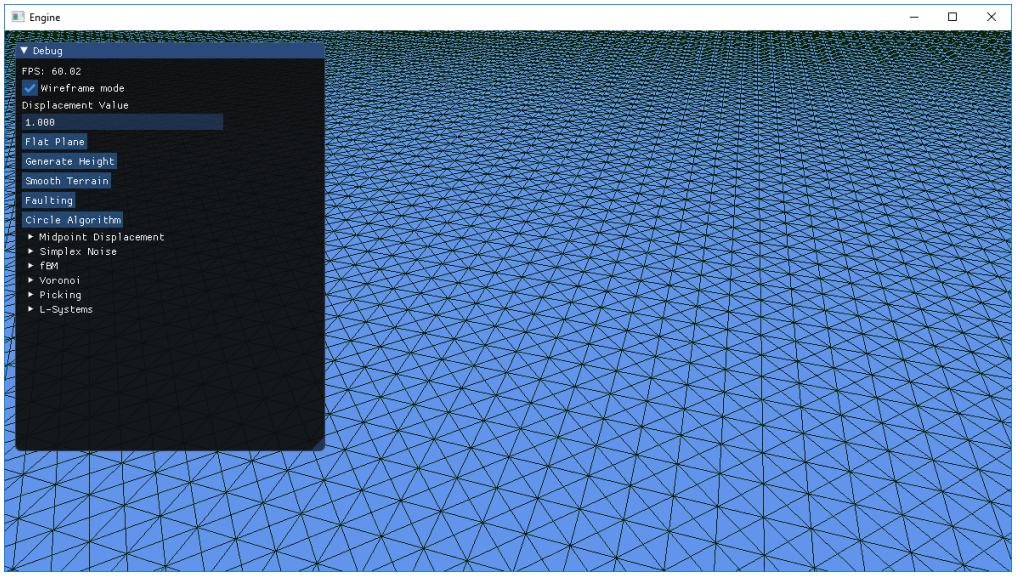


When the user has the mouse unlocked from the centre of the screen, they can interact with the above *ImGui* menu. As you can see, the menu is composed of many sections which relate to specific terrain altering functions. These functions are detailed below.

Basic Features

The ImGui menu show the current framerate of the application. This is useful as it really shows just how terrible that one function you wrote really was as you watch the framerate plummet.

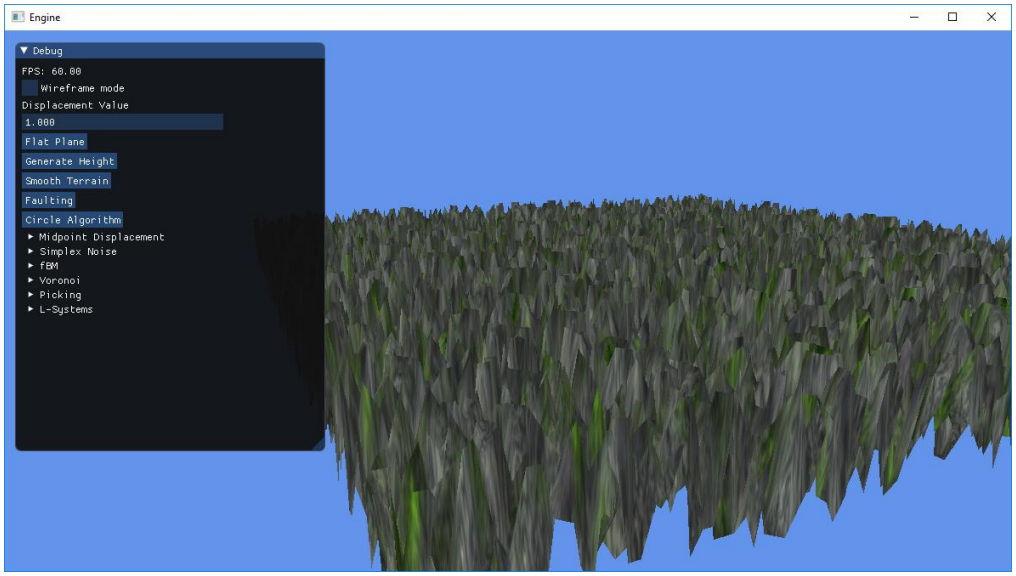
It also has a “Wireframe mode” toggle which - as the name suggests - allows the user to see all the objects in the scene in their wireframe mode.



The displacement value is a float that is passed into most - if not all - of the terrain altering functions to determine how much the terrain should be displaced by. This may have been a poor choice, as you can sometimes forget you set it to a high value when you begin experimenting with other features, which starts to cause bizarre results.

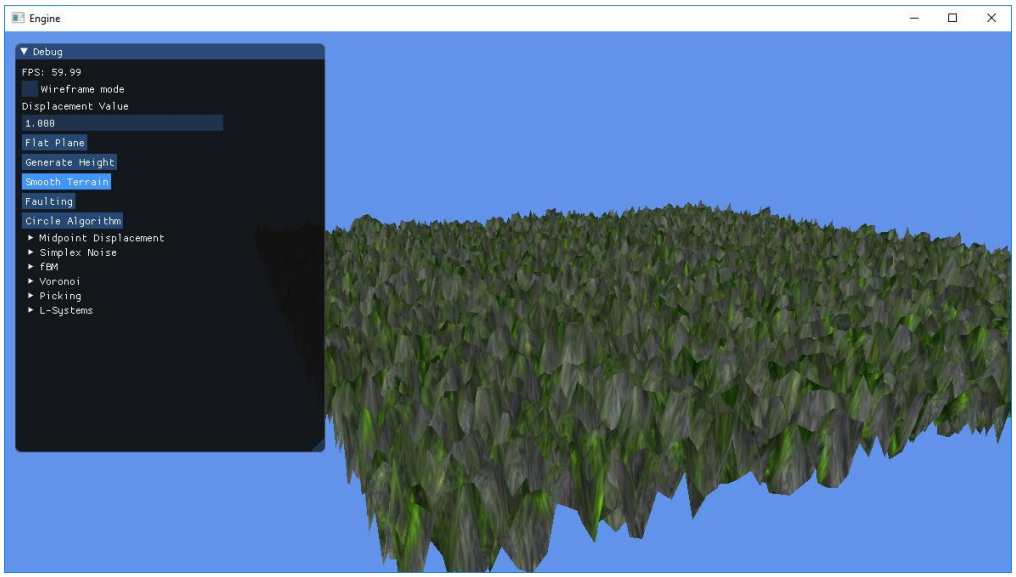
The button “Flat Plane” simply does as the name suggests, acting as a reset button it allows the user to revert the terrain back to its default flat state.

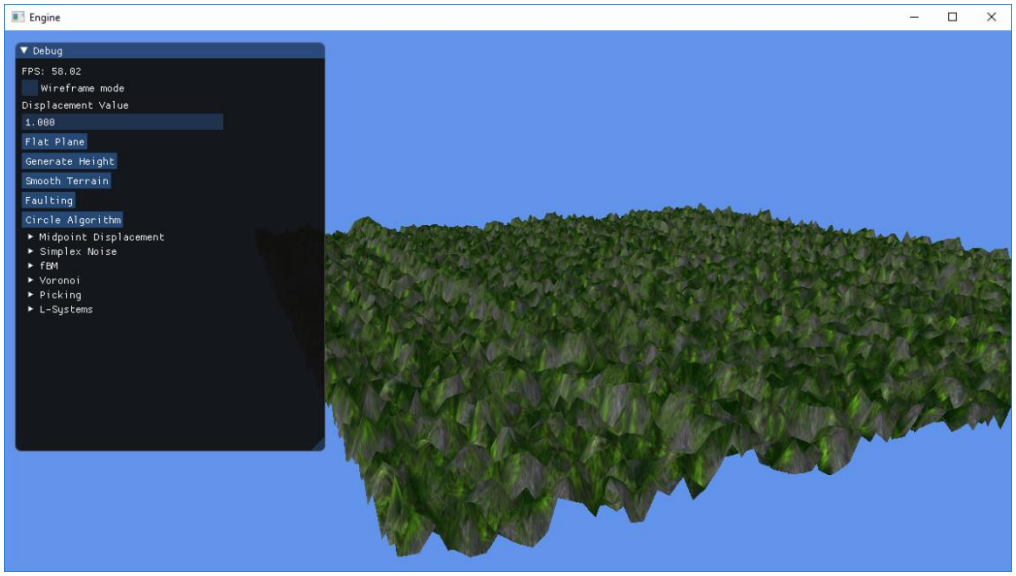
The button “Generate Height” was more of an introductory feature that kind of just stuck around. Very basic, in that all it does is loop through the terrain’s vertices array assigning random values to the y-position at each point.



Smoothing

Following the *Simple terrain smoothing* blogpost, the Smoothing function again loops through the terrain's vertices array checking the neighbours of each vertex and averages the value for the y-position. Below are two pictures of smoothing the above generated height displaced terrain.

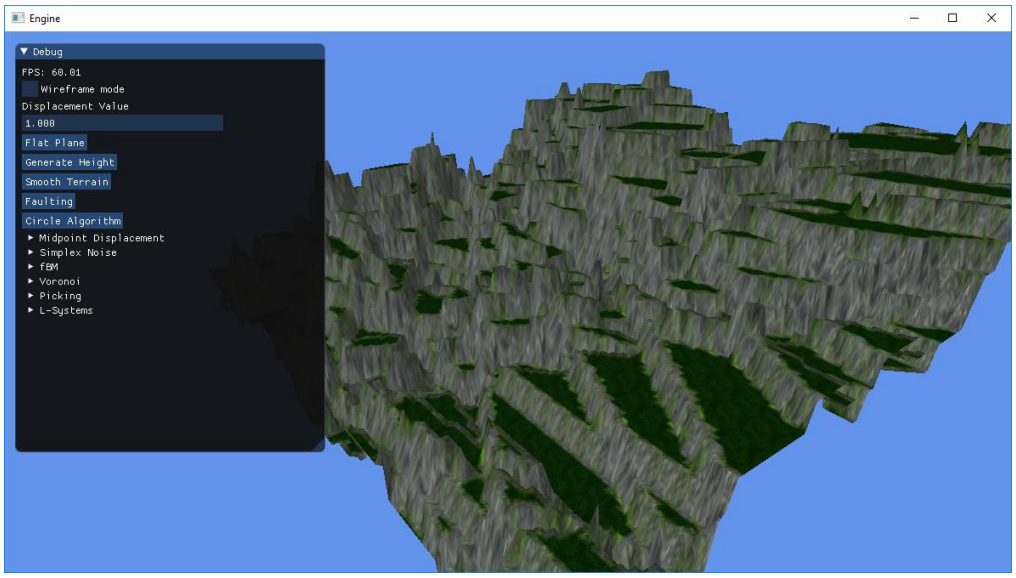




Its current implementation could do with a bit of refinement, but I'll cover this in the Critical Appraisal section.

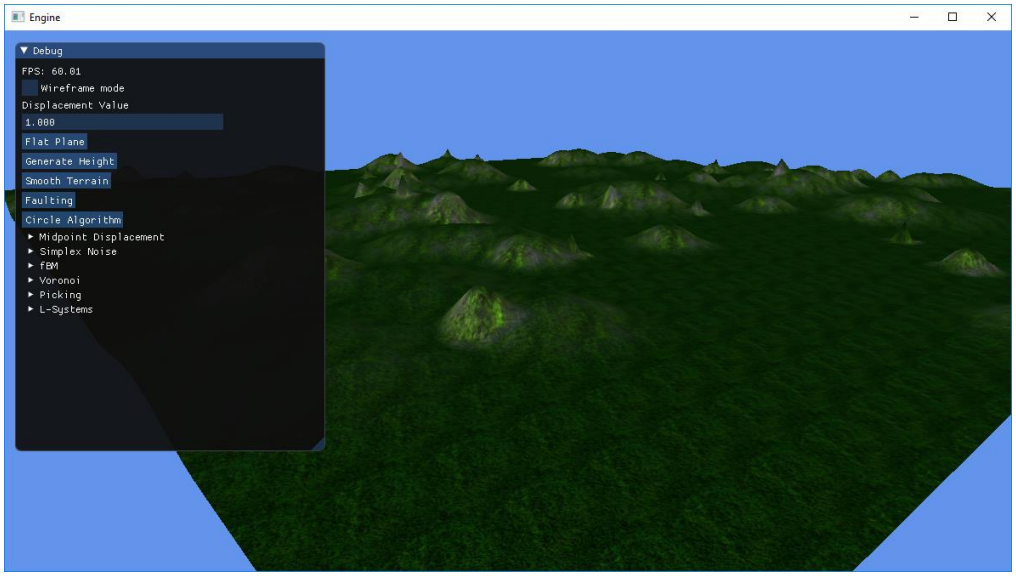
Faulting

The Faulting function utilises the fault line algorithm (as detailed in *The Fault Algorithm*). This works based on choosing two random points at each side of the plane, drawing a line between them and depending on what side of the line a vertex is, its position is either raised or lowered.



Circle Algorithm

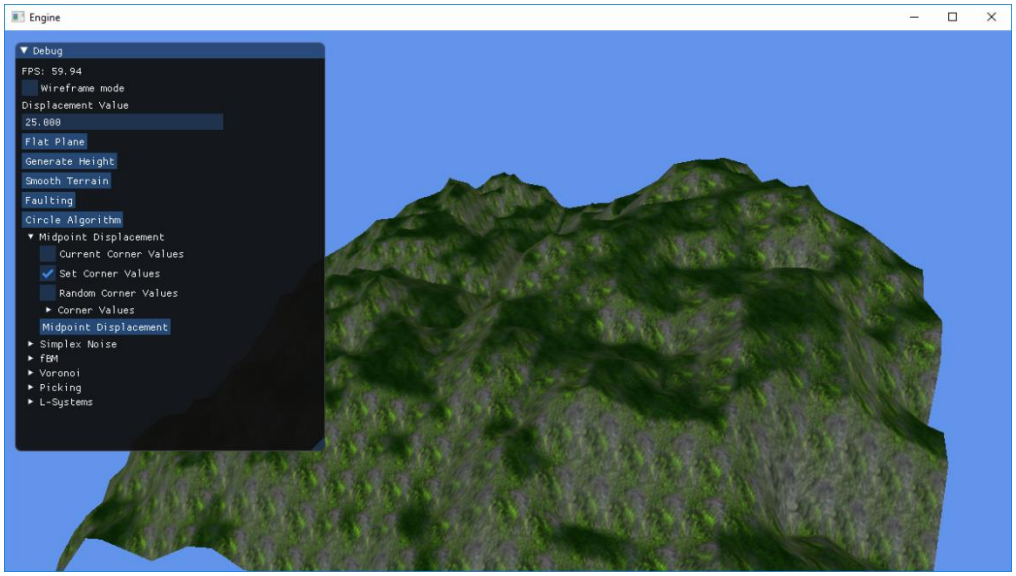
Quite possibly the best function of the application. Utilising the circle algorithm (as detailed in *The Circles* Algorithm). There are two variations of this function; a random circle algorithm and a refined circle algorithm which is used in the terrain picking function. The random circle algorithm loops one hundred times choosing random points on the plane to alter based on whether a vertex is within a circle of random radius around the random point.



Midpoint Displacement

The Midpoint Displacement function (as adapted from *Terrain Generation with Midpoint Displacement*) alters the terrains height by getting the "corners" y-position value and setting the mid-points y-value of these two "corners" to be an average of the two y-position values. This is the recursively done as it moves closer to the centre of the terrain. A good displacement value is 25.0f.

Note: The terrain **must** be an odd value dimension for this function to work. E.g. 3, 5, 7, 9 etc.

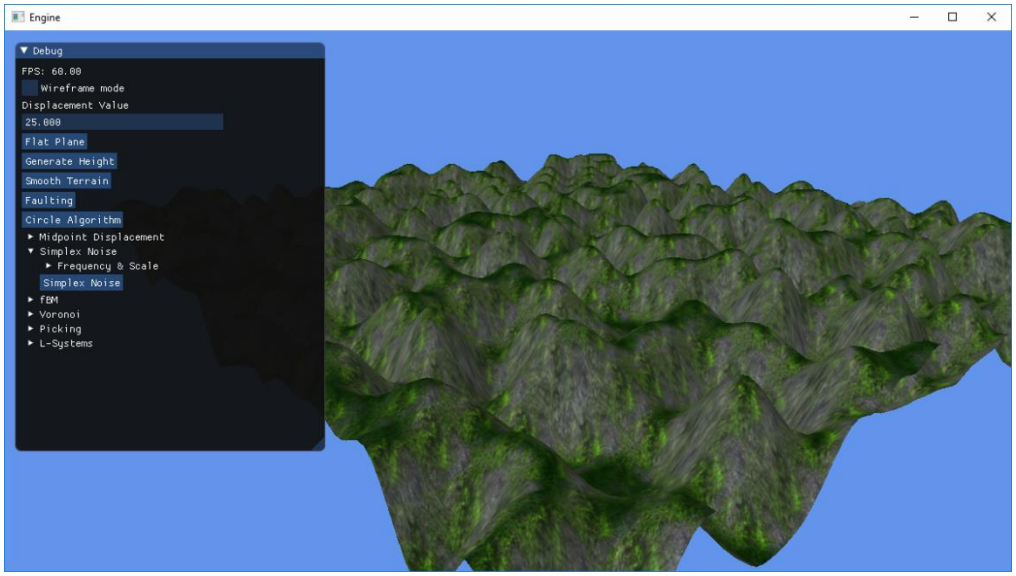


Simplex Noise

The Simplex Noise function (as detailed in *Simplex Noise Demystified*) utilises the simplex noise algorithm – a replacement for Ken Perlin's "Classic Perlin Noise" algorithm.

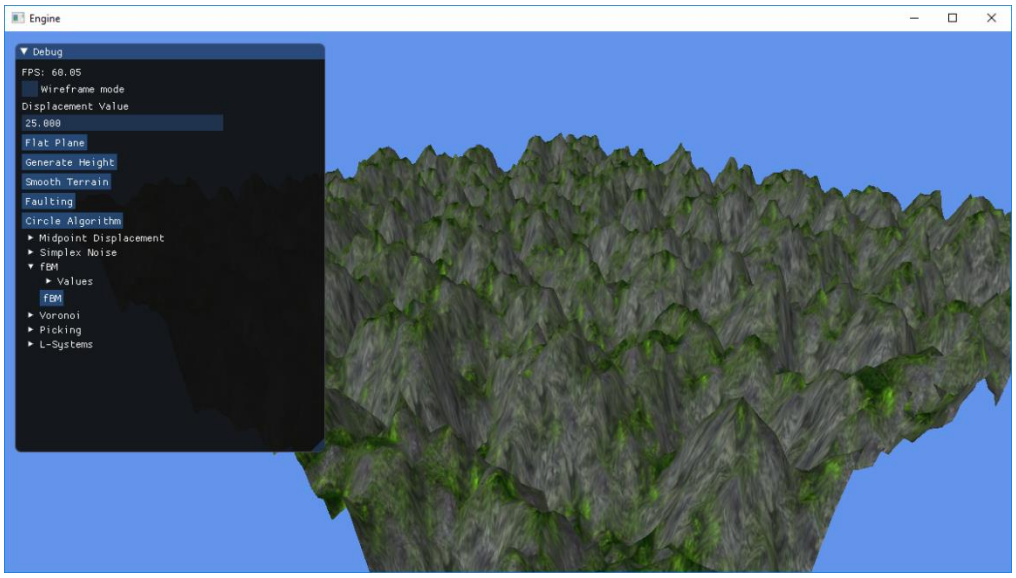
This is an improvement of Perlin's Class Noise algorithm in that it is less computationally complex, scales to higher dimensions with less computational cost and has no noticeable directional artefacts.

It works by taking in a position (in this case, the x and z positions on the plane), and calculating noise values for three surrounding "corners" – essentially where an equilateral triangle would form around the passed in position – and then sums these values up. The summed value is then scaled to get it into the -1 to 1 range. This is then done throughout the entire terrain.



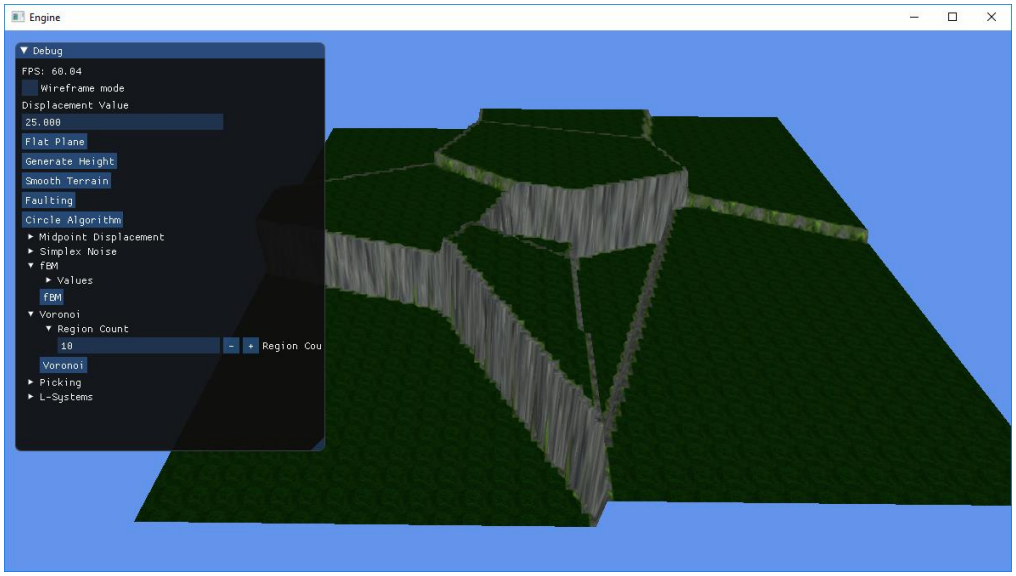
Fractal Brownian Motion

Thanks to the explanation on *Fractal Brownian Motion on bookofshaders.com*. It was understood that Fractal Brownian Motion is more of an extension of a conventional noise function, in that it would use something like Simplex Noise as a base (this application does) and merely repeats the noise function based on set values.



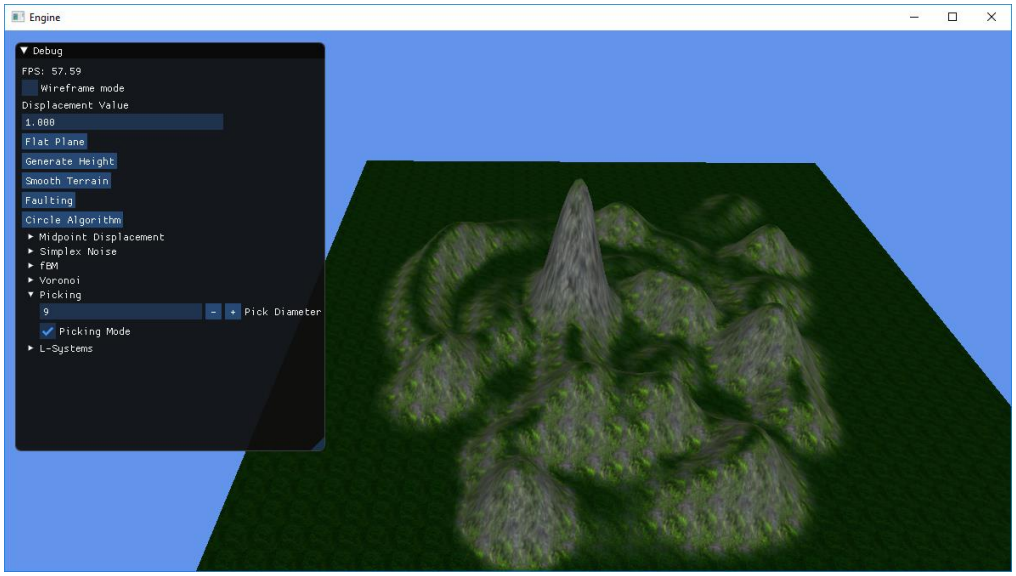
Voronoi Regions

Like the fault line algorithm, the Voronoi Regions function splits the terrain into sections (more details on *Efficient Triangulation Algorithm Suitable for Terrain Modelling*). This works by picking random points on the terrain, deciding (based on nearest distance to these random points) which "region" these points belong to, and then finally displacing the height based on what region it is in.



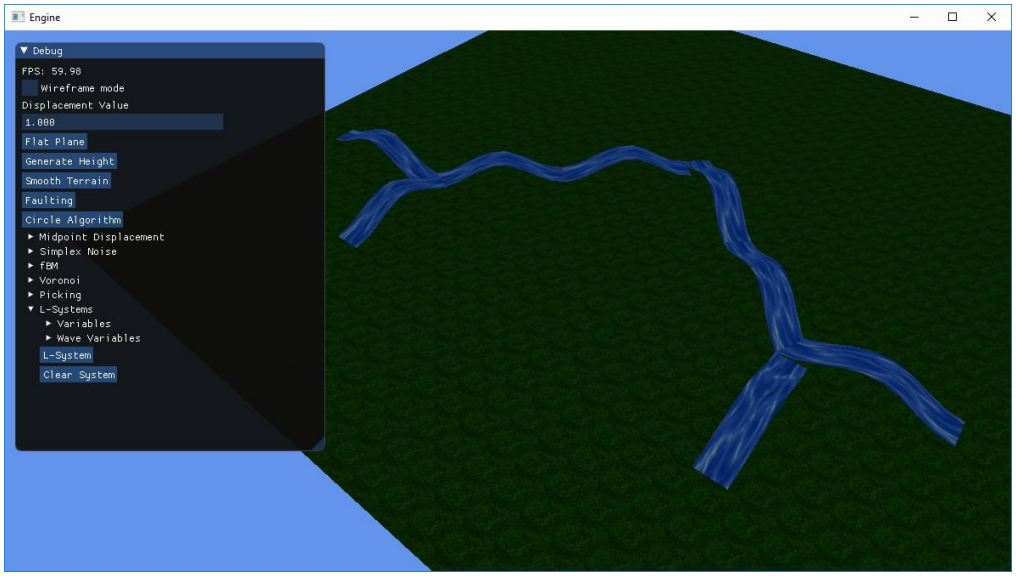
Terrain Picking

Essentially, ray-casting. Adapted from the tutorial *Picking*. When it is enabled, and the user clicks on the terrain, the screen co-ordinates of the mouse are converted into world co-ordinates via calculating a directional vector from the cameras position to the terrain. Where this directional vector intersects the terrain, a check is done to see what specific vertices it is nearest to. For the vertices near this intersection point, the refined circle algorithm is used to displace them if they are near this point and in the radius of the circle in the algorithm. Ensure the displacement value is low for this function to work properly.



L-Systems

Linden-Mayer systems or L-Systems are used to generate self-similar fractals (as detailed in *Lindenmayer Systems, Fractals, and Plants*). In this applications case, it generates a 2D river as it follows two specific rules. The L-System could be better integrated with the terrain, but this will be explored in more detail in the Critical Appraisal section. Based on a starting character, some input rules and an interpreter to determine what should be done when coming across certain character (E.g. F means move forward), a variety of shapes can be made.



Organisation

Terrain

By far the largest class in the application. This is due to it containing all the functions (with the exception of Simplex Noise) necessary to alter the terrain. It inherits from the BaseMesh class which allows it to be more easily initialised.

It contains integers for width and height, as well as the texture repeat amount which is used to set how many times the texture will repeat over the terrain.

It contains an XMFLOAT2 to help scale the terrain, as the current implementation does not feature tessellation meaning that higher dimensions must be used (and subsequently scaled down) in order to have displaced terrain looking relatively smooth.

The remaining contents are the VertexType vertices array which is used to store all the data of the terrain (position, normals, texture co-ordinates) and a secondary indices array which is used in the Pick() function to be able to access the index array.

A simplex noise object is included in this class so that the terrain can be displaced using simplex noise. This was done to reduce the number of Simplex Noise specific functions within the class.

A similar strategy was attempted with the Midpoint Displacement function although due to time constraints it never fully came to fruition.

The L-System class was also almost included in this class to allow the L-System to be better integrated with the terrain, however initial tests of merging these two systems proved too complex with the time remaining.

L-System

The L-System class includes the Manipulation Shader and Sizeable Quad classes. This is so that it can generate sizeable quads which will be used to render the L-System using the Manipulation Shader.

The rest of the class purely contains variables used in the generation of the L-System itself.

A starting character/string “axiom” which is used to start the L-System string generation off. A map which takes a character and string to contain the rules of the L-System. A stack of SavedTranform to save the position, rotation and world matrix at that point in the function. A vector of matrices which is used to render the quads of the L-System. A string which is used to show the current path of the L-System. And float 3’s to hold the initial position, current position and current rotation.

The float 3’s may not be used in the final iteration of the class as quads are generated using the saved world matrices and generating what is effectively unit quads at that point.

Sizeable Quad

This is a pretty simple class. Set up in a similar fashion to the Terrain class, it has a VertexType vertices array to hold position, normal and texture co-ordinate data, a float 2 used to scale the quad in the x and z directions and two floats for the width and height of the quad.

Initially it would take in the initial position and current position from the L-System class, however it became apparent that it would be easier to render *essentially* unit quads using the altered world matrices it received from the L-System class.

Other than what is mentioned above, it's rendered using the Manipulation Shader class to displace its y-position based on its z-position along its normal.

These values can be altered via ImGui to refine the look of the wave movement.

Simplex Noise

A class used to hold all the relevant data for the Simplex Noise algorithm. This was done to reduce the complexity of the Terrain class.

Its primary use it just to calculate what noise value should be applied to the terrain at a given position.

Terrain Shader

A relatively simple class that inherits from the BaseShader class (which contains all the base values and variables used to initialise all shader classes).

Specifically this class is used to set up a shader for rendering the terrain.

It takes in the world, view and projection matrices alongside textures for the terrain and a light used for light calculation on the terrain.

Manipulation Shader

A relatively simple class that inherits from the BaseShader class (which contains all the base values and variables used to initialise all shader classes).

Specifically this class is used to set up a shader for rendering the L-Systems quads.

It takes in the world, view and projection matrices alongside a texture for the quads, light used for light calculation on the on the quads and variables used to alter the wave motion of the quads.

Critical Appraisal

Terrain Class

The terrain class is the largest class in the application. This means it could benefit from having some of its terrain displacement functions put into their own class (such as the midpoint displacement functions).

As all of the functions run on the CPU, this does impact performance quite a bit when doing complex calculations (or just lots of simple ones in some cases). If this class was to be re-written most – if not all – terrain altering functions would take place within a domain shader so as to utilise the GPU to do the heavy lifting when it comes to calculations (which is what the GPU is good at).

The secondary indices array could be removed in favour of a better approach/access to the original indices array for use within the picking function.

The smoothing function, whilst it does work, does seem a bit extreme in certain cases, and in others, seems to cause weird lines (almost like divisions of the plane). Refining the function to allow the user to define how much the terrain should be smoothed by would be a better alternative as then it would allow the user to fine-tune just how much they want to smooth the terrain by.

The picking's circle algorithm could utilise simplex noise to further enhance the visual output from that function.

Regardless, the displacement functions do seem to work as intended.

L-System

The main flaw with the current implementation of the L-System class is that it's more like a secondary component of the application alongside the terrain rather than it being merged into something more cohesively.

If there was more time to refine the application this would mean that the L-System class would functionally be another feature of the Terrain class to allow it to access the required vertices of the terrain so that it could be better mapped to the terrain generated.

Tessellation

A rather major flaw of the application is that it does not utilise tessellation to sub-divide the terrain and l-system meshes to allow for higher resolution meshes.

This was done – on the terrain at least – due to the normal calculation function currently being run on the CPU. This would also have complicated access to the vertices data as it would have been a control point patch-list rather than the current triangle-list meaning that the normal re-calculation, vertices access and terrain functions would occur in the domain shader.

This would possibly complicate linking the L-System and the Terrain although without testing it's a rather moot point.

Overall

Whilst there are certainly areas I know that could be improved, the current implementation does function as intended.

If it was to be re-done a larger focus on utilisation of the GPU would be a primary goal to tessellate the meshes, recalculate normals and maybe utilise the geometry shader to produce a more fitting background.

The addition of a skybox could help to cement the user in feeling like they are in more of a mountainous terrain than a terrain tech-demo.

The texturing could also do with a bit of a refinement but again the current implementation of slope based texturing (as adapted from *Slope Based Texturing*) does do the job.

Changes

Just to re-iterate, the main changes to the application if it were to be done again would be:

- Improved texture tiling, current implementation does have certain issues concerning the "moss" texture.
- Add in a variety of lights to showcase the terrains ability to work with more complex lights, such as point/spot.
- Tessellation for both the terrain and the river l-system. This would reduce the performance hit of terrain manipulation functions as a relatively low resolution terrain quad could initially be defined at the start of the application, but due to tessellation it would retain a higher quality when being displaced. Tessellation of the terrain quad would also possibly move the terrain manipulation functions to the hull and domain shader stages, making the GPU do the calculations as opposed to the CPU.
- Better integration between the river l-system and the terrain. The current implementation makes the river l-system seem like more of an afterthought. Mapping the river l-system to any generated terrain would increase visual appeal.

Extensions to the current application could involve:

- Further experimentation with ridges in the noise functions (specifically the fractal Brownian motion function in this application) to demonstrate an understanding of both ridged and non-ridged terrain generation techniques.
- Generate trees as well as the river using the L-System class, this would increase visual appeal as well as demonstrating the ability to work in three dimensions instead of the current two the river l-system utilises.
- Implementation of terrain erosion similar to that as featured in *Fast Hydraulic Erosion Simulation and Visualization on GPU* and *WebGL Landscaping and Erosion*. This would greatly increase the visual appeal of the terrain/make it looks closer to that of which can be seen in real life.
- If *all that* could be completed within the time given of one semester, the next step (rather crucially wanted more than anything else) would be to add some sort of fluidity to the water as seen in *WebGL Landscaping and Erosion* (but that's probably a step too far).

Reflection

Building on top of the previous semester's introduction to programming with shaders, this module introduces procedural content generation as an important addition to that knowledge. From procedural modelling to advanced image synthesis, this module effectively allows you to take your base shader knowledge and put a procedural twist on it. Want to have awesome looking terrain? You got it. Prefer dungeons instead? No worries. Bit of an eco-centric person? Maybe L-Systems are more your style. This module allows you to go in whatever procedural direction you desire. This application in particular had a larger focus on terrain manipulation with a secondary focus on two dimensional l-systems, but that doesn't mean it couldn't have had l-system trees or small dungeons plotted about the landscape.

If there was any advice to give to someone who took on a similar task, it would be advisable to ensure that the terrain manipulation functions are carried out on the GPU via use of the hull and domain shaders to reduce the performance hit with certain functions.

Also, don't wait too long if you're thinking about introducing an l-system into your scene. They take a while to get your head around how they work within the DirectX framework, but once you have it working you can do pretty much anything with them.

The *L-Systems Renderer* was immensely helpful when it came to debugging the L-System as it output the final string it used to generate the system and what it looked like visually. This saved many hours from trying to see what it should look like when the L-System was broken.

More crucially, if you're going to implement perlin/simplex/other noise functions, make sure you understand them.

If you can, try extending what you had from the previous semester's introduction to programming with shaders. This would allow you to also have some image manipulation via shaders. But also, keeps you grounded in a framework that's familiar.

Not that the Rastertek framework is *bad* necessarily, it's just different to what became familiar in the previous semester. How certain things are handled differently can cause a bit of wasted time, and the fact you have ImGui from the get go due to using the same framework is one of the largest advantages you can have.

References

Elias Daler. 2019. Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies. [ONLINE] Available at: https://github.com/ocornut/imgui [Accessed 23/04/2019]

Nic. 2013. Simple terrain smoothing. [ONLINE] Available at:

http://nic-gamedev.blogspot.com/2013/02/simple-terrain-smoothing.html [Accessed 23/04/2019]

Antonio Ramires Fernandes. Unknown. The Fault Algorithm. [ONLINE] Available at:

http://www.lighthouse3d.com/opengl/terrain/index.php?fault [Accessed 23/04/2019]

Antonio Ramires Fernandes. Unknown. The Circles Algorithm. [ONLINE] Available at:

http://www.lighthouse3d.com/opengl/terrain/index.php?circles [Accessed 23/04/2019]

Steve Losh. 2016. Terrain Generation with Midpoint Displacement. [ONLINE] Available at:

http://stevelosh.com/blog/2016/02/midpoint-displacement/ [Accessed 23/04/2019]

Stefan Gustavson. 2005. Simplex noise demystified. [ONLINE] Available at:

http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf [Accessed 23/04/2019]

Patricio Gonzalez Vivo & Jen Lowe. 2015. Fractal Brownian Motion. [ONLINE] Available at:

https://thebookofshaders.com/13/ [Accessed 23/04/2019]

Glare Technologies. 2019. Advanced Fractional Brownian Motion Noise. [ONLINE] Available at:

https://www.indigorenderer.com/indigo-technical-reference/indigo-shader-language-reference/built-functions-%E2%80%93-procedural-noise-fun-0 [Accessed 23/04/2019]

Matej Zabsky. 2016. Procedural Terrain with Ridged Fractal Noise. [ONLINE] Available at: https://stackoverflow.com/questions/36796829/procedural-terrain-with-ridged-fractal-noise [Accessed 23/04/2019]

Red Blob Games. 2016. Making maps with noise functions. [ONLINE] Available at:

https://www.redblobgames.com/maps/terrain-from-noise/ [Accessed 23/04/2019]

Paul Bourke. 1989. Efficient Triangulation Algorithm Suitable for Terrain Modelling. [ONLINE] Available at: http://paulbourke.net/papers/triangulate/ [Accessed 23/04/2019]

Iedoc. 2015. Picking. [ONLINE] Available at:

https://www.braynzarsoft.net/viewtutorial/q16390-24-picking [Accessed 23/04/2019]

Przemyslaw Prusinkiewicz & James Hanan. 2016. Lindenmayer Systems, Fractals, and Plants. [ONLINE] Available at: http://algorithmicbotany.org/papers/lsfp.pdf [Accessed 23/04/2019]

Richard Hawkes. 2016. Unity Algorithms – The Lindenmayer Series (L Series). [ONLINE] Available at: https://www.youtube.com/watch?v=uBEA6VSUybk [Accessed 23/04/2019]

Sher Minn Chong. 2018. L-Systems Renderer. [ONLINE] Available at:

http://piratefsh.github.io/p5js-art/public/lsystems/ [Accessed 23/04/2019]

Rastertek. Unknown. Slope Based Texturing. [ONLINE] Available at: http://www.rastertek.com/tertut14.html [Accessed 23/04/2019]

Xing Mei, Philippe Decaudin & Bao-Gang Hu. 2007. Fast Hydraulic Erosion Simulation and Visualization on GPU. [ONLINE] Available at: http://www-evasion.imag.fr/Publications/2007/MDH07/FastErosion_PG07.pdf [Accessed 23/04/2019]

Florian Bosch. 2011. WebGL Landscaping and Erosion. [ONLINE] Available at: http://codeflow.org/webgl/craftscape/ [Accessed 23/04/2019]

Gareth Robinson – Helped to debug the midpoint displacement algorithm when it wasn't working correctly.

Gabriel Lacey – Helped by giving a basic understanding of L-Systems and introducing the wonderful L-Systems renderer for debugging l-system.

Paul Robertson – Compiled the framework which this submission uses instead of Rastertek.