**CMP305 Documentation**

**Michael Keenan 1801284**

How to Use

Controls: WASD – Move the camera forward, back, left or right respectively

QE - Move the camera down and up respectively

Right Click – Alters the view of the camera

UI: The debug menu has all the controls to access and alter the procedural methods.

Checkboxes:

Wireframe Mode: Activates Wireframe Mode for the meshes

Voxel Mode: When activated this change the generation mode to a voxel

style and when not activated the generation mode is based off

of a plane mesh

LSystem Mode: Changes the procedural generation to an LSystem based

mode. Only has pre-set generation currently.

Sliders:

Terrain Resolution: Alters the vertex count of the terrain mesh or increases

the number of generated voxels. Will not apply until another generation is selected. The voxels will render out of view if over roughly 200.

Buttons:

Regenerate Terrain: Generates a height map based on random values  
 Smooth Terrain: Performs smoothing on the terrain.

Fault Terrain: Performs terrain faulting on a randomised height map and

then performs 3 iterations of smoothing.

Particle Drop: Performs a particle deposition to raise the height map in a set radius

Generate Flat Plane: Generates a flat terrain mesh

Perlin Noise: Performs 2D Perlin Noise on a terrain mesh to generate a landscape

Voxel Perlin: Performs 2D Perlin noise in a voxel style

LSystem 1 & 2: Generates a line based LSystem based on already set rules

New Seed: Reseeds the random function for Perlin and Regenerate

Overview

The application was designed to implement and generate a range of procedural generation techniques and to show how factors of these techniques can influence the outcome. The techniques are mainly focused on terrain generation using a height map and plane mesh or a voxel style. These terrains have both been textured using a height-based texturing system and the plane mesh style implementation also has had texture blending implemented alongside steepness-based texture generation. Additionally, a LSystem has been implemented to demonstrate an additional generation technique.

Effects of Sliders on Generations

**Faulting: Offset** is the change in height per iteration. **Scaling** is the amount that the faulting offset decreases by per iteration. **Iterations** is the number of times the program will fault. **Smoothing Iterations** is the number of times smoothing is applied

**Particle Drop Amount** is the change in value of the height around the drop.

**LSystem Iterations**  is the number of times the LSystem will run. **LSystem Iterations**  **360** is the angle that the System rotates by and repeats.

For the voxel and terrain Perlin, **Octaves** is the number of additional times the function will run in a Fractional Brownian Motion style. **Amplitude and Frequency**  are self explanatory.

Perlin Noise Explanation

Noise is in essence random values and is generally attributed to unwanted background sounds. The most basic of this is white noise which is completely randomly generated values.

Perlin Noise was created by Ken Perlin in 1997 as an improvement to noise and aims to create a smoother and more gradual version of a noise function. It is a pseudo random function that uses a permutation table and linear interpolation to generate a value for an array of points.

The function takes an input coordinate in any number of dimensions and generates a unit square/cube depending on the number of dimensions. For each of the vertexes in the unit cube a pseudorandom vector of the gradient is generated based on the permutation value that matches with the input coordinates and fractional part of the input position. Due to this Perlin Noise can only generate for float values as providing integer data will always output 0. Next using the input point, a distance vector is generated to each of the vertexes and the dot product of the distance vector and the gradient vector is taken to give the amount of influence each point has on the final output value. These influence values are then linearly interpolated with each other to acquire the result of the 4 influence values on the point. The function outputs value between -1.0 and 1.0 and such is usually required to be scaled up for procedural techniques This gives us values for each point given to the noise function but the values whilst generated by the same algorithm aren’t link and still don’t have the gradualness that is require for procedural generation and so the fade function is used. The fade function creates a more gradual blend between the values, especially when you start to approach values closer to integer values.

It is defined by Ken Perlin as 6t5-15t4-10t3 and this is considered by him to be the most effective fade function.

LSystem Explanation

An LSystem, also known as a Lindenmayer system, is a parallel rewriting system used typically to generate complex shapes algorithmically through iteration to simulate natural growth of the geometry. The results of each iteration is used as the base foe the next iteration. The recursive nature of its generation leads to similarity between iterations and is used mostly to create fractal and plant like forms but also has applications in the generation of artificial life. Systems that generated forms of geometry tend to manipulate an array of characters to generate a system.  
An LSystem consists of 3 main parts:

The initial string given, known as the Axiom. This defines the start position of the generation.

The variables and constants, these are alphanumeric characters and symbols that make up the axiom and its further iterations. Each unique character within the system has an effect upon the final output generated such as increasing the length of a drawn line or rotating it along an axis.

The rules, this is the main section of the LSystem, these define how the LSystem evolves. The LSystem iterates through the current system/axiom that currently exists and modifies it based on these rules that it has been given and returns a new output system that can be iterated upon again or used to generate geometry.

Minor Technique Explanation

**Smoothing:** When terrain is generated, depending on the algorithm used, it can appear jagged or rough especially when working with high resolution terrain that has per vertex generation. Smoothing is a technique that creates a more gradual change in height or steepness values by using the values of the surrounding points. The algorithm generates the average height of the surrounding point and then averages that value with the value of the initial point. This is then repeated upon every point to create a smoother terrain. Due to the averaging of heights, repeated smoothing will have a lower and lower effect and eventually trend towards zero without alterations.

**Faulting:** Faulting is a simple terrain generation technique that is not algorithmically or computationally heavy. It consists of defining a line across a surface, then shifting all the point on one side of the line up and down on the other side of the line. This is then iterated a number of times to generate a generally realistic looking terrain based on its complexity. Increasing the number of iterations increases the detail of the terrain but tends to generate very jagged terrain and is usually combined with other techniques such as smoothing to generate a more realistic terrain.

Code Architecture

The application was designed to be well organised and manageable to demonstrate a high-quality code architecture. Three main factors were considered when implementing the application, the application should be easily readable, easily portable and reusable and easy to debug if necessary. These were the main focuses, but it was also important that the robustness and maintainability and scalability of the application was not lost in attempting to focus on these.

To create a robust and scalable code it was critical that changes to the code could be implemented without affecting other section of the application and new classes and procedural algorithms should also be able to be easily and safely added. If the application were to be extended upon or modified or repurposed the core functionality of it should remain. To achieve this goal the application aims to make use of the Object-Oriented programming advantages of the language and as such all of the major functions were created within their own class definition and designed in a way that they could be easily implemented into another application if necessary. A focus on containing all the functionality of the technique within the class was given a high priority to achieve this so they have minimum reliability on external libraries that are not included within the standard library. The exemption to this is the DXFramework itself. Each class should be able to be taken from this application and implemented to another application without having to majorly change anything or rely on anything else apart from its other key components such as the LSystem class and the rule class. The use of clear and frequent commentary will allow for easy portability to not only other C++ application but allows user of other languages to understand the structure and function of them to help with implementation.

The organisation of the code is clear and examinable and alongside the comments aimed to create an easily readable code for any other programmer looking at it. The variables and functions were named to be clear and informative whilst remaining functional. Whilst not the most important feature of code architecture, readability was an important factor to keep in consideration. The use of a selection of minor techniques such as using the Camel Case naming scheme and consistent indentation of loops and function accumulated into helping to create a much more consistent and readable application

The focuses on portability and readability were key factors in creating a code that was also easy to debug, many of the problem encountered during development were easily identifiable and solvable with little issue. The focus on Object-Oriented programming allows many of the problems to be self-contained within their respective classes creating an environment with easy to locate and understand bugs which in turn lead to simple debugging processes.

Critical Appraisal

The application created provides a very satisfactory solution to the brief given, it demonstrates a accurate implementation of the procedural techniques that have been implemented and allows for effective visualisation and interaction with said techniques with the ability to manipulate and explore the landscape. The code is implemented in an organised and concise way and the implemented procedural techniques are efficient and effective.

Overall, the application achieves what it was set out do achieve, the techniques have been implemented in a mostly efficient, organised and satisfactory way. The implementation of Perlin Noise with the plane mesh and the voxels especially achieve what they were implemented to achieve. However, there are still some areas where the application did not achieve exactly what it achieved and, with more time, improvements and alterations would be implemented. A system for rendering the LSystem using 3D meshes and implementing Poisson sampling for an algorithmically generated landscape were the main shortcomings.

The most major ways the systems could be improved in in the LSystem and in the voxel generation. Currently the LSystem generates many separate lines for its generation of shapes however it would be much more efficient to render it as one continuous but manipulated line or to instance the line mesh. The first approach may not be as effective however if the 3D style was implemented for shape generation, but the instancing would still be more efficient. Similarly the voxel generation, is quite efficient as it uses instancing to generate each cube but as the cubes are rounded to integer positions instead of floats, this created an over lap in rendering and wastes processing time on rendering cubes that are already in the world.

Reflection

Developing the application has given a new insight into procedural generation. It has helped to develop a whole new way of creating convincing worlds and scenarios for users and allows those who do not have the knowledge of skills such as 3D modelling or art to create realistic worlds. Alongside that it gives a new way to look at creating objects through otherwise not thought about means such as generating plant life through the use of systems like LSystems. From a technical standpoint is has also developed skills in understanding and implementing complex techniques and how mixing many techniques will almost always give a greater result than merely using one.

Procedural generation is not limited to merely creating landscapes like it is demonstrated in this application, the knowledge learned is applicable to many areas such as sound, animation and in areas that would not initially come to mind when considering procedural generation such as story and evolving game systems.

Learning about procedural techniques opens many more potential future work and career paths that were not considered options before development as it encompasses such a large field with many different avenues that can be taken.

References

**Perlin Noise**

Wikipedia. (2020). *Perlin noise*. [online] Available at: <https://en.wikipedia.org/wiki/Perlin_noise>. [Accessed 6 Jan. 2021].

adrianb.io. (n.d.). *Understanding Perlin Noise*. [online] Available at: <https://adrianb.io/2014/08/09/perlinnoise.html>. [Accessed 6 Jan. 2021].

Khan Academy. (n.d.). *Perlin noise (article) | Noise*. [online] Available at: <https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise> [Accessed 6 Jan. 2021].  
rtouti.github.io. (n.d.). *Perlin Noise: A Procedural Generation Algorithm*. [online] Available at: <https://rtouti.github.io/graphics/perlin-noise-algorithm> [Accessed 6 Jan. 2021].

Scratchapixel (n.d.). *Perlin Noise: Part 2*. [online] Scratchapixel. Available at: <https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2> [Accessed 6 Jan. 2021].‌

gpfault.net. (n.d.). *Using Perlin Noise to Generate 2D Terrain and Water*. [online] Available at: <https://gpfault.net/posts/perlin-noise.txt.html> [Accessed 6 Jan. 2021].

**LSystem**

www.sidefx.com. (n.d.). *L-System*. [online] Available at: <https://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html> [Accessed 6 Jan. 2021].

‌Uni-hamburg.de. (2019). *An Introduction to Lindenmayer Systems*. [online] Available at: <http://www1.biologie.uni-hamburg.de/b-online/e28_3/lsys.html>. [Accessed 6 Jan. 2021].

‌ Wikipedia. (2021). *L-system*. [online] Available at: <https://en.wikipedia.org/wiki/L-system>

[Accessed 6 Jan. 2021].

‌paulbourke.net. (n.d.). *L-System manual*. [online] Available at: <http://paulbourke.net/fractals/lsys/>.

[Accessed 6 Jan. 2021].

‌Harzallah, H. (2020). *L-systems : draw your first Fractals*. [online] Medium. Available at: <https://medium.com/@hhtun21/l-systems-draw-your-first-fractals-139ed0bfcac2>

[Accessed 6 Jan. 2021].

‌MORPHOCODE. (2010). *Intro to L-systems*. [online] Available at: <https://morphocode.com/intro-to-l-systems/> [Accessed 6 Jan. 2021].

‌**Minor Algorithms**

www.lighthouse3d.com. (n.d.). *OpenGL @ Lighthouse 3D - Terrain Tutorial*. [online] Available at: <http://www.lighthouse3d.com/opengl/terrain/index.php?fault> [Accessed 6 Jan. 2021].

Nic (2013). *Game Programming and Development: Simple terrain smoothing*. [online] Game Programming and Development. Available at: <http://nic-gamedev.blogspot.com/2013/02/simple-terrain-smoothing.html> [Accessed 6 Jan. 2021].