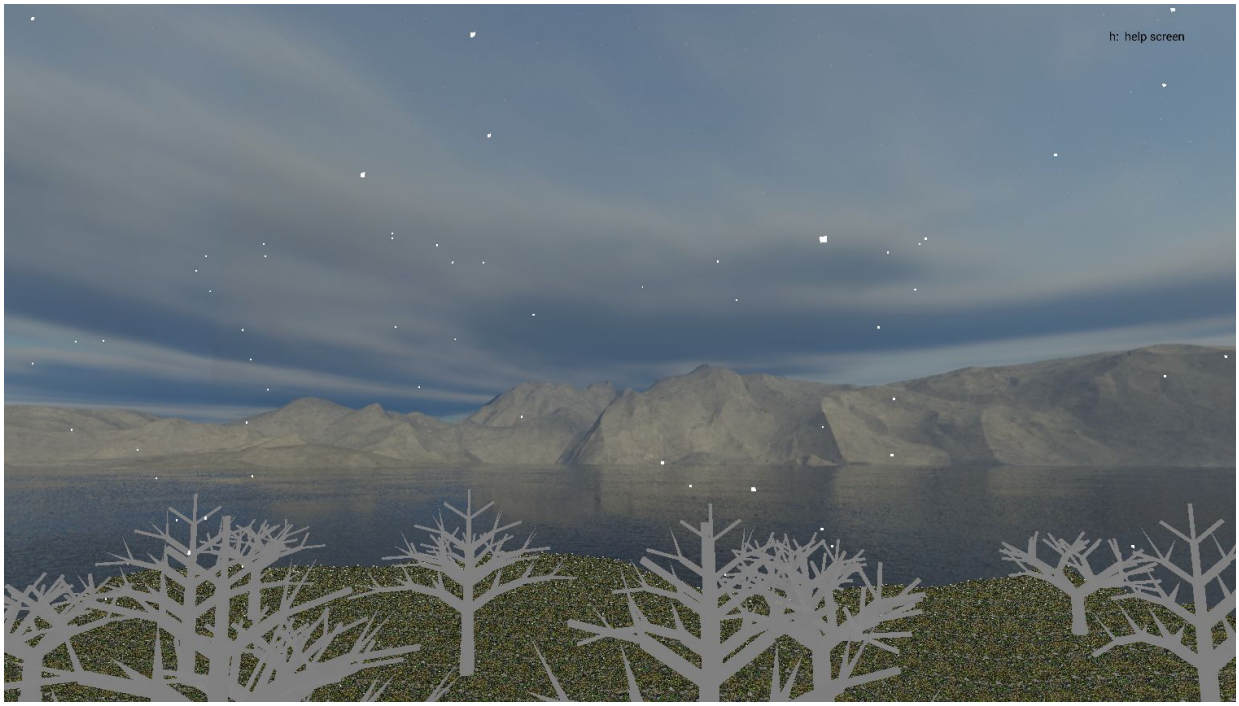


World-101110011

Final report

COMP 371 - Team # 13

13.04.2017

Name(s): Vsevolod (Seva) Ivanov,
Tarik Abou-Saddik, Sami Boubaker,
Justin Velicogna, Eric Morgan

Index

Overview

The purpose of this application is to explore computer graphics using modern OpenGL with the C++ programming language. More precisely, we are interested in the creation process of a procedurally generated world. We decided to go with the idea of recreating the nature in a form of a forest. For the world, we attempted to underline the artificial nature of the computer generated world. Depending on the time constraints, this forest will have various “objects” on its terrain like trees, plants, bushes etc.

This world will present an artistic simulation of a forest to be explored by its user. It will be presented as an extendable base structure. World-101110011 will serve as an inspirational environment relying on a visual representation of an artificial world.

Project URL: https://github.com/sevaivanov/world-101110011

Goals

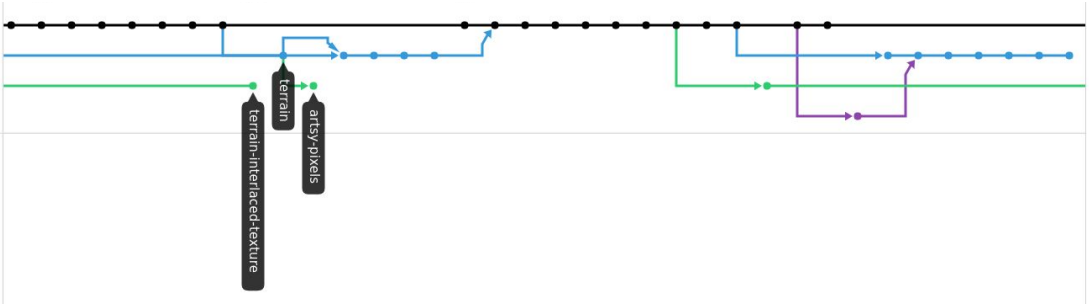
1. Create a procedural world with a terrain and a floating camera
2. Load procedurally generated objects into that world
3. Handle collisions
4. Integrate light, shadows and weather

World // Architecture

The world is procedural due to its unique representation after each creation.

At first, a basic Shader, Camera, Window and World classes were defined. From this point, a Makefile with targets was written to compile everything for various Unix platforms. To simplify the merging, we decided to use an abstract Mesh.cpp class to uniformize the rendering across different elements of our world. For each unique element of our world such as Terrain, Forest, Skybox and Weather, a simple render function is called from the World::draw method delegating the segmented implementation details to the assignee.

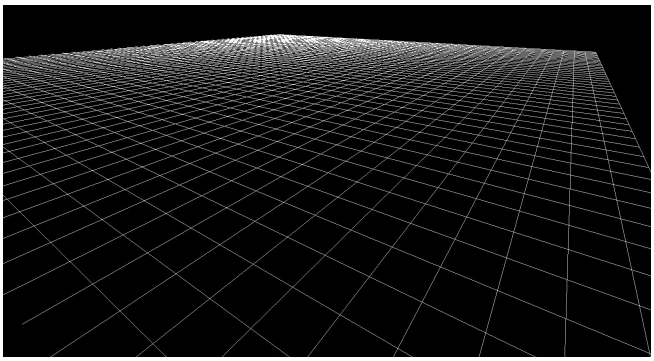
We decided to use Github revision system to help us to collaborate smoothly, avoid regression and snapshot various patches / features by using different branches :



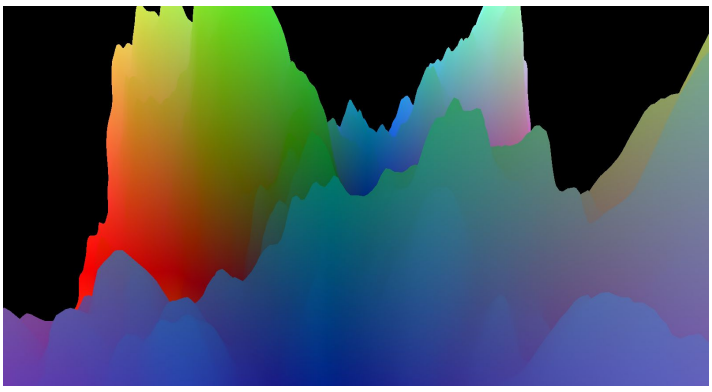
Terrain

The Terrain is procedural due to its usage of Height Map with customly written noise functions. Moreover, it uses recursion to generate its grid surface by wrapping the Height Map generator.

At first, I created a simple grid to explore a very basic layout of the cells using the dimensions of width and height of the world size input. This helped me to grasp the trivial flat terrain structure stretching along the x and z axes.



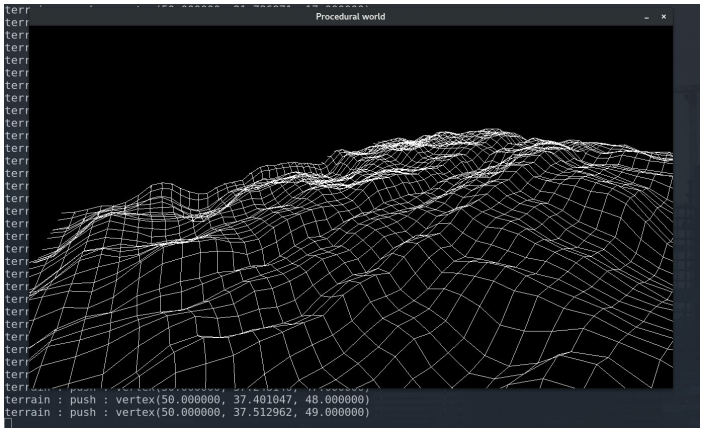
From the point, I started reading about the Height Maps and their implementation. A recurring element was the usage of the noise functions such as Perlin, Simplex and Cellular. It seemed that in order to create a realistic terrain one would need to blend many of those functions together to create a realistic irregularity of the real-world terrain. The latter brought me exploring various noise libraries.



In particular, the libnoise library seemed very promising. It was very simple to integrate and it allowed me to create highly realistic mountains. However, we wanted something personalized as a more flat terrain to allow a forest with trees and this is where everything got very complicated.

In fact, this library is perfect to wrap various noise functions and their blending but the extension of those requires their utils which are written in C and they are quite hard to understand and even more to personalize. I think it is a great framework which can even allow you to create Terragen alike planets but for our project and the learning purposes it seemed more reasonable to implement basic noise functions myself.

I went with a creation of HeightMap located in the TerrainHeight class which is based on research saved in *docs/research/PerlinNoise.pdf*



I needed to code a Perlin noise mathematical functions which should be in 2D since we are feeding an x and z values to have as a return an y elevation value. After quite a learning curve to understand frequencies, octaves, etc., the TerrainHeight class produced an interesting looking terrain. Of course, I could have copied this code directly into the terrain vertex shader but considering the impossibility of printing values and its early stage of development, I decided to leave this step for the stage when everything works properly.

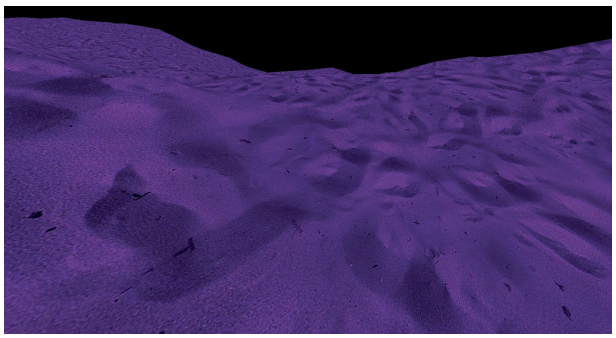
One thing that I didn't completely understand is to how properly "clamp" the noise output y (elevation) values to stay between the reasonable "min < y < max" range without breaking my Perlin noise functions.

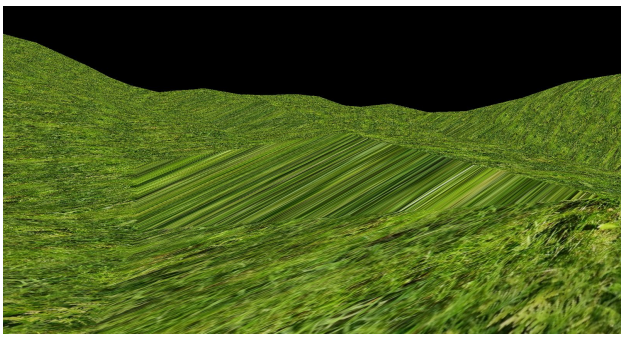
The way I understand it : to generate an infinite terrain in every directions, we have to create an illusion of movement by positioning the camera at the center of the initial terrain and then, advancing it in the proper direction by regenerating one row of a terrain on each step forward using an offset along x and z axes. However, the previous "clamping" issues to avoid excessive y heights got me stalled. Nevertheless, I programmed the architecture and logic of the advancing the terrain instead of moving the Camera forward in the world :

https://www.instagram.com/p/BS4JZwngLQO/

From this point due to lack of time, I decided to prioritise the compilation of the other world parts and the overall aesthetics of the terrain. Therefore, I left this Work in Progress under the "terrain-infinite" branch for future development.

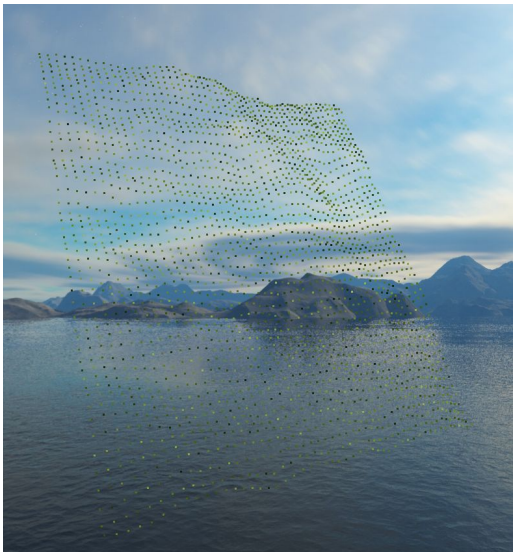
To enhance the aesthetics, I implemented textures and then I tried various looks:

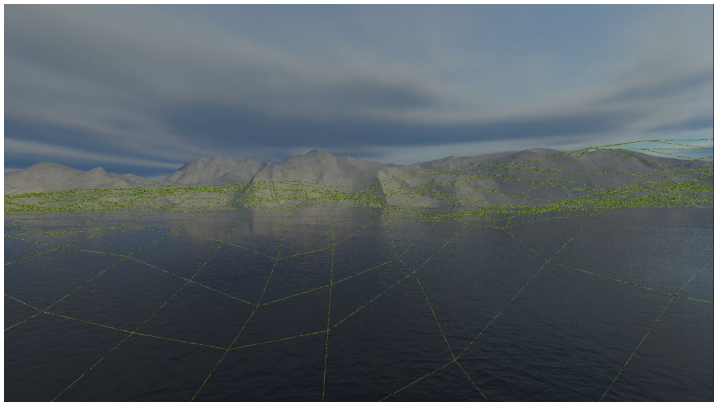




It led me to discover a little anomaly on the RHS that I will solve in the foreseeable future.

The support for multiple rendering modes such as points and lines was added using the Abstract Mesh implementation. This feature is very helpful for debugging due to the lack of proper lightning. For these vertices indices generation, an EBO buffer was used to refer to the real vertices found in the VAO buffer in order to avoid VBO vertices duplications :

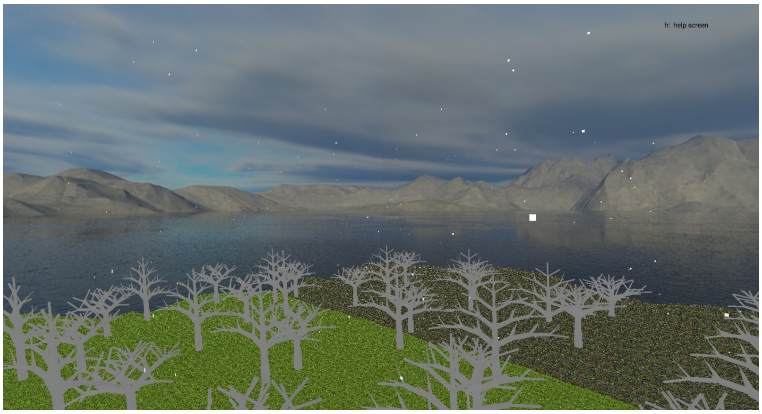




Note the fascinating Window thematic looks on the left hand side.

The render mode of points was used in combination with the snow weather as an overlay of white points to recreate an impression of having snow on the ground.

Finally, I implemented the multiple texturing bound to the elevation of the terrai. This feature would create rocky mountains with variant ground types. However, as my previous bug (involving an elevation growing excessively on the y-axis without any proper “clamping’) haunted me, it only changed the texture on one of the side of the terrain. Nevertheless, the implementation is there and it is working :



Ambiance

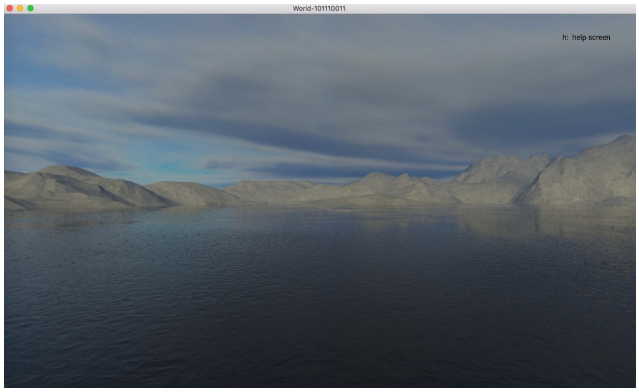
Sound

To stimulate the user's senses and re-create an illusion of the actual forest environment, a background soundscape was added using the SFML sound engine. Other engines such irrKlang were tested but SFML had an active cross platform support which is more promising in terms of out-of-the-box compatibility and long term support.

Skybox

In order to give the illusion of a vast open environment, implementing a skybox by way of a cubemap texture seemed to be the most feasible approach. In the simplest of terms, a skybox is essentially a 3D cube which has a six-sided texture, known as a cubemap, overlaid on it faces.

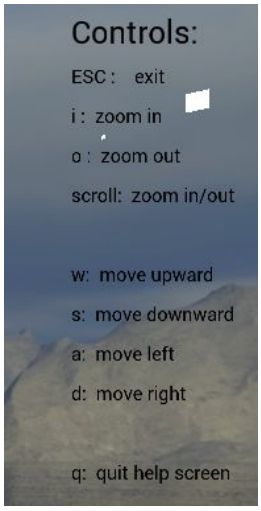
The skybox was defined as a class and simply called on as an object in the World.cpp file, where the main rendering of our scene takes place. Although the original idea was to render out two skyboxes and have one blend into the other (i.e. create a day/night cycle), there was little time in the end to actually implement the idea.



UI (On-screen help)

Although we had never really discussed the need for a sophisticated user-interface in our application, we decided in the end that it would be best to at least render out a basic on-screen 'help' panel.

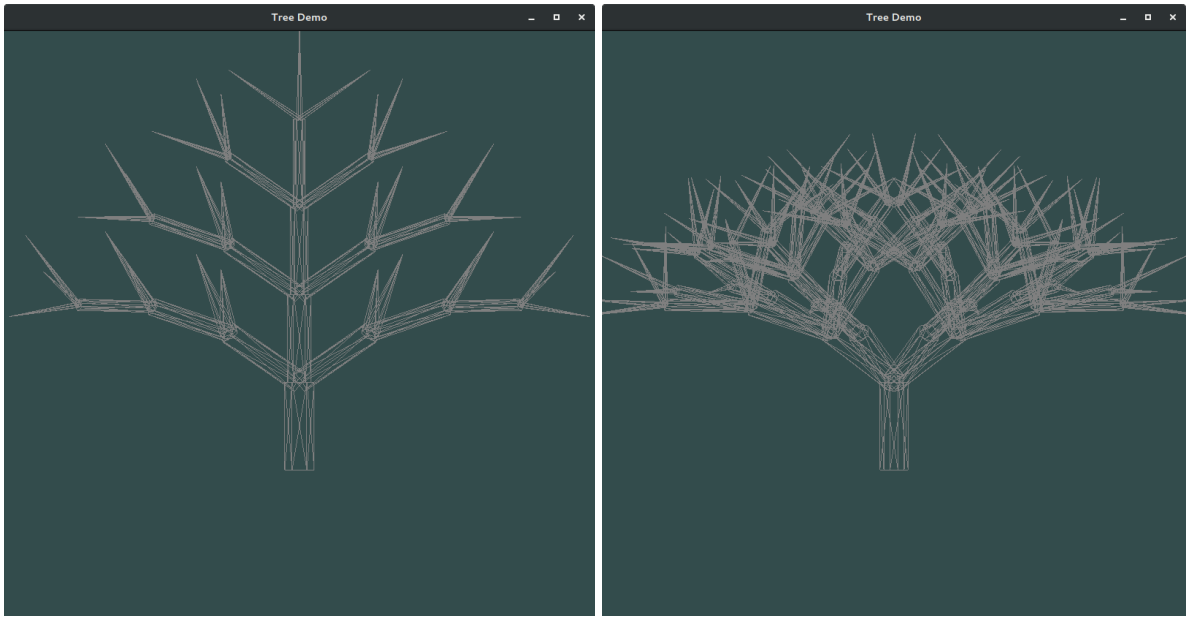
This was done using FreeType, a library able to load a multitude of fonts, such as the Roboto TrueType font that we utilized for our application.



Forest

The Forest is generated by instancing several procedurally generated trees. The trees uses an L-system to recursively generate, rotate, and translate branches to attach them together. The branches will decrease in radius at each iteration until it reaches a radius of 0 where the recursion will stop.

The programer can manipulate various aspect of the tree such as height, radius, number of iterations, and can modify the entire tree by using a different 1-system vocabulary.



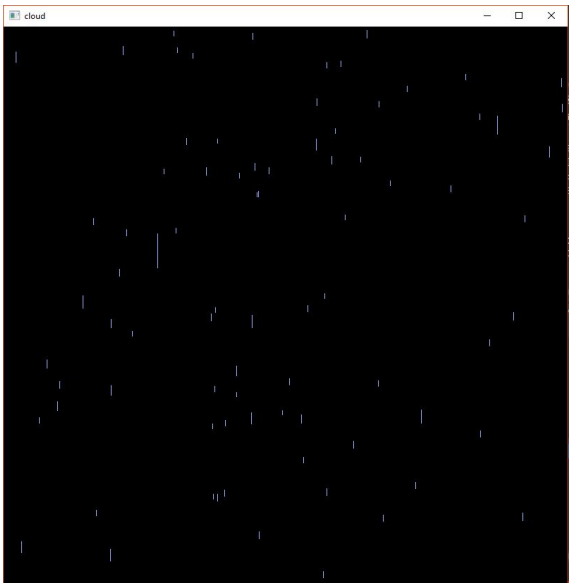
The procedurally generated objects support instancing to optimize rendering. All properties of the mesh (vertices, indices, etc.) are stored only once and the program will use a translation array to render this mesh at multiple locations.

Shadows

Shadows are added to tree objects. These shadows are created using a subclass of the GLObject class which creates a clone of a given object, flattens it, and changes it's color to black (Shadows via ray tracing were not used due to the large computational time required to calculate intersections). Ordinarily, the class would take a light source and a slope, and project the light source onto the slope, however, due to an error in the ray-line intersection function, the shadow is currently projected straight downwards onto a flat slope (however the architecture is still there, provided the ray-line intersection function is fixed).

Weather

The weather effects are rain and snow, both represented using a simple particle system. Both are created by a Cloud object which contains Particle objects, and each frame, the effects of gravity and wind are applied to each Particle (Wind is simply a vector which is added to a particle's velocity). A rain particle is simply a vertical blue line drawn between two vertices, and a snow particle is a 2D white quad.



For the life condition of each particle, I chose position rather than time, as all falling particles will hit the ground eventually. As a Particle falls, it is reset to somewhere in the Cloud whenever it crosses a set threshold denoting the ground. Rain and Snow differ in the form of each particle and how gravity affects them. A raindrop falls hard, while a snowflake drifts slowly down, and from side to side.

Documentation

The three documentation resources are:

- Doxygen code documentation: helps to understand the implementation
- README : containing the Roadmap of what is done and is not finished
- **docs/** folder: containing the resources such as:
    - Help markdown file
    - Process screenshots
    - Research & Reports

Authors

Student ID	Name	Parts
40004286	Vsevolod (Seva) Ivanov	Project architecture Compiling on Unix Floating camera Procedural terrain Ambiance sound Merging parts Documentation
40005294	Justin Velicogna	Trees shadows
26417404	Sami Boubaker	Procedural Trees (L-System) Procedural Forest with Trees Forest on terrain integration Documentation
27518722	Tarik Abou-Saddik	Skybox Camera mouse controls Textual on-screen help Documentation
26863426	Eric Morgan	Cloud, Rain, Snow, Wind Documentation

Notes:

- We took Justin on board following the March 30th lecture
- Details on the implemented parts can be found in the README > Roadmap