Computer Graphics 371
Project Report

Fall 2016

Michael Sterle-Contala (25469260)

Kendy Jeune (27014465)

Introduction

Our project consists of a world created procedurally that can be explored by the user. We chose this topic to learn how to produce models and textures algorithmically and also to take on the rendering challenges of an uncertain and dynamic scene. We chose a nature setting for our world because organic objects express themselves in a greater diversity while still being constructed from systems and rules that can be expressed with algorithms.

The project was divided into 4 parts: asset management, rendering implementation, asset generation and user interface. The challenges faced and solutions chosen are detailed below.

Asset Management

Models

Due to the nature of procedural model generation, there would be a great many copies of simple primitive meshes, and so efficient instancing would be key. To this end, 3 classes were defined, the Model, Mesh and MeshInstance classes. MeshInstance objects hold the instance-specific data, namely a transformation matrix and material properties. Mesh objects hold the vertex data, texture data, and MeshInstance objects as well as a VAO and various OpenGL buffer objects. Finally, the Model objects hold a tranformation matrix for model-scope transformations as well as a collection of Mesh objects. In this way, a call to Model::draw() calls each individual Mesh::draw() in its collection of Mesh objects. This Mesh::draw() in turn passes all necessary vertex and instance data to OpenGL.

To optimize the draw call, the Mesh's collection of instance data had to be both 1) contiguous in memory and 2) randomly accessible. This was a challenge, because no std container provides for both of these; vectors are contiguous but change location in memory, and list members remain in the same memory location but are not conitguous. Our solution to this was to hold our MeshInstance objects in a vector and implement a MeshInstancePtr class which behaves like a pointer, but actually keeps track of the vector container address and offset. In this way, our two conditions were satisfied.

Textures

We needed cubemap, 2D textures and possibly 1D textures, so we created a general abstract Texture class to handle the common functionality of creation and modification. Because textures were going to be primarily generated (as opposed to loaded from file), simple texture data access was a defined in the Texture base class and implemented in detail at the subclass level. Texture objects are linked to OpenGL textures through their unique OpenGL reference.

Framebuffer Objects

Because all of our textures were going to be generated from simple functions, we knew that we would require post-generation effects. To this end, we created Filter and Blender classes to allow convolution filtering and texture blending respectively. These were accomplished through the use of framebuffer objects, which, similar to Texture objects, are linked to OpenGL assets through native reference ID.

Shaders

We would optimize our rendering process by having many specialized shaders, and so the Shader class was defined to make shader management simpler. Again, like the Texture class, the Shader class links objects to OpenGL shader assets through native ID.

Rendering Implementation

Shadows

Several approaches were looked at for shadow rendering, and although light mapping would have allowed faster rendering, we wanted to create a framework that could accommodate dynamic objects and animation in the future. To this end, we decided to implement shadow mapping for all shadows. We originally created our shadows by running the shadow map through a PCF subroutine in the fragment shader, but after implementing and testing VSM, we saw smoother shadows and increased framerate, as seen in Illustration 3 below.

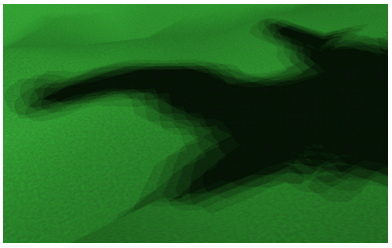

Illustration 1:
Shadows using PCF
(3x3 Kernel)

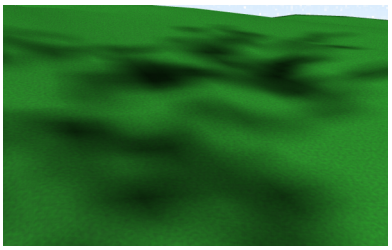

Illustration 2:
Shadows using VSM
(3x3 Kernel)

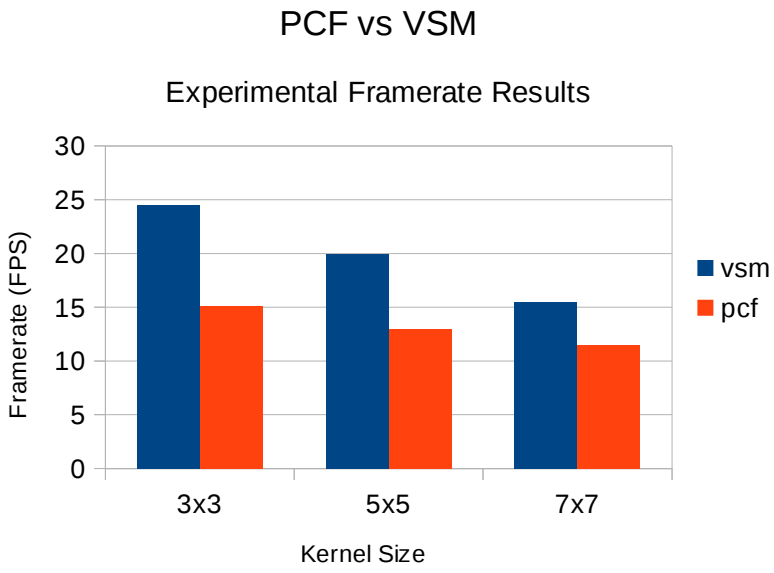

Illustration 3:
Comparison of framerates for shadow mapping methods

Transparency

Although translucency was beyond the scope of our project, basic cutout transparency would be required for tree leaves. To achieve this, we were able to avoid OpenGL blending and simply set an alpha threshold test in the fragment shader.

Procedural Asset Generation

Our goal was to recreate a natural setting, and to achieve this, we needed to create algorithms to generate organic-looking objects. Through research, we found that the methods at the heart of procedural asset generation are fractals and noise functions, both of which we used heavily.

Models

We settled on trees and shrubs for our model assets, as they would allow us to use the greatest range of our rendering features. We chose to generate these models through fractal generation. Fractals are self-repeating patterns of a recursive nature, and are ideally suited to simulating organic objects because they replicate the cellular growth process found in nature. To this end, Lindenmayer systems (L-systems) were chosen for their simple elegance, common use in botanical simulation and ability to express a large variety of outputs.

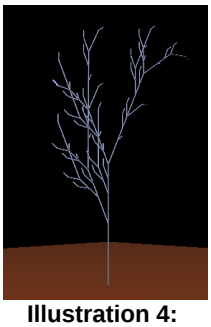

Illustration 4: Basic L-system (96,768 polygons)

We first built a basic L-system compiler that operated on strings. This L-system class used a set of production rules to operate on a string of symbols (word). Starting with an initial word (the axiom) and following the rules for each symbol, it generated a new value. This value could then be interpreted through the rules to create the next generation value, and so on iteratively. To “draw” these words we built a 3D turtle graphics interpreter. We adapted standard Turtle graphic commands to our model-based architecture – for example, instead of drawing a line, going forward causes the turtle to move forward and place a cylinder between its previous and current position.

To allow a greater variety of output expressions we replaced this with a parametric L-system compiler. This allowed each symbol to have associated parameters which affected both production rules and turtle actions. For example, while the non-parametric L-system would require X “forward” actions to produce a branch of length X, the same output could be achieved with a parametric action “forward(X)”. This resulted in far less polygons, vastly speeding up rendering. In a similar way, we were no longer constrained to fixed width or rotations of a fixed amount, allowing for the greater diversity.

Illustration 5: Parametric L-system (32,760 polygons)

Table 1: 8 variations using the same parametric L-system with different parameters

Textures

We implemented noise functions as the base of our textures. There were many types of noise functions to choose from, but the one we went with is Perlin noise because it is easily tileable and can be made fractal.

We implemented a basic 3D Perlin noise function and used it as a generator function for most of our textures. To add different effects to the noise we also developed several Filter and Blender classes for linear convolution filters and texture blending respectively. These work by processing the txture through a fragment shader using a framebuffer object. For example, the VSM shadow map texture has a 3x3 gaussian blur applied and the bark texture is octave Perlin noise first blended with a Sobel filtered copy of itself and then colorized.

For the terrain we produced 3 textures, one each for grass, snow and rock. These are tileable and are all passed to the shader; the once that is used is decided on a per-fragment level in the fragment shader based on the fragment's slope and altitude.

The skybox is textured using 3D perlin noise. The 2D texture coordinates for each face of the cubemap are converted to 3D rays starting from the camera and passing through the given point on the cube. These rays are extended to a virtual sky plane parallel with the ground. The final coordinates on the sky plane are the ones passed to the perlin noise function, allowing the skybox to give the impression of an endless sky in any direction. In addition, we add an exponential fade-to-white above the horizon to simulate Raleigh scattering.

Note: *Due to time constraints, the tree leaf texture is the only texture that is not generated, but instead taken from a public domain image.*

User Interface

Key Interaction Input

The main objective for the keys interactions input was to let the user have access to different sets of keys for the navigation modes implemented in our project. We desired the controls to be fairly simple and straight forward. We sought the navigation to be similar to first person controls with the mouse moving the camera look-at direction and the keys moving the position of the camera. Implementing the key interactions is a very interesting part of design since it allows you to look around and have a first sensation of control. Every resources needed for the achievement of the keys callback function was provided by the GLFW library therefore we did not need to add any specific implementation.

The keys are as follow:

Walk Mode		Explore Mode
W, A, S, D	Move forward, backward, left, right	W, A, S, D
Unavailable	Rotate terrain around the X-Y-Z axis	X, Y, Z

Unavailable	GLPolygonMode. GL_FILL, GL_LINE, GL_Point, GL_TRIANGLE (respectively)	T, V, P, U
H	Toggle Between Walk Mode, Explore Mode	H
Unavailable	Fly upward /Move camera to +y axis	Spacebar
Shift(Hold)	Run/Increase camera speed	Unavailable

Camera Movement

Initially we had in mind to implement diverse way of moving the camera. After discussion we finally agreed upon implementing two different ways of moving the camera resulting in different traversal modes. This project consists of two different navigation approach. Explore mode allow the user to traverse the terrain without any height or ground collision restriction letting us observe the field in infinitely different views. Additionally, this mode was intended to provide complete control over the terrain in order for us to fix bugs. To accomplish this mean of navigation we made sure that the mouse movement moved the camera as well. This first step was greatly needed in order for us to know in which direction we are going (toward the terrain or outside the terrain). Walk mode allow the user to move along the terrain height to represent an object/player on the terrain. For this mode, the objective was to lock the camera at a certain height hence making the user not able to move along the y-axis or penetrate the terrain. We wanted this approach to be both slow and fast, thus we added a running feature allowing us to increase the camera movement speed by a factor of 0.7. To use that feature the user simply has to hold the left shift key while moving.

Collision Detection:

By far, one of the biggest challenge in this project. With little experience in collision detection implementation, this topic was very hard to grasp mostly because of a general topic it is. Each objects interact differently when colliding with another object. For this design we, again, tried to keep it highly simple. The technique applied in this project can be described in the following algorithm:

1. Finding the position of the particular object,
2. Creating a bounding volume around it’s main by calculating the top, center and base position
3. Using glm build in function to calculate the distance between the camera and the bounding volume.
4. When a certain distance has been reached we restrict the camera from moving forward or in a direction that will place it over the maximum distance.
5. Reset the camera position away from the bounding volume each time it collides with the object to give the impression of a ‘’bump’’ collision.

Fortunately, a huge variety of resources were available online and we learned that there exist more than a dozen techniques to implement Collision Detection such as Raytracing, spatial partitioning or

AABB (shown in class). For the sake of the software we wanted to make sure that we reduce the amount of calculation needed and it would make sense since object in our terrain are generated procedurally.

Conclusion

This project was extremely informative, and we learned a lot about rendering and procedural generation. The greatest challenge was working with the OpenGL state machine while maintaining an OO approach. Also challenging was debugging OpenGL, especially the shader programs. The most interesting part was studying models of plant growth and terrain formation – we strongly look forward to continuing research in this area.

Although we now feel more comfortable working with OpenGL, there is a lot of progress we could make on this project, such as implementing:

- More advanced shadow techniques: cascaded shadow maps, hybrid light map / shadow map
- Water: translucency and refraction effects
- Bump maps
- Better rendering optimization, further optimizing shaders, using texture mipmaps
- More complex L-systems including contextual and stochastic implementations
- Better simulation of terrain generation using hydrological and tectonic models

We look forward to continuing to learn about 3D graphics and procedural modelling in the future.

References

Rendering

1. Learn OpenGL (http://learnopengl.com)
2. Tom Dalling OpenGL Series (https://github.com/tomdalling/opengl-series)
3. thebennybox 3D Game Engine (https://github.com/BennyQBD/3DGameEngine)
4. NVIDIA Developer (https://developer.nvidia.com/)
5. Learn OpenGL - Shadow Mapping (http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping)
6. http://fabiensanglard.net/shadowmappingVSM/

Model Generation

7. Outerra – Procedural Grass Rendering (http://outerra.blogspot.ca/2012/05/procedural-grass-rendering.html)
8. University of Calgary Dept. Of Computer Science – Algorithmic Botany (http://algorithmicbotany.org)
9. Prusinkiewicz, Przemyslaw, and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. New York, NY: Springer New York, 1990.

Terrain Generation

10. Making maps with noise functions (http://www.redblobgames.com/maps/terrain-from-noise/)
11. Generating Random Fractal Terrain (http://www.gameprogrammer.com/fractal.html)

Texture Generation

12. Synthesizing Bark (http://www-evasion.imag.fr/Publications/2002/LN02/bark.pdf)
13. Improved Noise reference implementation (http://mrl.nyu.edu/~perlin/noise/)
14. Understanding Perlin Noise (http://flafla2.github.io/2014/08/09/perlinnoise.html)
15. OpenGL.org (https://www.opengl.org)

User Interface

16. http://learnopengl.com/#!Getting-started/Camera
17. http://www.glfw.org/docs/latest/group__input.html
18. https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects
19. http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/
20. http://stackoverflow.com/questions/6083286/having-a-little-issue-calculating-the-bounding-sphere-radius
21. http://www.gamedev.net/page/resources/_/technical/math-and-physics/simple-bounding-sphere-collision-detection-r1234
22. https://xoppa.github.io/blog/using-the-libgdx-3d-physics-bullet-wrapper-part1/
23. Course lab sample code
24. Course Notes