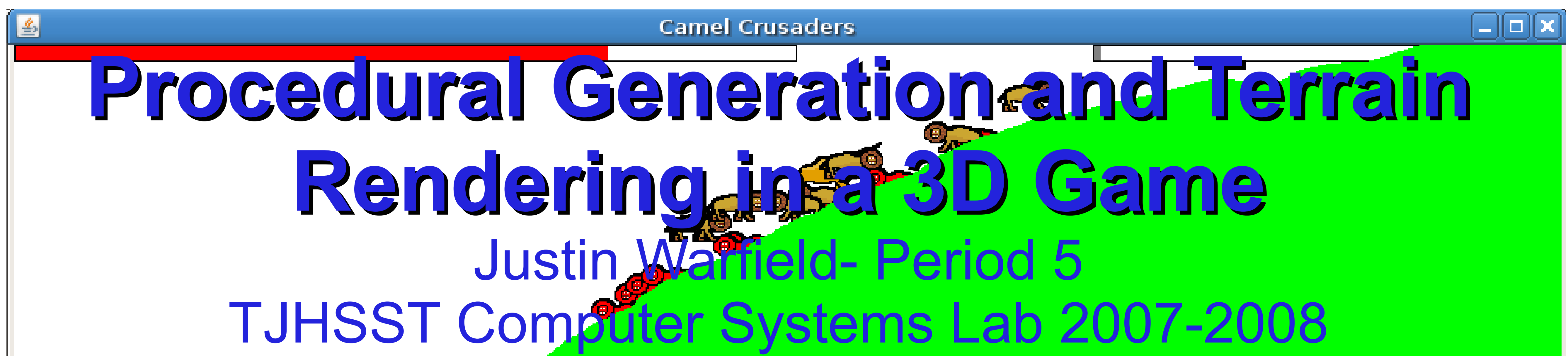


Procedural Generation and Terrain Rendering in a 3D Game

Justin Warfield- Period 5
TJHSST Computer Systems Lab 2007-2008

Abstract:

The goal of this project is to create a basic 3-dimensional video game utilizing several techniques (especially fractal geometry, multi-variable algebra, and statistical analysis) to procedurally generate terrain and game environment and render them in an efficient and effective manner.

Introduction:

Many techniques are out there for creating random terrain, which is the most common use of procedural generation. Fractal geometry is widely used in such algorithms. Similar techniques are commonly used to create random textures, such as cloudy skies and ground. The unreleased game, Spore, is expected to be groundbreaking in the area of procedural generation, using procedural algorithms to create 3D creature models and animations. The use of 3D equations to model terrain is seldom used and research is lacking, but the inherent referential transparency, speed and potential of terrain functions has drawn me to the use of multi-variable equations. My program implements procedural generation in a new way, paving the way for further testing and experimentation. Naturally, this has also led to investigation into the best and most efficient way to render the terrain generated.

Methods:

The Camel Crusaders Game:

The Camel Crusaders Game is programmed in Python, utilizing OpenGL through the PyOpenGL library. The first step was to create a generic, 1st person, 3D game. The object of the game is to move about the world, killing continuously respawning lions by throwing an endless supply of bombs at them. At the same time, the player cannot get to close to the lions or he'll lose health. Once all health is gone, the game is over. A getHeight(x, y) method is called to determine the height of the terrain at a given point, while the genEnviro(ax, ay, bx, by, cx, cy) method determines the environment for a given area—applying the proper ground texture and generating foliage. Within the CamelCrusader code, only these two methods had to be altered to compare various methods of procedurally generating the game's world. A separate drawterrain() method is used to render the terrain in various ways.

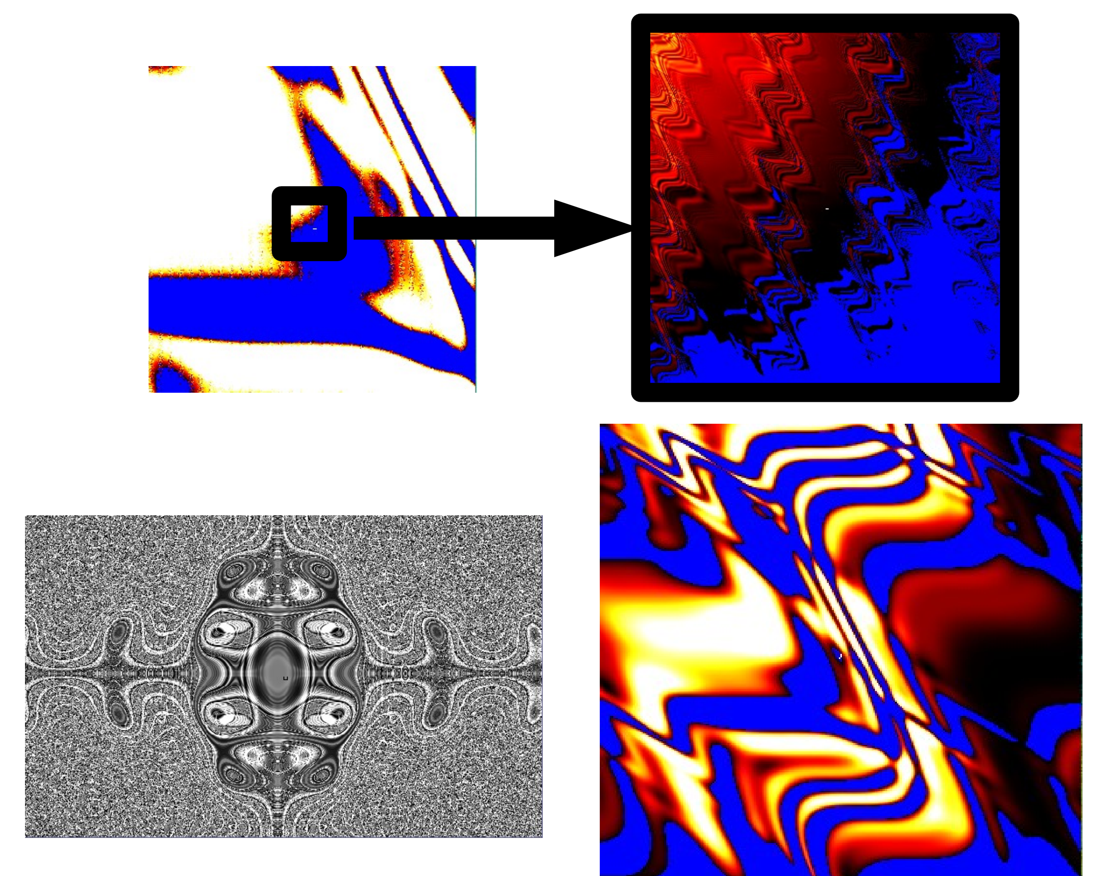
A debug framework was also put in place utilizing the various function keys. Other keys can lower and raise the water level, modify the view for an in-game map, and even change how extreme or how vast the land itself is. This enabled quick, in-game comparison and editing of the terrain itself.

Results:

Generation Methods:

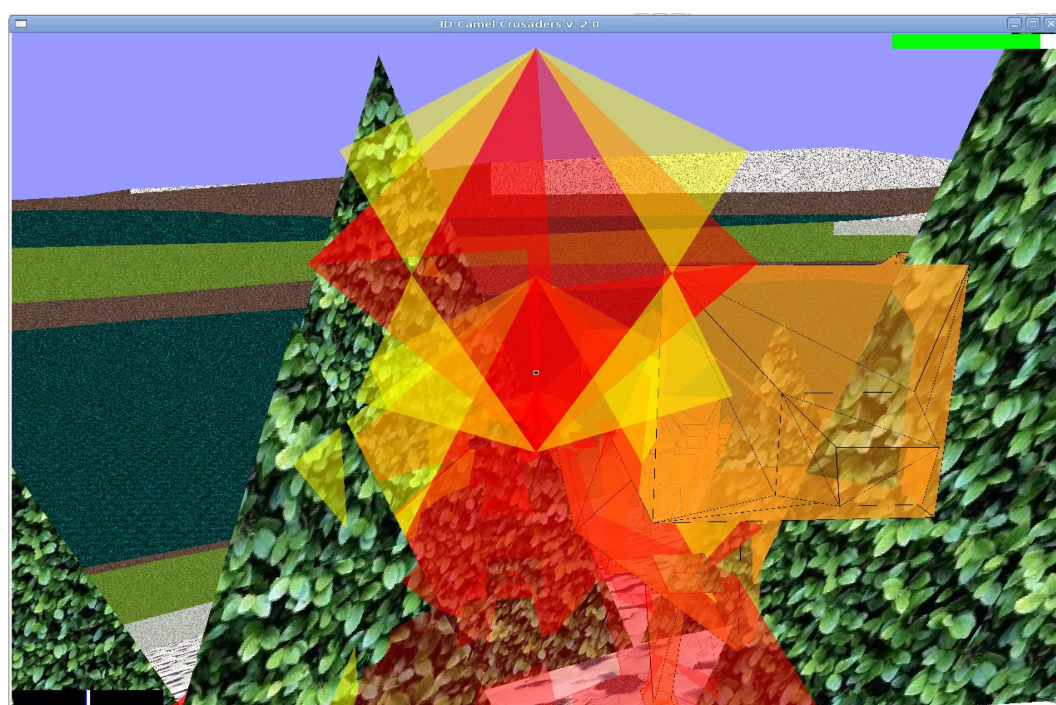
Initially, recursive fractal methods were used to generate terrain, but this required the terrain data to be generated beforehand and stored in arrays. This was time consuming and although the result was locally realistic, there was no way to insure consistency of the game map each time the game was played.

Next, an entirely new method was attempted, with little literature to be found of any previous use: function-based terrain. The function used is the sum of a sine function of x and a sine function of y, whose amplitude and wavelength values are in turn other sine functions. The resulting height map is above, on the bottom right. Two other variations were tried out as well. The top map is the sum of the original function called three times, but at three different scales. This function is much more chaotic and realistic, but at large values, the function becomes too chaotic. A second method (on the bottom right) involved calling the terrain function recursively, resulting in a highly chaotic and pseudo-random, yet consistent, landscape.



Render Methods:

Originally, terrain was simply generated by squares around the player. This method proved slow, especially since I wanted to display a wider area of the terrain, with more detail. The initial fix to this was to code the drawterrain() method so that the squares drawn were larger, the farther away they were from the player. This eliminated unnecessary detail far away, while allowing more detail nearby. However, the disparate sizes of the squares created large gaps in the rendered terrain. This was fixed by drawing the ground in concentric circles about the player instead of squares. Once ground textures were added, the seams between ground polygons became invisible. By only rendering the polygons within the players view, efficiency was also greatly increased. At first, the terrain could only reasonably be rendered within 20 units, while now the game easily renders the terrain up to 20, 000 units away.



Conclusion:

I may not have reached the area of procedural modeling, sound, or gameplay in my researches, but this is because I found a sufficient challenge in the generation and rendering of terrain. Procedural methods are unique and powerful, given that they don't need to store large amounts of data, and the scope of the land they generate is virtually limitless. Working with terrain functions and exploring various methods and equations to achieve a realistic terrain method was intriguing. From my research, there was little to no use of multi-variable equations to generate terrain. They have proved to be much more effective and efficient than fractal methods, as they are quicker to generate and exactly consistent each time the method is run. Hopefully this avenue for terrain generation will grow, with its use more thoroughly understood and its applications become better known.