TJHSST Computer Systems Lab Senior Research Project
Use of Various Techniques Implementing Procedural Generation and Rendering in Video Game Design
2007-2008

Justin Warfield

June 10, 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. The methods use generate terrain which is both locally random and realistic, while globally consistent with each rendering.

Keywords: procedural generation, midpoint displacement, referential transparency, procedural modeling

1 Introduction and Background

Procedural generation is the technique found in many computer applications where data or game content is created as the program runs, instead of storing it as data in memory. This cuts down on memory, but often increases processing time. Two factors are important to keep in mind when generating game content procedurally. First, realism and apparent randomization are important in many applications, such as in creating textures and get land heights, to ensure realism. This means common characteristics of generated content, such as symmetry and patterns, are undesirable in our gaming application. The second characteristic is referential transparency, which means that the content generated is the same each time the program is run. Although referential transparency may not be necessary for certain games, in more sophisticated games it is allows for more fulfilling gameplay. Practically, this means that, if terrain is generated procedurally, the terrain the player navigates will be the same every time the game is played.

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.

2 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 hell 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 areaapplying 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 games 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. See Appendix B for key actions.

The lions (enemies) and bombs are objects with their own Physical objects. The player, also has a Physical object. During the course of the game, for each iteration, all Physical objects are modified using the physics() method as a physics engine. Textures are uploaded and saved in an array at start up. To increase efficiency, proposed triangles to be drawn are stored in a dictionary with the keys being the textures. In this way the program can be sure not to draw the same triangle twice and can avoid having to reload textures by grouping the polygons to be rendered and rendering them in order of their textures.

3 Terrain Generation

Initially, recursive fractal methods were used to generate the terrain, but this required the terrain data to be generated beforehand and stored in arrays. A random mid-point displacement method was used to generate height values for a given area. After getting, values for the corners and center, midpoints were calculated with a small, random amount of displacement added. This method was not, of course, referentially transparent and the calculation of height values in between pre-calculated points was tricky.

A new method was added called terrainfunc(x, y), which could be used in place of the fractal method depending on how a flag was set. The speed of simple sine functions was significantly faster and more accurate. Also, referential transparency is perfect when using multi-variable functions. The old method ran at about 10 fps, while the new method reached 60 frames per second. After experimenting with various polynomic and trigonometric functions, it became clear that trigonometric functions had potential to be the most realistic. Little to no literature is to be found on the subject and potential of function-based terrain, but the speed and consistency is admirable. The function accumulated more factors as it grew in an attempt to make it more realistic and unpredictable. In the end, the final function is the sum of a sine function of x and a sine function of y. For each of these two functions, both the amplitude and wavelength are in turn additional sine functions. An additional sine function, the ruggedness of the terrain, is multiplied to the final result. One problem that remains to be solved regarding the final function is that at distance points from the origin, resulting values for the height function become more extreme to the point that the computer can freeze up. The function itself is only a series of nested sine functions, so at distant values the range shouldn't approach infinity. This may be due to a round off error of some sort, but has yet to be fully investigated.

3.0.1 Final Height Equation

```
scale = 10

absx = absx + 10000*math.cos((absx+absy)/10000.0) + 30000
absy = absy + 3700*math.cos((absx+absy)/3700.0)

htx = 45*scale*math.sin((absx/(1000.0*scale)))
widx = 200*scale*(2+math.sin((absx)/(1000.0*scale)))
terrx = htx*math.sin((absx)/widx)

hty = 45*scale*math.sin((absy/(1000.0*scale)))
widy = 200*scale*(2+math.sin((absy)/(1000.0*scale)))
terry = hty*math.sin((absy)/widy)

roughness = 0.5+0.5*math.sin(absx/16621.0)
xprop = 0.5
yprop = 1-xprop
fht = (terrx*xprop + terry*yprop) * roughness
```

Two other variations were tried out as well. In one, the original function is called three times, but at different scales. This function is much more chaotic and realistic, being the sum of local, normal, and global scaled functions. However, due to the previously stated extreme value problem, the small scaled function encountered these extreme values at a much closer distance to the origin. This new function was also significantly slower than the others. With the addition of textures, the fps was down to about 3 with this function. The complex hills, mountain ranges, island chains, and coastlines created by this function are still remarkable and if the extreme value error is fixed and speed increased, may be by far the most realistic. A second method involved calling the terrain function recursively, by taking the resulting height the first time and calling the method again with the calculated height values as the location. This resulting in a highly chaotic and pseudo-random, yet consistent, landscape. In this circumstance, although apparent randomization is strong, realism is compromised, resulting in a less than satisfactory function.

3.1 Environment Generation

Recently, I've been able to start work on procedurally generated environment. Free downloaded textures are used in place of procedurally generated textures which were never coded. Five different terrain textures were used, as well as water, and some basic trees. Basic methods of determining textures in a given area were used in the genEnviro() method. Grass is the default texture, used when other textures do not meet their requirements. Rock textures are drawn when the terrain is steep (when the slope is greater than two). Dirt textures are used when the land height is less than the water height, or within two units of it. The snow texture is used when the land height is greater than 200 and a transitional, grassy snow texture is used on altitudes between 150 and 200 units. More sophisticated methods could easily be implemented, but the priority of this project became the land height methods. Trees are also drawn as planes, which are always oriented to face the player. Whether they are drawn is determined by a modded polynomial function: treeht = 5+(abs(treex*treey*treey+treex*treex*treey)) mod 7.

4 Terrain Rendering

Initially, the terrain was rendered in squares, using just the data generated and saved in the array. The common practice of drawing farther things with less detail was implemented once the method of generation switched to the terrain function. This allowed much more terrain to be rendered. When before only 20 units in each direction were rendered, now terrain is rendered in almost 20,000 units in every direction. Because of the different sizes of the polygons, large gaps between them became a common problem. A different technique was used to eliminate the gaps in the rendered terrain. Instead of rendering squares about the origin, polygons were rendered around concentric circles. With the addition of textures, seams between the polygons rendered are indetectable. To increase efficiency, only terrain in front of the user is rendered. Water is also rendered when the land height is less than the water height. Also, when the player is underwater, visibility decreases and the physics change accordingly.

5 Results and Discussion

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. Before much else could be implemented, new ways to increase running speed are necessary, with the game currently running at six frames per second. Procedural methods are useful and effective, given that they dont 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. The most promising of these methods was calling the function under three different scales and summing the result, creating the most unique and varied landscapes. Hopefully this avenue for terrain generation will grow, with its use more thoroughly understood and its applications become better known.

Appendix A. Game Controls

Key	Action
Click and drag	Change view direction
w	Accelerate Forward
s	Accelerate Backwards
a	Accelerate Left
d	Accelerate Right
t	Accelerate Extremely Fast
y	Stop player (set velocity to zero)
Space	Jump
e	Drop a bomb
f	Throw a bomb
g	Launch a missle bomb
r	Detonate live bombs
q	Quit
p	Toggle the map
-	Zoom in on map
+	Zoom out on map
.[	Make the map bigger
]	Make the map smaller
.	Broaden the map's height scale
,	Narrow the map's height scale
1	Decrease player's acceleration
2	Increase player's acceleration
5	Shrink the terrain
6	Increase the vastness of the terrain
7	Make terrain less dramatic
8	Make terrain more dramatic
9	Lower water level
0	Raise water level
o	Toggle the trees
n	Draw less terrain
m	Draw more terrain
u	Set player on ground
j	Freeze all active lions
k	Toggle cheat mode (flying with Space bar)
F1	Print input debugging
F2	Print location debugging
F3	Print system debugging
F4	Print other temporary debugging
F5	Print track debugging
F6/F7	More location debugging
F11	Print all debugging
F12	Turn off debugging