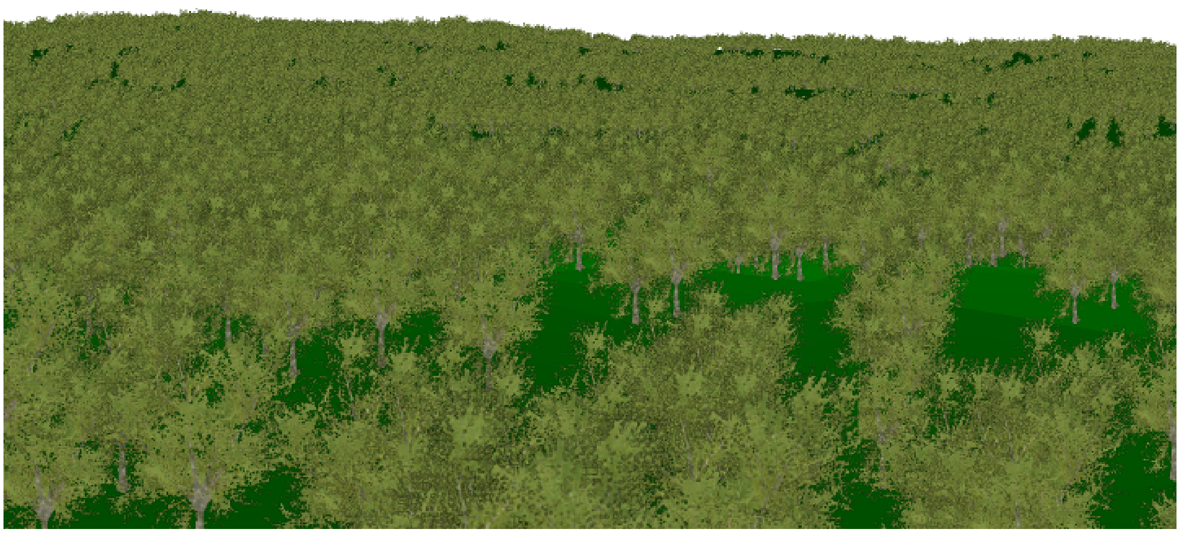
# Advanced Computer Graphics Final Project Report

Auston Sterling: sterla@rpi.edu
Brendon Justin: justib@rpi.edu



## 1 Introduction

For our final project, we implemented a system for rendering forests of trees in real time. We are using the method outlined in Eric Bruneton and Fabrice Neyret's *Real-time Realistic Rendering and Lighting of Forests*[1], specifically their z-field method for rendering trees of apparent sizes larger than a few pixels. Trees are often difficult to render in real-time applications as their geometry is complex, though simplified methods are easy to identify as lower quality. Our program provides fast and accurate rendering of trees as well as a fractal based landscape to simulate a forest. In this paper, we explain the motivation for this model, then describe the algorithms used. We finish up with some ideas for future work and some of the bugs we encountered.

## 2 Motivation

Trees are often a plentiful part of landscapes, and so rendering them at the same level of fidelity as foreground objects can make real-time rendering impossible. At the same time, lower fidelity objects can detract from a scene's appearance, even if they are part of the background.

Moderate quality tree models can be 250,000 or more triangles[2]. Just 20 of such trees would involve 5 million polygons. For comparison, a frame in the 2007 game Lost Planet can have 3 million polygons in a frame. Clearly, conventional rendering methods do not permit the use of such tree models in games. Games must instead rely on include low quality tree models or other workarounds of reduced fidelity.

Bruneton and Fabrice [1] implemented a fast means of rendering high quality mid-distance trees, displaying 180,000 such trees at over 30 frames per second. We hoped to duplicate their success, rendering tens of thousands of trees across a landscape in real time.

# 3 Algorithms/Techniques

## 3.1 Terrain Generation

The ground under the trees is dynamically generated at runtime when the program is run. The diamond-square method of fractal terrain generation is used to create a height map, which is then applied to a square grid to create varied terrain. Points are set to the average height of particular surrounding points, then a random offset is added for variation. The offset is generated uniformly at random on the range $(-x, x)$, where $x$ is set before the algorithm is run, and is reduced by $2^{-y}$ each iteration, where $y$ can be adjusted. $x$ is directly related to the maximum and minimum heights produced, and $y$ is inversely related to the steepness of generated terrain. Lengths in the algorithm are the number of points between the current point and the other point in consideration, and can only be considered in horizontal or vertical lines. Points which are adjacent form a line of length zero. The diamond-square method in pseudocode [3] follows:

```
Start a square size equal to that of the whole grid
While the length of the side of the squares is greater than zero {
  Pass through the array and perform the diamond
  step for each square present.
  Pass through the array and perform the square
  step for each diamond present.
  Reduce the random height offset range.
  Reduce the square size by half.
  Divide each square into non-overlapping squares of the
  current square side length.
}
```

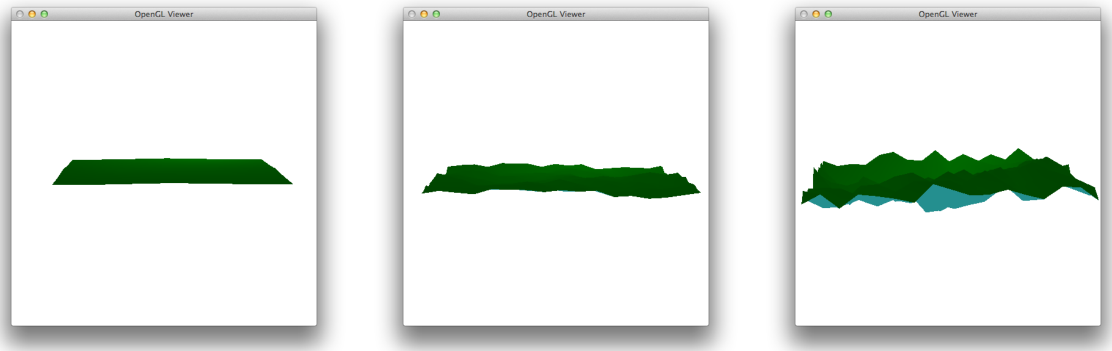
The diamond step consists of finding the point in the center of the current squares corners, setting its height to the average of those points heights, and adding the random offset. The diamond step is so named because the five points considered look like four diamonds meeting at the center point. In the square step, for each diamond meeting at the center from the diamond step, find the center point. The points height is set to the average of the points from the diamond shape, plus the random offset [3].

## 3.2 Tree Distribution

Trees are placed in each square of the world grid independently, and their positions varied in a set pattern based on the number of trees in that square. Small random perturbations, inversely proportional to the number of trees in that square, are added to each position to avoid producing a regular grid. The number of trees in each square in the terrain grid is a Poisson random variable; the Poisson distribution was chosen because it works with integer values, has an equal mean and median, and only requires one parameter. After terrain is generated, trees are vertically positioned to sit directly on the ground.

## 3.3 View Computation

In order to ensure high frame rate during run time, views of the tree are pre-computed. Cameras are set on the upper hemisphere around the center of the tree and scaled to properly capture the whole tree. The




(a) Terrain, y = 1.0

(b) Terrain, y = 0.8

(c) Terrain, y = 0.5

Figure 1: Terrain generation with varying jaggedness

tree is then drawn to a Frame Buffer Object for each view which is saved to a texture to be used later. One significant challenge is the proper distribution of camera locations on the hemisphere. While there are many methods to accomplish this[4], our algorithm evenly splits the hemisphere into rings from the equator of the sphere up to a point at the top. Each ring is then divided up into an equal number of points. This results in effectively an even grid with one axis being angle on the $XZ$ plane and the other axis being the angle up along the $Y$ axis from the $XZ$ plane. The points near the top of the hemisphere end up tightly clustered; though the camera angle still shifts noticeably around the $Y$ axis, it is not reflected as strongly in its position.

### 3.4 Displaying Trees

For each tree in the forest, a quad is displayed and rotated around its center to face the camera. The view of the tree which is nearest to the angle between the tree and the camera is drawn to the quad. There is noticeable popping between the views since no interpolation is used, but with 300 views the transitions are smooth. Selection of the nearest view can be accomplished with simple arithmetic using the grid-like distribution of camera positions, while quad placement requires two vector cross products and some vector addition. These are some of the only computations that need to be done with each frame, allowing for interactive control of scenes of thousands of trees.

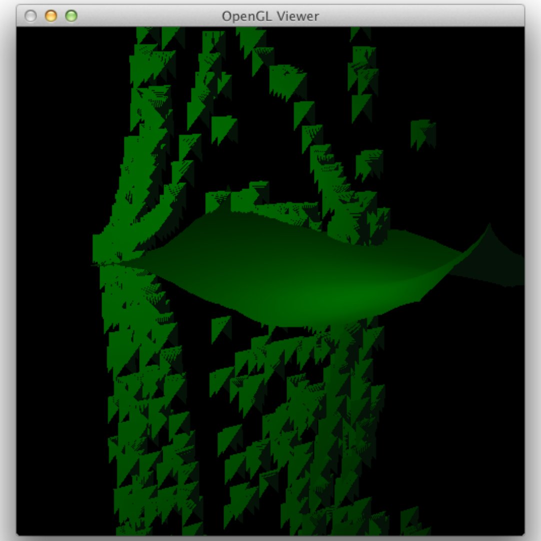
## 4 Results

### 4.1 Terrain Generation

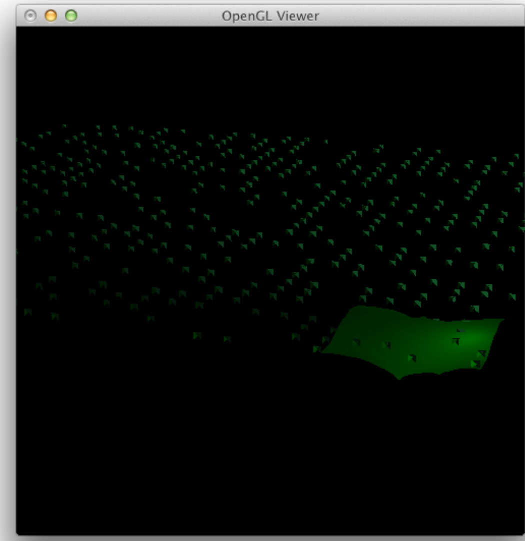
The fractal terrain generation algorithm produced terrain with plausible variations in height for nearly level ground. Figure 1a shows relatively flat ground generated, while Figure 1b and Figure 1c show greater height variation and more jaggedness as the $y$ height offset parameter is varied. All three images have a $x$ parameter proportional of the square root of the terrain area. The generated terrain appears smooth for $y$ values of 0.8 or greater, as in Figure 1a and Figure 1b.

### 4.2 Tree Distribution

Tree distribution, when combined with terrain generation, produced several incorrect results before completion. Figure 2a shows an error in multiplying the current square coordinates with the specified height for each tree, moving the trees far away from the terrain surface. Figure 2b demonstrates an issue where tree

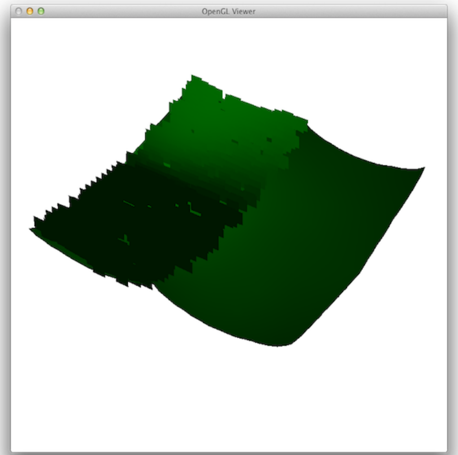



(a) Tree heights

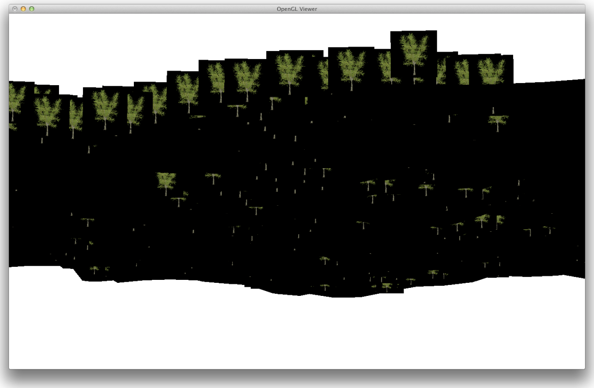



(b) Grid scaling

Figure 2: Tree placement bugs

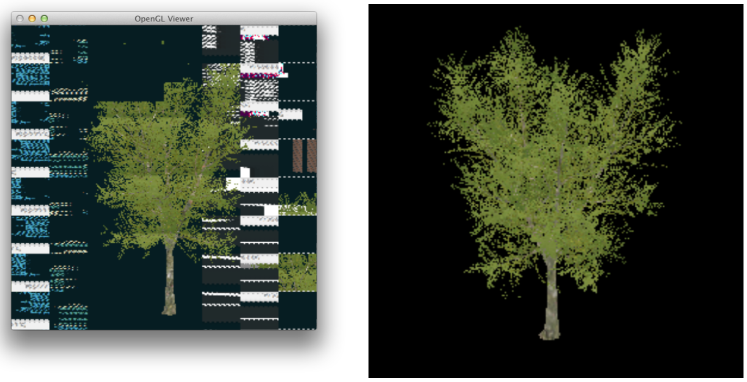


(a) Half coverage



(b) Underground trees

Figure 3: Errors in the process of placing trees



(a) View Artifacts (b) A Standard View

Figure 4: Tree Views

positions were specified in absolute coordinates, but did not take into account terrain scaling below an area of 1 unit for each square. Figure 3a shows the largest remaining error in distribution, where only half of the squares in the terrain have trees placed and rendered. Figure 3b shows the other outstanding error in distribution, where trees are sometimes not placed at the same height as the terrain.

### 4.3 View Calculation

After a few days of attempting to ray trace views of the tree, which took nearly six hours per view, rendering to a Frame Buffer Object was a much simpler task. Figure 4a shows a quick issue where the buffer was not properly cleared before it was drawn to, but that was an easy fix. View calculation now only takes a few seconds for to compute 300 views.

### 4.4 Alpha Blending

Generally, using alpha-blended textures requires that they be drawn in sorted order from back to front. This is because when drawing a textured quad near the camera, the depth buffer is still updated for the transparent fragments. When the quads further back are drawn, they fail the depth test and are ignored. If this sorting does not occur, there will appear to be boxes around each image (see Figure 5). Sorting each tree by distance from the camera would be time consuming and harm the frame rate of the program. The problem is still unsolved, although another solution would be to use a fragment shader to specifically drop transparent fragments.

## 5 Conclusions

We have implemented automatic view generation in a hemisphere around an object, the automatic selection of the nearest such view, fractal terrain generation with adjustable height variation, distributing trees with varying density, and displaying trees in the correct orientation to the camera. The program runs in real time for smaller cases and shows promise for its goal of rendering enormous numbers of trees quickly and accurately.

## 6 Future Work

Future work includes using a different texture for each tree represented in the scene, interpolating multiple views to create arbitrary views, all aspects of lighting effects and shadows, and rendering distant trees more

Figure 5: Alpha Blending Problems

efficiently.

Texturing each tree differently will require a change to shader programming, from the OpenGL fixed function pipeline. Shaders will enable the use of array textures, with a limit of 2048 compared to the limit of 160 textures that the fixed function pipeline allows on the NVIDIA GTX 470 [5]. Interpolating multiple pre-rendered tree views will also require the use of shader programming, performing calculations on the GPU to maintain real-time calculations and rendering.

Lighting effects and shadowing are not implemented at all, and are a clear opportunity for improvement. Shadows from trees on the ground, from leaves on other leaves and trees, surface light scattering and hotspotting in leaves would all increase the plausibility of rendered scenes.

Distant trees can be rendered more efficiently as in Bruneton and Neyret's work [1], using terrain maps and shaders. Distant trees around the size of one or two pixels could be rendered with reduced computation time by implementing their techniques.

# References


[1] Bruneton, E.& Neyret, F. *Real-time Realistic Rendering and Lighting of Forests*. Computer Graphics Forum **31.2** (2012), available at http://hal.inria.fr/hal-00650120/en.

[2] available at http://www.sharecg.com/v/36271/gallery/5/3D-Model/Alder-Tree-OBJ?interstitial_displayed=Yes.

[3] Paul Martz *Generating Random Fractal Terrain* available at http://gameprogrammer.com/fractal.html.

[4] Dave Rusin *Topics on Sphere Distributions* available at http://www.math.niu.edu/~rusin/known-math/95/sphere.faq.

[5] WildfireGames *OpenGL capabilities report: GTX 470* available at http://feedback.wildfiregames.com/report/opengl/device/GeForce%20GTX%20470