

Assignment 3 Part 2 COMP 557

Instructor: Prof. Michael Langer
Teaching Assistant: Fahim Mannan
Posted: Friday, Nov. 21, 2008.
Due: Tuesday, Dec. 2, 2008 at midnight.

Overview

- We provide you with code for generating a fractal terrain using the midpoint displacement method. We also provide near-complete code for computing level of detail, and for rendering the terrain. Your task is to modify the code by changing how level of detail is computed, and how the terrain is rendered.

Note: For you to carry out this assignment, you should *not* attempt to understand all the details of the code. Time constraints do not permit this. Instead, it should be sufficient for you to understand what the various functions do, and roughly what data structures they use.

- There are 15 points for this assignment (Part 2).
- Late assignments will be penalized by 4 points per day.
- Hand-in instructions are the same as in previous assignments.

Level-of-Detail and Procedural Texturing

Introduction

As discussed in lecture 11, level-of-detail (LOD) is an important technique for reducing the number of triangles rendered for large meshes (e.g. terrains used in computer games and GIS applications). One of the most popular and easiest way of representing a terrain is to use a regular grid, where each grid cell stores a height value. In this assignment, you are given code¹ that implements level of detail using a height field defined over a square grid.

To render a triangulated mesh with different levels of detail, we implicitly represent the surface as a Binary Triangle Tree (BTT). The lowest possible level of detail of the surface is approximated by a single large square. This is divided into four triangles which are each represented using BTT. In a BTT, every node (internals and leaves)

¹The code is based on an algorithm that is inspired (though not identical to) an algorithm described in the paper - "Real-Time, Continuous Level of Detail Rendering of Height Fields", by P. Lindstrom *et al.* SIGGRAPH 1996, <http://www.cc.gatech.edu/~lindstro/papers/siggraph96/siggraph96.pdf>.

represents a triangle. Each internal node has two children. (This is analogous to Quadtree data structure but instead of quads we have triangles, and each non-root internal node has two children rather than four.) The two children triangles are formed by splitting the parent triangle, namely splitting a right angle isosceles triangle from its apex to the middle of its hypotenuse. The two children are thus also right angle isosceles triangles (see figure 1).

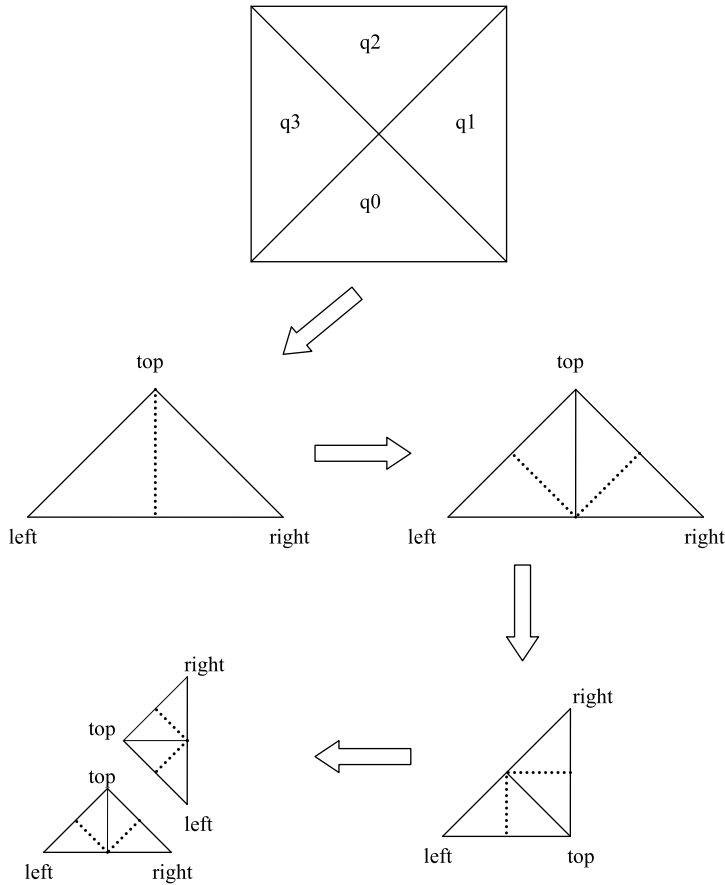


Figure 1: Triangle Tree. The initial square (lowest level of mesh) is divided into four triangles each of which are represented using BTT. The solid line through the middle represents a split and the dotted line shows the split for the next level.

From the triangle tree described above, we can also define *vertex tree* (figure 2), whose nodes are vertices in the mesh rather than triangles. Like the triangle tree, the vertex tree is defined implicitly only, as follows. As discussed above, when a triangle is split, the split occurs along an edge between the apex vertex of the triangle and

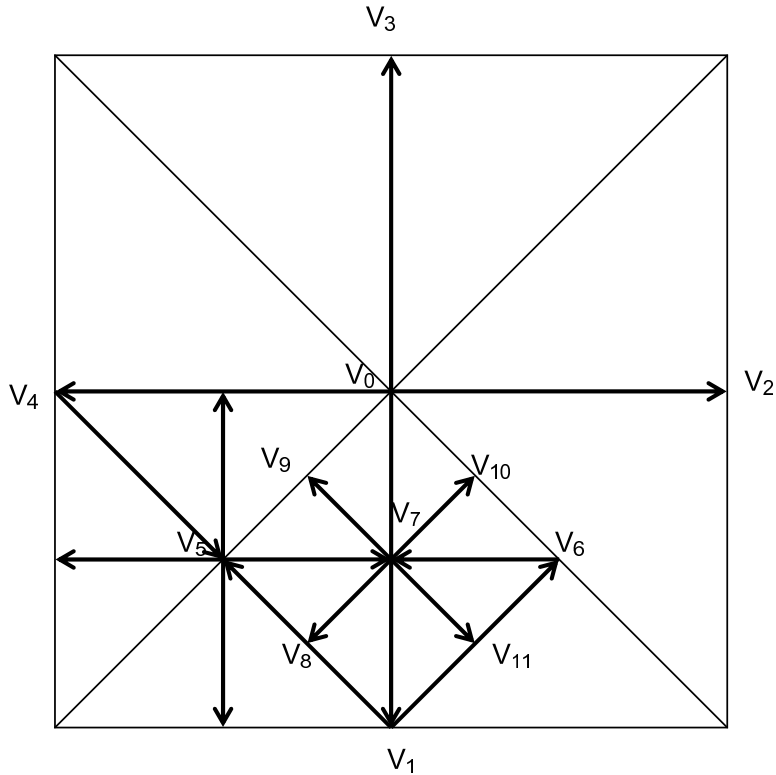


Figure 2: Vertex Tree showing parent child dependencies. An arc from A to B indicates A as a parent of B. Every node (except for leaf and internal nodes that are on the boundary) has four children (e.g. V_0 has V_1 , V_2 , V_3 and V_4) and two parents (except for the root) (e.g. V_7 has parents V_5 and V_6).

the midpoint of the hypotenuse. In the vertex tree, the apex vertex is a parent and the hypotenuse vertex is its child. As is evident in the above figure, and as discussed later (see the `isEnabled()` function), the hypotenuse can be vertically/horizontally oriented, or it can be diagonally (± 45 degrees) oriented, with the two alternating between levels of the tree. Also note that a vertex in the mesh may give rise to multiple vertices in the vertex tree since, if two triangles share a hypotenuse, then the apices of the two triangles will have the same child vertex.

The LOD algorithm works as follows. For each vertex in the mesh, determine if the triangle which has that vertex as its apex should be split into two triangles. If so, such a vertex is **enabled**. A vertex is enabled (here's the definition) if any one of its descendants in the vertex tree satisfies some criterion chosen by the user, in which case that descendent is **active**. This leads to a recursive definition: a vertex is enabled either if it is active or if one of its children is enabled (see function `isEnabled()`). Each cell of the terrain grid has an enable bit that is computed prior to triangle generation, and then stored as a flag along with the height field. Then, to render the

mesh, we traverse the vertex tree (which is equivalent to traversing the square grid) and render triangles based on the enable bit.

Requirements

The questions require you to generate very little new code. The main challenge, rather, is to understand enough of the existing code so that you know what needs to be added.

1. (6 points)

Determine whether a vertex is active or not by modifying the `isActive()` function in `terrain.c`, and based on the criteria that a vertex is active only if it projects to a 100×100 pixel region at the center of the window². This gives a higher level of detail in the center.

Note: in the starter code, the level of detail is set by the vertex's Z distance from the camera. (See `isActive()` code.)

Hint: you need to consider the point in normalized device coordinates (i.e. `projection * modelview`) and you need to consider the viewport size.

2. (3 points)

Modify the terrain such that all height values below a given height threshold are clamped to that threshold. The constant height surface that results will be the water surface in the next question. See comment in function `createTerrain()`.

3. (6 points)

In the starter code (in `renderTriangle()`), the specular and diffuse maps have been generated using midpoint algorithm.

Change this code so that you now generate diffuse, specular and shininess values based on the average height of the triangle that is being rendered. (Triangles are rendered by calling `renderTriangle()` in `terrain.c`). You need to consider four types of materials - water, grass (or moss, other living green matter), rock and snow. The water surface is defined in the previous question. Each other surface types is restricted to its own interval of heights above the water i.e. the heights are partitioned into different materials.

You should briefly describe in the README file how you chose material reflectance values. Your grade will depend on the appropriateness of your choice e.g. see the commented out "sample snow".

²Such a level of detail criteria could be used in an application in which a user's eyes are tracked, and high level of detail is rendered only near the screen locations where the user is looking. (The human eye has a greater concentration of photoreceptors near the center of the field of view. That is why you are able to read the words you are looking at, but you are not able to read words in the periphery.)

Starter Code:

You mainly have to work with `isActive()` and `renderTriangle()` functions which are implemented in `terrain.c`. Here we provide a brief overview of the other important functions.

1. Loading/Creating Terrains

The first two of the following functions are implemented in `terrain.c`. The third is implemented in `mpd.c`.

- `loadTerrain()`
This function allocates necessary space and loads a terrain from a pgm file.
- `createTerrain`
This function allocates necessary space and calls a user-defined terrain generator to generate the height field values.
- `midpointAlgorithm`
This function generates the heightfield using the Midpoint Displacement Algorithm.

2. Rendering and Level-of-Detail

All of the following functions are implemented in `terrain.c`.

- `renderTerrain()`
This function assumes that the terrain is already created or loaded and stored as heightfield data. Since we are doing view-dependent LOD, the function performs LOD calculations only when the current view is updated (ie. the camera moved). This is done by calling the `isActive()` function for each vertex (this will update the active flag that is associated with each vertex). Finally, the terrain is rendered by dividing the quadrilateral mesh into four triangles and calling `splitTriangle()` for each of these triangles.
- `splitTriangle()`
This function first checks if the root vertex of the triangle tree is enabled. If so, then the Binary Triangle Tree is traversed in post-order manner (ie. the processing order is left child, right child then root). The parent triangle is rendered only if none of its children were rendered. The order of the vertices are shown in the figure.
- `renderTriangle()`
This function renders a given triangle. The triangle vertices are passed as arguments to the function and these are used for determining the diffuse, specular and shininess values of the triangles (see Question 2).

- **isEnabled()**

The `isEnabled()` function recursively checks if the current vertex is enabled or not. This flag needs to be determined only once for each vertex after the viewpoint has changed. We keep track of this by initially setting the flag to -1 for all vertices. Once the enable bit is determined, it is set to either 0 or 1. To determine the enabled bit, we first check if the vertex is active. An active vertex is always enabled. However if it is not active it might still be enabled if one of its descendant is enabled. A vertex always has four immediate children in the vertex tree and two parents unless it is on the boundary of the original square.

In the implementation, we do not have any explicit datastructure for denoting the parent-child relationships, either in the triangle tree or the vertex tree. However, the parent-child relationship can be determined, given a vertex and the level (distance from the root node). The edge joining parent to child is horizontal/vertical if the `level` is even, diagonal if `level` is odd. The distance of the child node from its parent is $2^{((level/2)-1)}$. Note: Note: The grid has size $2^{(L/2)} + 1 \times 2^{(L/2)} + 1$, for some L which can be modified by the key 'L'/'l' (see following table).

- **isActive**

This function simply checks if the current node satisfies some criteria. It may be distance of the point from the camera or if the vertex is within some bounding volume (eg. the view frustum), etc (refer to Question 1).

3. Keyboard and Mouse Controls

The terrain is navigated using a 6 degree-of-freedom (x, y, z, pitch, roll, and yaw. ("Pitch" is rotation about x , "yaw" is rotation about y , "roll" is rotation about z .)

The camera controls are as follows -

Mouse Left + Move	Rotate about x and y axis (pitch and yaw)
CTRL + Mouse Right + Move	Rotate about the camera's z-axis (roll)
Mouse Right + Move	Zoom in/out
CTRL + Mouse Left + Move	Move the camera up/down or sideways
Keyboard Left/Right	Rotate about camera's y-axis (yaw)
Keyboard Up/Down	Rotate about camera's x-axis (pitch)
'f'/'F'	Zoom in
'b'/'B'	Zoom out
'w'/'W'	Toggle Wireframe
't'/'T'	Create a new terrain
'l'/'L'	Decrease/Increase the size of the grid
'r'/'R'	Reset the camera (top-view of the terrain)