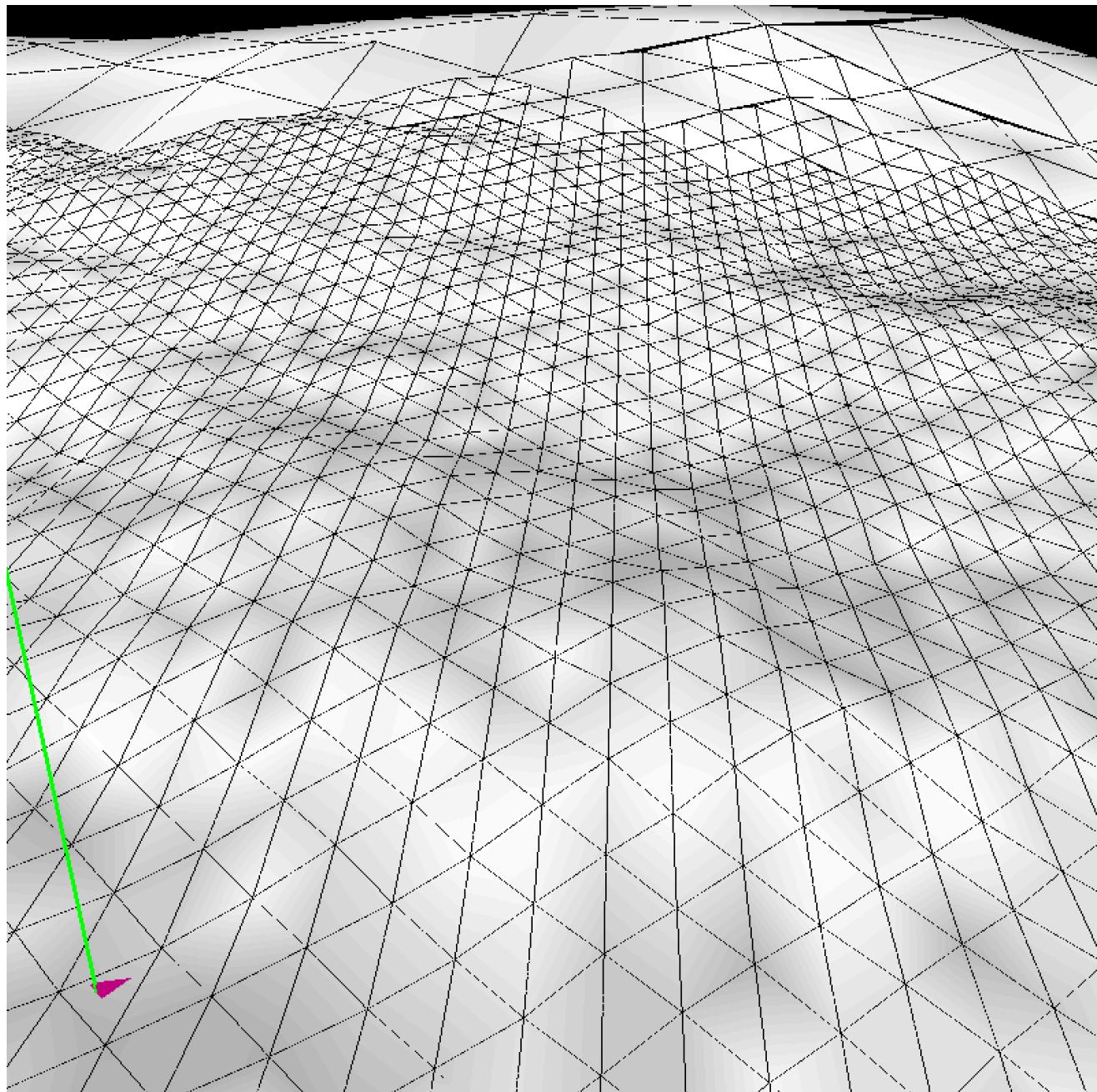


## QUADTREE PROCEDURAL TERRAIN FOR AGENT-BASED SIMULATION



# Quadtree LOD Procedural Terrain

Development Documentation

Prepared by: Eugene Ch'ng ([www.complexity.io](http://www.complexity.io)) | [www.github.com/drecuk](https://www.github.com/drecuk)

September 15, 2018

# OVERVIEW

This article documents the implementation of a Quadtree Level of Detail (LOD) procedural terrain. A quadtree is built up of nodes which continually divide into four child nodes, each child node being divided into more child nodes, and so on and so forth. Quadtree can be used for various purposes, for optimising agent-based simulation interactions by partitioning spaces so that only agents within the vicinity, i.e., agents that are local interact with each other. This reduces computational complexity from  $O(n^2)$  to  $O(\log n)$ . Quadtrees are also used for partitioning landscapes so that only agents within the local space interact. Here, we document the implementation of a quadtree structure which a C++ terrain class uses to decide which node to display based on the camera position.

## Introduction

The code presented here is now a 'systems-level' development, as a continuation of the series of C++ code in my repositories for the ERC 'Lost Frontiers' Advanced Research Grant. The Quadtree Terrain System is composed of a set of decoupled classes, each for managing objects within the ABM system:

- main.cpp - main code tying everything together
- QTTerrain.h/cpp - a terrain rendering system
- TerrainQuadTree.h/cpp - a quadtree data structure used for managing the procedural terrain
- Camera.h/cpp - a simple camera for moving around the virtual space
- Grid.h/cpp - a simple grid used for orientation
- MoveableOnQTTerrain.h/cpp - an agent used for skating on the surface of the quadtree terrain
- Vector3f.h and Matrix4x4.h are 3D math classes and utilities

## Quadtree Terrain Features

- Procedural terrain rendering based on a view-range of the camera, and level of detailing based on distance
- Terrain functions for calculating surface normals, raycast intersects of planes, etc.
- A quadtree data structure built for generating boundaries, quadtree layering, vertex indices, etc.

# CONSTRUCTING THE QUADTREE TERRAIN

## TerrainQuadTree.h and TerrainQuadTree.cpp

The class above builds and maintains the quadtree structure. Each node in the structure is described by a **struct TERRAINQUADTREENODE**. ID and parentID links the current node to the parent, and branchIndex stores the IDs of child nodes. The rest are explained in the code. The important component is **struct VERTICEINDEXINFO**. This describes the points from which OpenGL quads (terrain surface) are constructed. Each node has  $3 \times 3$  vertices which **QTTerrain.h** uses to draw **GL\_TRIANGLE\_STRIP**.



## Quadtree Levels and Triangles

Let's see the spatial and layer structure of the quadtree used in this document (Figure 2). In layer one, there is only one node, the root node. It has  $3 \times 3$  vertex index making up 4 quads. Each quad has 2 triangles (dotted) joined as a GL\_TRIANGLE\_STRIP. The quadtree cascades into the next level (layer 2), composed of 4 quads, etc. The algorithm and formula below calculates the number of nodes in a 3 layer quadtree data structure.

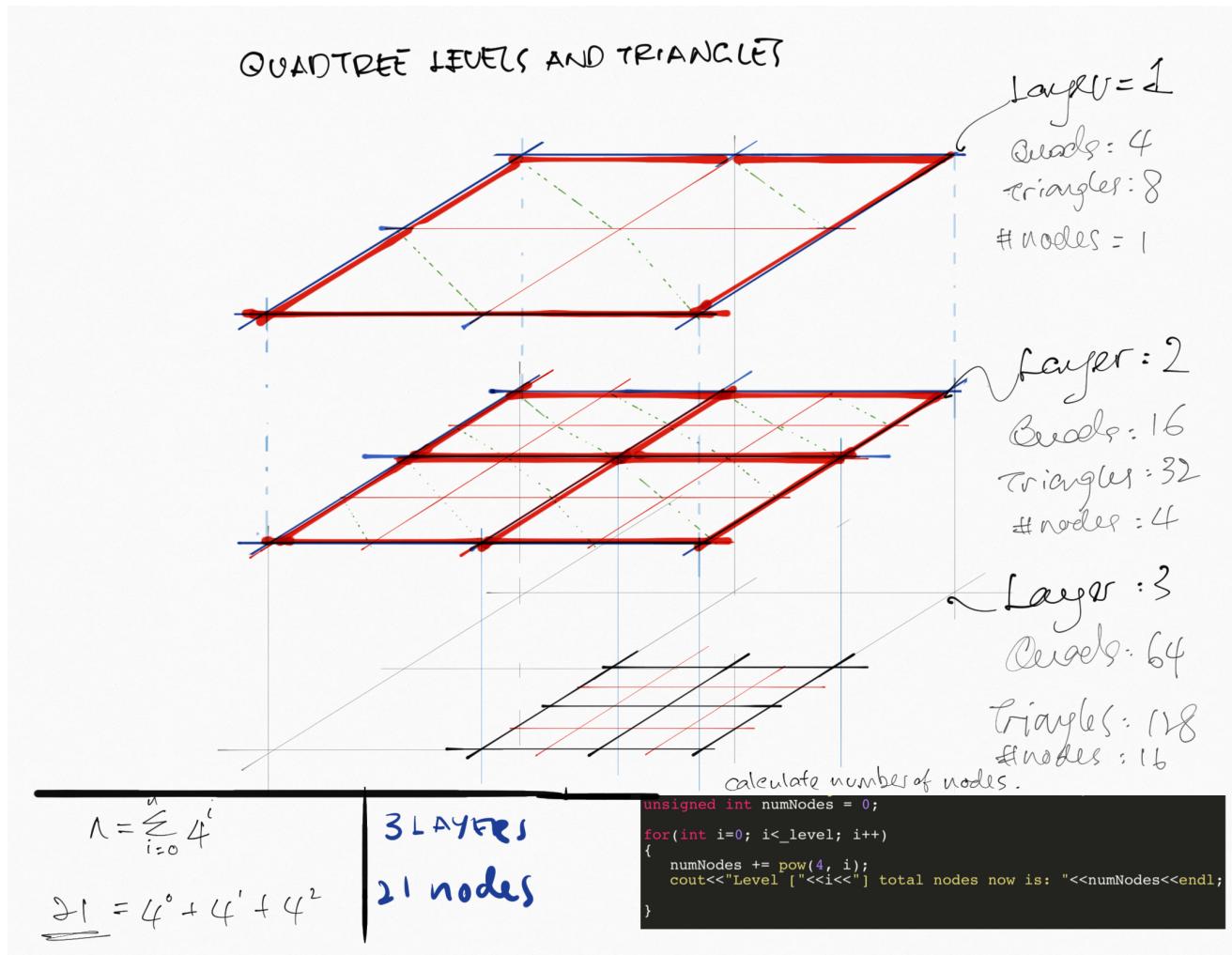


Figure 2. Quadtree levels and triangles

## Quadtree Structure of Nodes

How does the conceptual and actual structure looks like when each node is described with an ID? Figure 3 shows the structure, comparing two quadtree - one with 2 levels and the other has 3 levels. Take note of the sequence of node IDs cascading down into the leaf nodes on level 3.

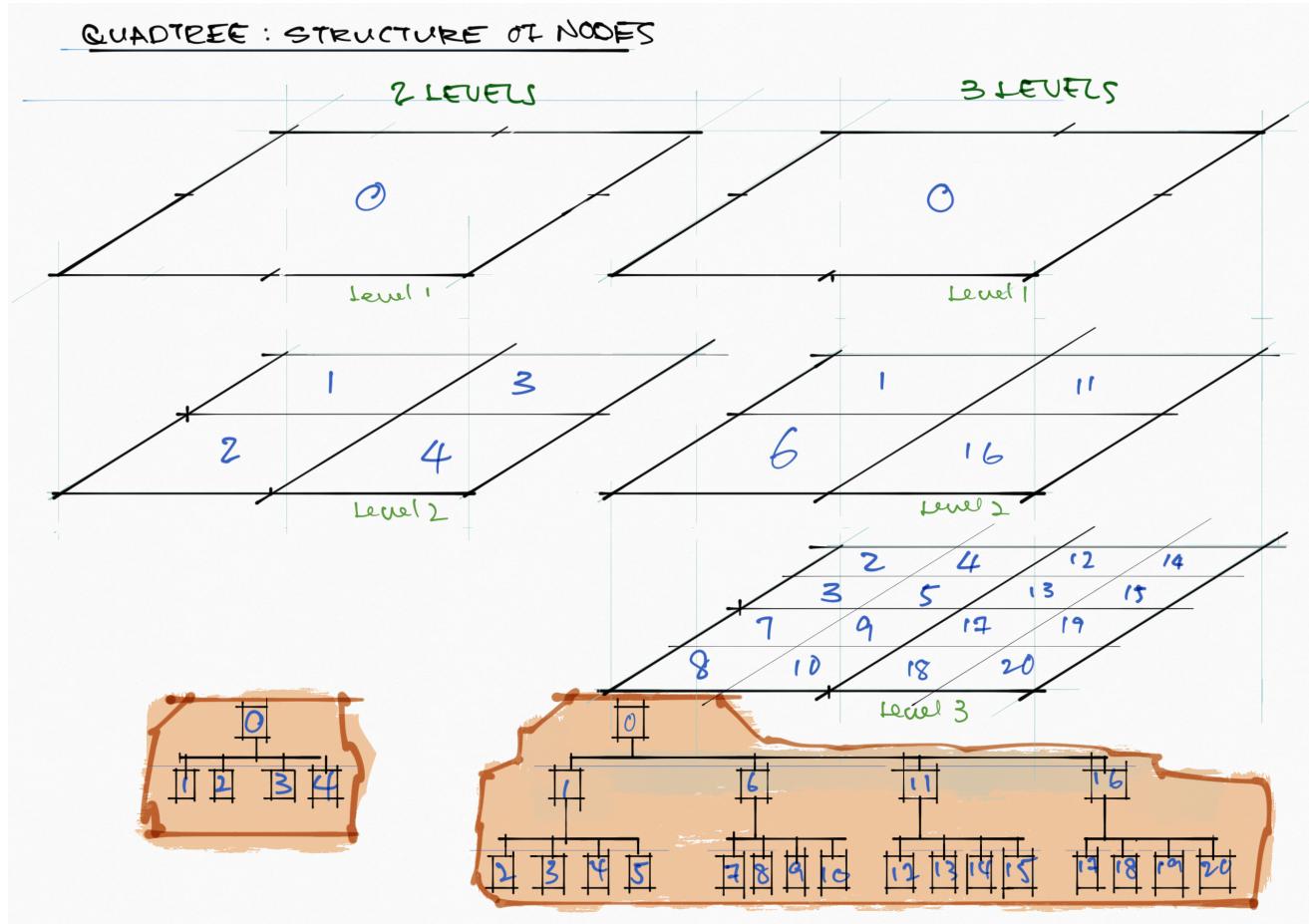


Figure 3. Quadtree structure and how node IDs are cascaded.

**TerrainQuadTree** receives 7 parameters - top, bottom, left and right describing the boundary of the terrain, and \_vertX and \_vertZ defines the number of vertices spanning the x and z. **\_vertX** and **\_vertZ** read from the RAW files in the code - 512 x 512 resolution. **\_level** defines the depth of the tree, number of levels to create for the terrain.

**calculateNodeSize** calculates the number of nodes in the entire tree.

```

// unsigned int _vertexX, unsigned int _vertexY
TerrainQuadTree::TerrainQuadTree(float _top, float _bottom, float _left, float _right,
    unsigned int _vertX, unsigned int _vertZ, unsigned int _level)
{
    cout<<"----->> Creating QuadTree" << endl;

    // calculate levels log4(1024) = 5
    // log(1024)/log(4) = 5
    int levels = log(pow(_bottom - _top, 2)) / log(4);
    cout<<">> Number of levels calculated from the size of the terrain: " << levels << endl;

    // calculate the number of nodes for memory allocation
    nodeSize = calculateNodeSize(_level);

    // calculate minimum node boundary size
    // this is for determining when a leaf node is found, for stopping further child node creation
    minSizeOfQuad = calculateMinQuadSize(_bottom - _top, _level);

    //leafNodesIndex = nodeSize - (unsigned int)pow(4, _level-1);
    //cout<<"***** LEAF NODES STARTING FROM INDEX: " << leafNodesIndex << endl << endl;

    // allocate memory for it
    qtNodeArray = (TERRAINQUADTREENODE*)malloc( sizeof(TERRAINQUADTREENODE) * nodeSize );
}

```

**minSizeOfQuad** is calculated next. This is the unit size of the node at the lowest level of the terrain, any node of this size is stated as a **QT\_LEAF**. Each **nodeSize** is the size of **struct TERRAINQUADTREENODE**. When **nodeSize** is available, heap memory can be allocated to store the entire tree. The first node is created next:

The image shows a code snippet for creating a quadtree root node, with several handwritten annotations in red and purple ink:

- Declare the first node**: A red circle highlights the line `TERRAINQUADTREENODE firstNode; // declare the first node`.
- the root node**: A red circle highlights the variable `firstNode`.
- Define the Boundary**: A red bracket groups the boundary definitions: `firstNode.top = _top;`, `firstNode.bottom = _bottom;`, `firstNode.left = _left;`, and `firstNode.right = _right;`.
- no parent**: A red bracket groups the parent ID assignments: `firstNode.ID = 0;` and `firstNode.parentID = 0;`.
- creates the tree with recursion**: A red arrow points to the line `createQuadTree(firstNode); // ***** create the quadtree structure (recursion)`.
- this function decreases the vertex index so that QTerrain.cpp's terrainData structure doesn't read over the boundary of the array.**: A purple arrow points to the line `adjustVertexIndex();`. Below this, a purple note says `terrainData[512][512]` and `[511][511] // actual read after adjustVertexIndex is called.`

```

TERRAINQUADTREENODE firstNode; // declare the first node

// input boundaries
firstNode.top = _top;
firstNode.bottom = _bottom;
firstNode.left = _left;
firstNode.right = _right;

// set IDs
firstNode.ID = 0;
firstNode.parentID = 0;

// input layer information
firstNode.layerID = 1; // first layer (this is used as 4^layerID to divide the landscape)
firstNode.vInitX = 0; // where to start the vertexIndex X
firstNode.vInitZ = 0; // where to start the vertexIndex Z

// how many vertices on x and z
vertX = _vertX; // 512
vertZ = _vertZ; // 512

createQuadTree(firstNode); // ***** create the quadtree structure (recursion)
adjustVertexIndex();

```

## Constructing the VerticeIndex

VerticeIndex are used to define the ‘internal’ points within each Quadtree node from which **QTTerrain.h** can use to construct **GL\_TRIANGLE\_STRIP**. There are  $3 \times 3$  vertex index per node. In the first layer, there will be  $3 \times 3$  vertex indices as there is only one node - the root node. In the second layer, there will be 4 nodes, each node having  $3 \times 3$  vertex indices and in total, the second layer would have  $6 \times 6$  vertex indices, one of the vertex index shares the same location. Figure 4 illustrates where the vertex indices are (red dots), and how QTTerrain.h code uses the vertex indices for constructing the surfaces of each quad.

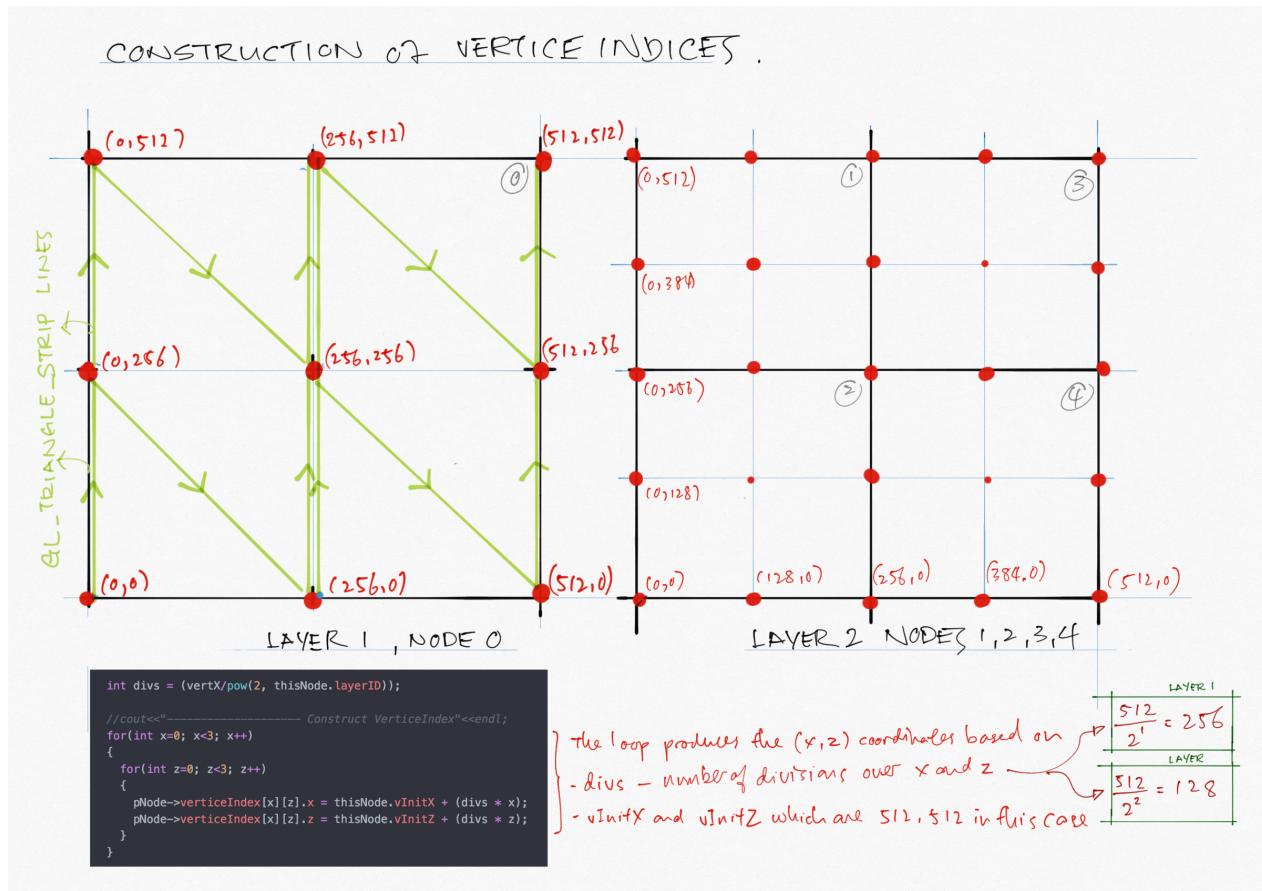


Figure 4. Construction of Vertice Indices. Red dots: vertice indices  $(x,z)$ , green directional lines: OpenGL **GL\_TRIANGLE\_STRIP** using vertice index for each quad, constructing two triangles as surfaces.

## QTerrain.h: Constructing OpenGL Terrain Surface

A suitable OpenGL surface is the GL\_TRIANGLES, which this QTTerrain class uses. There are three arrays used for constructing the surfaces. They are **heightField[x][z]**, **terrainNormals[x][z]**, and **terrainData[x][z]**. They can be consolidated as one, but it's easier to understand by separating them.

```

uint8_t heightField[512][512];          ← LOADS FROM RAW FILE
Vector3f terrainNormals[512][512];       ← CALCULATED LATER FOR LIGHTING,
vector<vector<Vector3f>> terrainData;   ← (STORING X, Y, Z) VERTEX FOR RENDERING TERRAIN

```

```

cout<<">> Initialising 2D <Vector> terrainData" << endl;
for(int x = 0; x <= width+1; x++)
{
    terrainData.push_back( vector<Vector3f>() );
    for(int z = 0; z <= length+1; z++)
    {
        terrainData[x].push_back(Vector3f(0.0f, 0.0f, 0.0f));
    }
}

```

The code shows the initialization of **terrainData** as a 2D vector of **Vector3f**. It first creates a row of empty vectors for each x coordinate, and then adds a single point (0,0,0) to each row. Red annotations explain: **heightField** loads from a raw file, **terrainNormals** is calculated later for lighting, and **terrainData** stores the vertex data for rendering. A curly brace on the right indicates the use of **vector<vector<...>>** initialization.

### Constructing **terrainData**: Loading RAW bitmap into **heightField[x][z]**

Once **terrainData** is initialised, the **heightField** is loaded, filling the data structure with the 8bit RAW bitmap data for each element in the array, in this case, 512 x 512

```

// open the terrain texture heightmap file
cout<<">> Opening heightField: "<<terrainFilename<< endl;
theFile = fopen(terrainFilename, "rb"); // rb means reading in data as binary (not writing out)

// this reads only RAW files
// read in the data from "theFile" into "heightField" from the start of the file "1"
// until "width*height" then stop
// fread(pointer to file, size per read (1 byte), size of file, the file);
fread(heightField, 1, width * length, theFile);
fclose(theFile); // close the file after reading

```

The next step is to populate the y element of the **terrainData** with the value of the **heightField**. The loop goes through x and z with **dWidth = dHeight = 512**. **terrainData[x][z]** constructs each point based on the x and z loop sequence {0, 1, 2, 3..., 512}. Each point is scaled with **terrainScale**, and adjusted to have the landscape at the origin with **adjFromOrig**. The y component of **terrainData** normalises the 8bit **heightField[x][z] / 255.0f** and scale them on the **scaleHeight** and **terrainScale**.

```
// loop through the x and z (vertices) with heightField points as y
// store each point in the terrainData 2D array.
void QTTerrain::generateTerrainPoints()
{
    cout<<">> Generating Terrain Points..."<<endl;

    for(int x=0; x<dWidth; x++)
    {
        for(int z=0; z<dHeight; z++)
        {
            // store vector of each point, build row x at col z first
            // adjustments to align the terrain origin to the world origin
            terrainData[x][z] = Vector3f( x*terrainScale - adjFromOrig,
                                         (heightField[x][z] / 255.0f) * scaleHeight * terrainScale, // y from height map
                                         z*terrainScale - adjFromOrig
                                         );
        }
    }
    cout<<">> Terrain Points Generated Successfully"<<endl;
}
```

## Calculating Cell Boundary

The cell boundary is then calculated. Each cell is made up of 4 vertices. The rationale for needing the cell boundary is so that we can determine which 3 vertices to use when calculating normals. Normals are needed for agents to move on each triangle.

```
// this calculates each cell's boundary (each cell is made up of 4 vertices)
// the rationale for this is so that we can determine which 3 vertices to calculate the normals
// for getting the height map of the terrain
void QTTerrain::calculateCellBoundary()
{
    cout<<">> Calculating cell boundary..."<<endl;

    for(int x=0; x < dWidth; x++)
    {
        for(int z=0; z<dHeight; z++)
        {
            cellinfo[x][z].top = z * terrainScale - adjFromOrig;           // top
            cellinfo[x][z].bottom = (z+1) * terrainScale - adjFromOrig; // bottom
            cellinfo[x][z].left = x * terrainScale - adjFromOrig;          // left
            cellinfo[x][z].right = (x+1) * terrainScale - adjFromOrig; // right
        }
    }
}
```

## Calculating Normals

There are two functions for calculating normals: `calculateNormals(...)` and `calculateFaceNormals(...)`. The first calculates all normals in `terrainData` and stores them within `terrainNormals[x][z]`, which is used for OpenGL lighting during rendering. The parameter flag is used for rendering either as OpenGL flat shading or as smooth shading using interpolation.

```
void QTTerrain::calculateNormals(int flag)
```

The second function calculates the normal vector of a triangle, which can be used for allowing agents to ‘skate’ on the surfaces of the terrain.

```
Vector3f QTTerrain::calculateFaceNormal(Vector3f p0, Vector3f p1, Vector3f p2)
```

`calculateFaceNormals(...)` calculates each face using the 3 points of a triangle. Normals are calculated by doing a cross product of vA and vB, where vA = p1-p0 and vB = p2-p0. vA x vB produces a normal vector.

```
Vector3f QTTerrain::calculateFaceNormal(Vector3f p0, Vector3f p1, Vector3f p2)
{
    // -----
    // calculate normals

    // get vectors between points
    Vector3f vA = p1-p0;
    Vector3f vB = p2-p0;

    // cross product of vectors
    Vector3f vN = vA.crossProduct(vB);
    vN.normalise();

    //cout<<"----->> Plane Normal Vector: ";
    //vN.print();

    return vN;
}
```

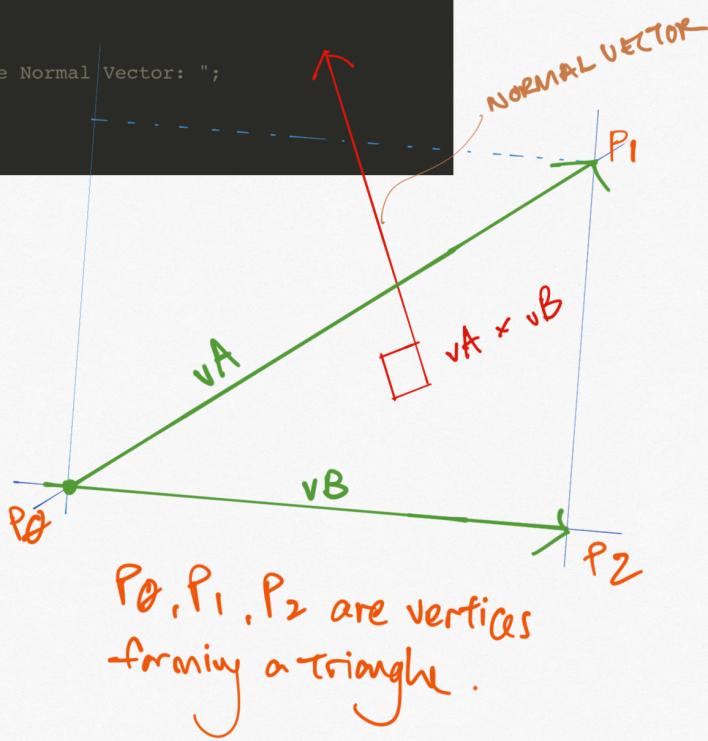


Figure 5. Calculating face normals

## Rendering the Terrain

The render() function loops through ALL quadtree terrain nodes `for(int i=0; i<nodeSize; i++)`, checking for the visibility of the nodes based on camera position and its view range, and render nodes which are visible.

```
// run through all the nodes and test if each node, based on the camera position
// is visible, if it is, render it.
for(int i=0; i<terrainQT->nodeSize; i++)
{
    //cout<<"node [ "<<i<<"] "<<terrainQT->qtNodeArray[i].visible<<endl;

    // if the node is visible, draw it's vertices
    if (terrainQT->qtNodeArray[i].visible == true)
    {
```

The nodes which are visible are rendered based on the x, y and z component of the `terrainData[x][z]`, using the `GL_TRIANGLE_STRIP` to construct the quads (2 triangles) using `vertexIndex` from `TerrainQuadTree.h`.

The second loop draws (or connects) the four points of each quad, making them a surface. **terrainNormals** are input for OpenGL lighting calculations. For each loop in z, the code takes the **verticeIndex** coordinates and give it **terrainNormals** and **terrainData** for OpenGL for rendering (Figure 6). The loop traverses each row [x], drawing all columns [z] first, before going to the next row, from **quadSize=2**, with x={0...2} to z={0...2}. The **verticeIndex** retrieved for a 512x512 landscape, for a quadtree with 2 layers, for the level=2 will be vX={0,127,255,383,511} and vZ={0,127,255,383,511}.

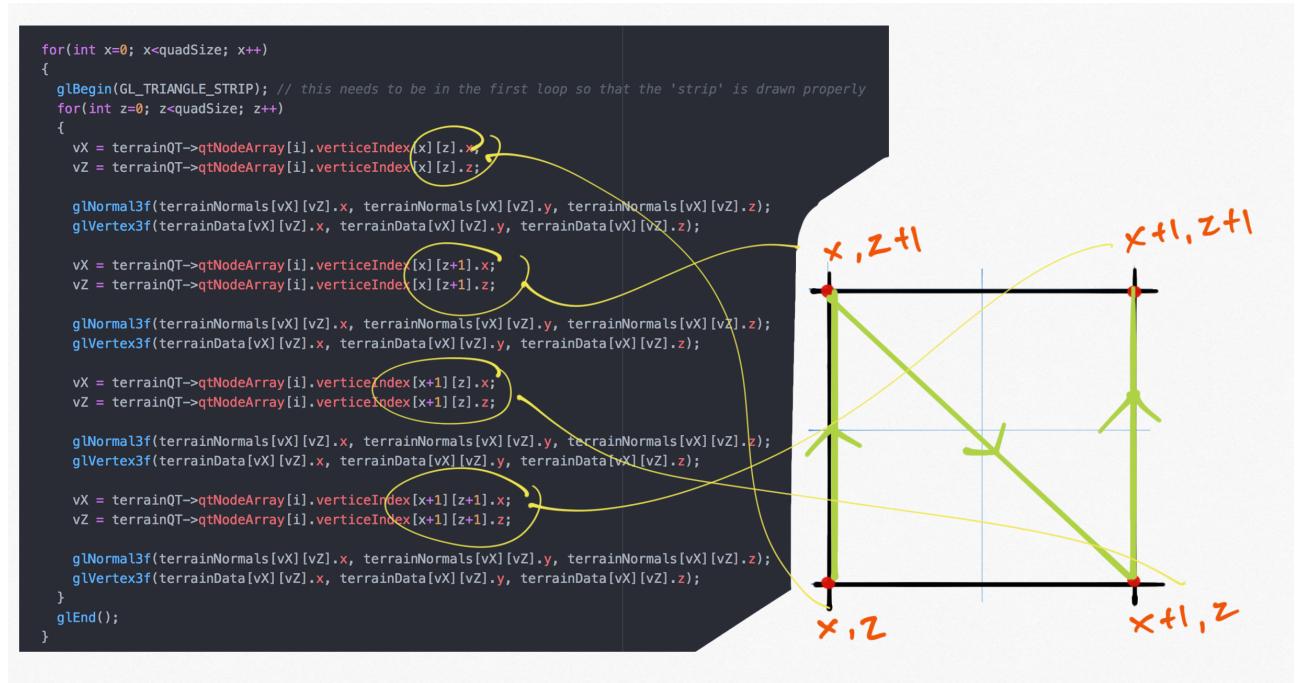
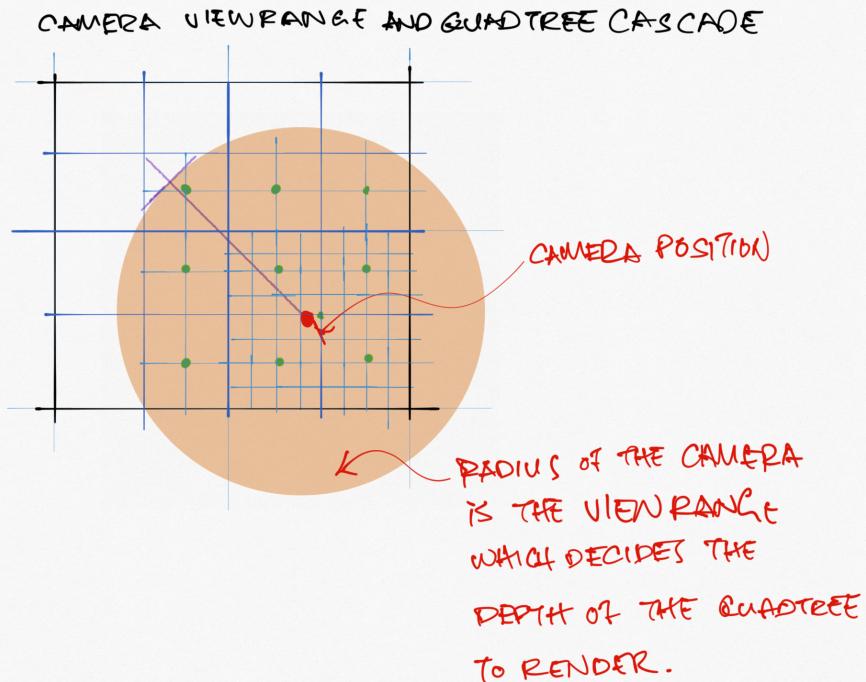


Figure 6. Each z loop constructs GL\_TRIANGLE\_STRIP using the four points in terrainData[x][z].

## Camera View Range

The view range of the camera and its position decides which nodes to draw, and how deep the cascade of the nodes are. The nodes nearest to the camera are cascaded down to the QT\_LEAF nodes, drawing denser geometry. The function `terrainQT->resetNodeVisibility` resets all nodes to invisible, the function `terrainQT->testRenderable(...)` tests the view range and switches on nodes within the view range.



```
// update quadtree LOD based on Camera position
terrainQT->resetNodeVisibility();
terrainQT->testRenderable(terrainQT->qtNodeArray[0],
    Vector3f(cameraPos.x, cameraPos.y, cameraPos.z), viewRange);
```

Figure 7. Camera view range and quadtree cascade of nodes.

## QUADTREE FOR AGENT INTERACTION