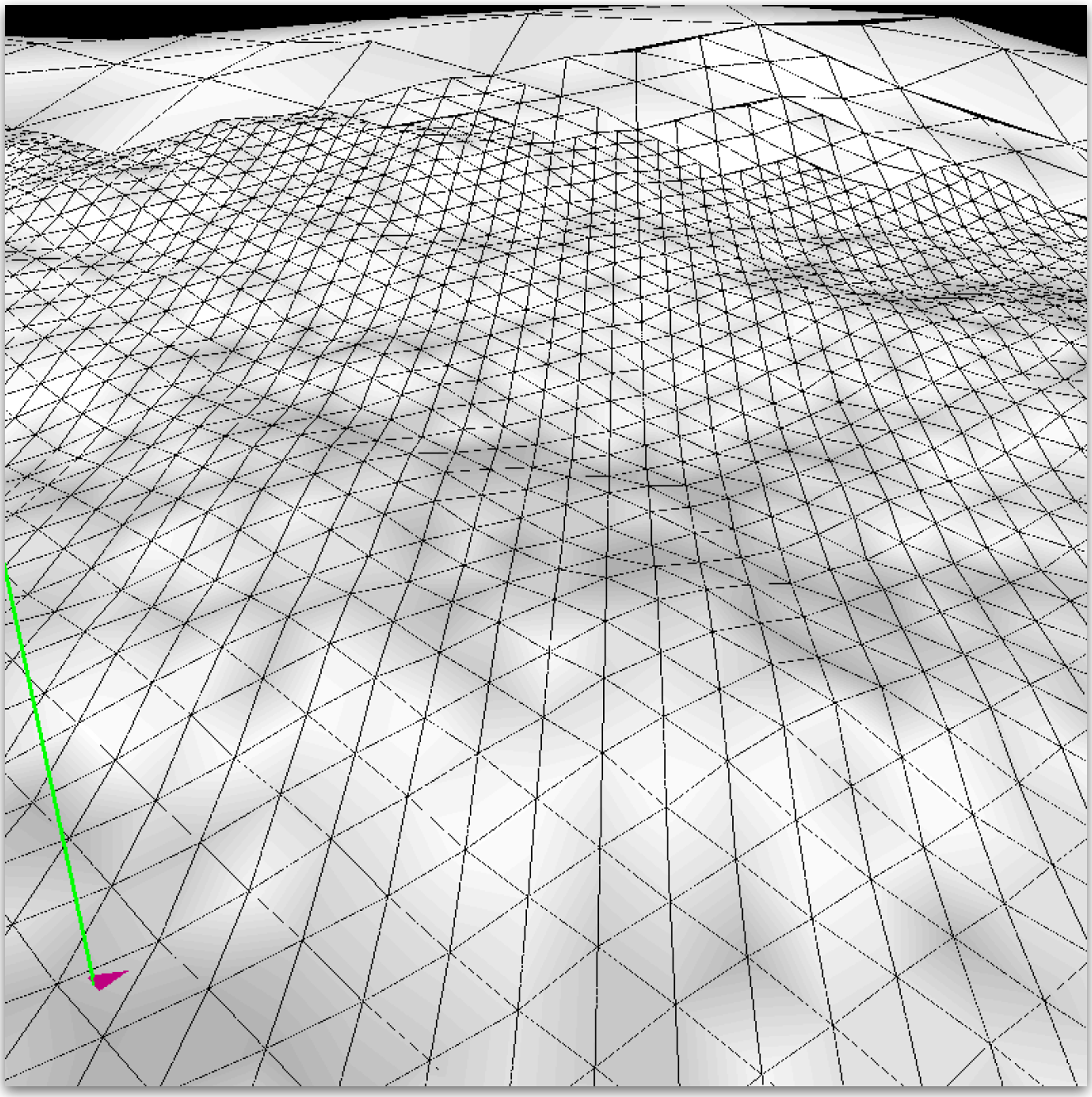
QUADTREE PROCEDURAL TERRAIN FOR AGENT-BASED SIMULATION



Quadtree LOD Procedural Terrain

Development Documentation

Prepared by: Eugene Ch'ng (www.complexity.io) | 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 build 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 interacts 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 x 3 vertices which ***QTTerrain.h*** uses to draw ***GL_TRIANGLE_STRIP***.


```
struct VERTICEINDEXINFO { int x, z; };
    // vertice index mapping node vertices to indices in terrainData[x][z]
enum NODETYPE {QT_NODE, QT_LEAF};  // for determining whether the node is leaf

struct TERRAINQUADTREENODE
{
    int nodeType;
    // node or leaf enum
    int ID;
        // my unique ID
    int parentID;
    // for identifying parent
    float top, bottom, left, right; // boundary
    float width, height;            // width and height of the node (bounding box size)
    Vector3f position;
    // a position for comparing distance between this node with camera pos
    bool visible;
    // is this node visible for drawing?


    //float terrainWidth;        // a permanent width for calculating

    // the nodes has 4 quadrants (branches), their array index is stored in this variable
    unsigned int branchIndex[4];

    // a node has 9 vertices, 3 on each sides 3x3=9
    // this amounts to 4 cells
    VERTICEINDEXINFO verticeIndex[3][3];

    // the division level: width/divs = LevelDivs
    // first level is 32/2^1 = 16, 16 is the division related to the index to terrainData[x][y]
    int layerID;

    // the variables tells the next node on where to start the verticeIndex from
    int vInitX, vInitZ;


};
```

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 x 3 vertice 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.

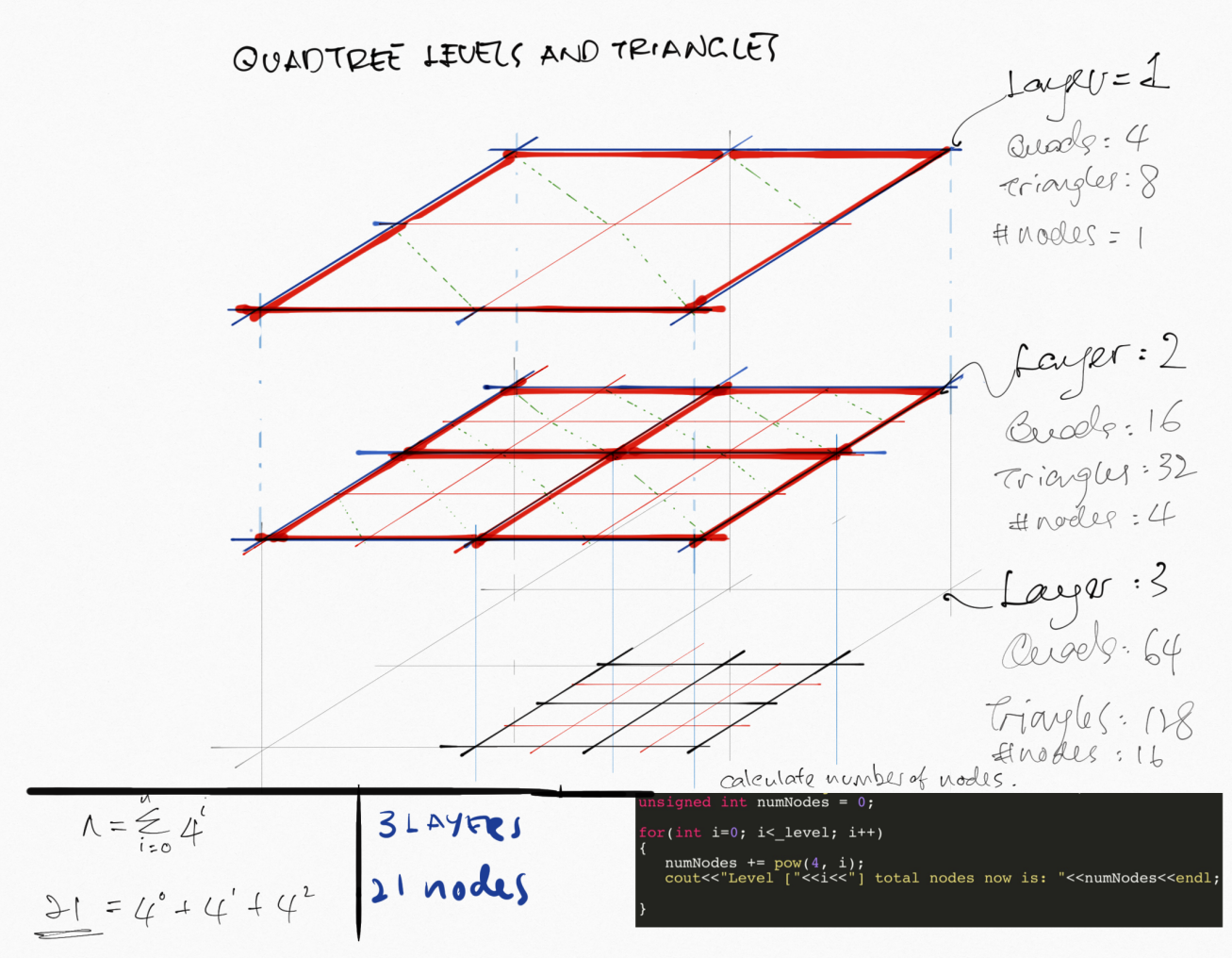


Figure 2. Quadtree levels and triangles

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.

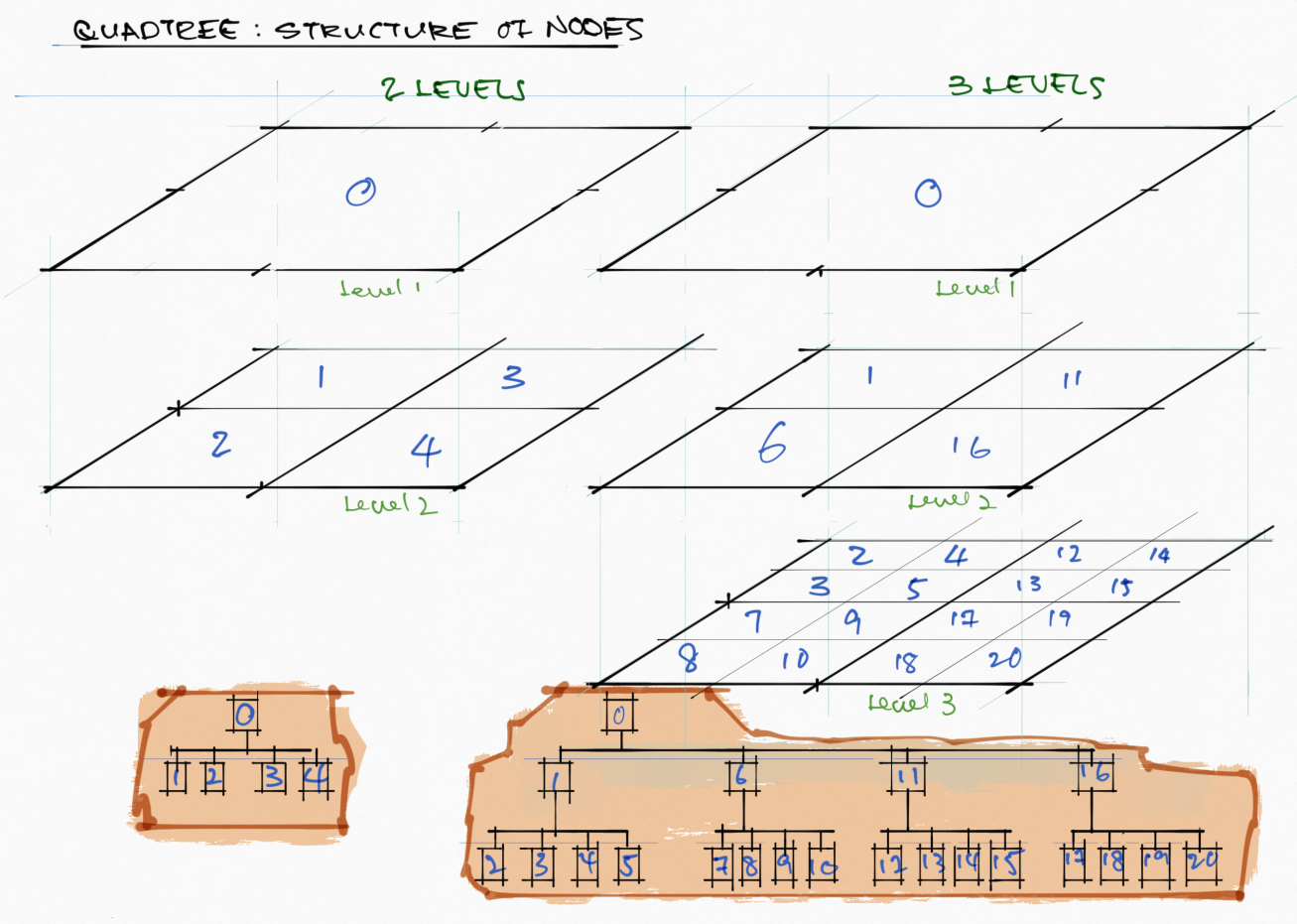


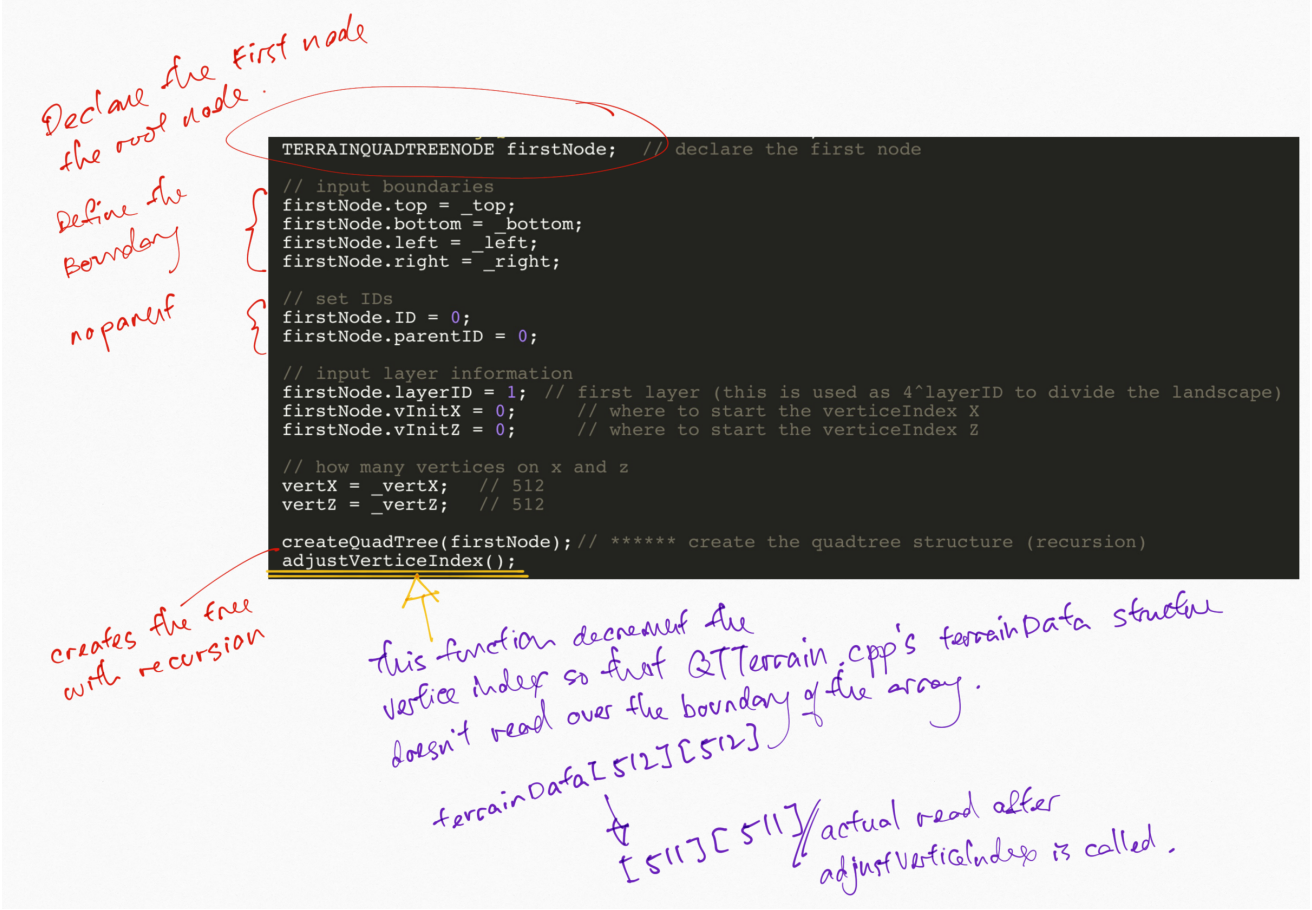
Figure 3. Quadtree structure and how node IDs are cascaded.

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:




```
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 verticeIndex X
firstNode.vInitZ = 0;      // where to start the verticeIndex Z

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

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

Constructing the VerticeIndex

Vertice indice 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 x 3 vertice index per node. In the first layer, there will be 3 x 3 vertice indices as there is only one node - the root node. In the second layer, there will be 4 nodes, each node having 3x3 vertice indices and in total, the second layer would have 6x6 vertice indices, one of the vertex index shares the same location. Figure 4 illustrates where the vertice indices are (red dots), and how ***QTTerrain.h*** code uses the vertice indices for constructing the surfaces of each quad.

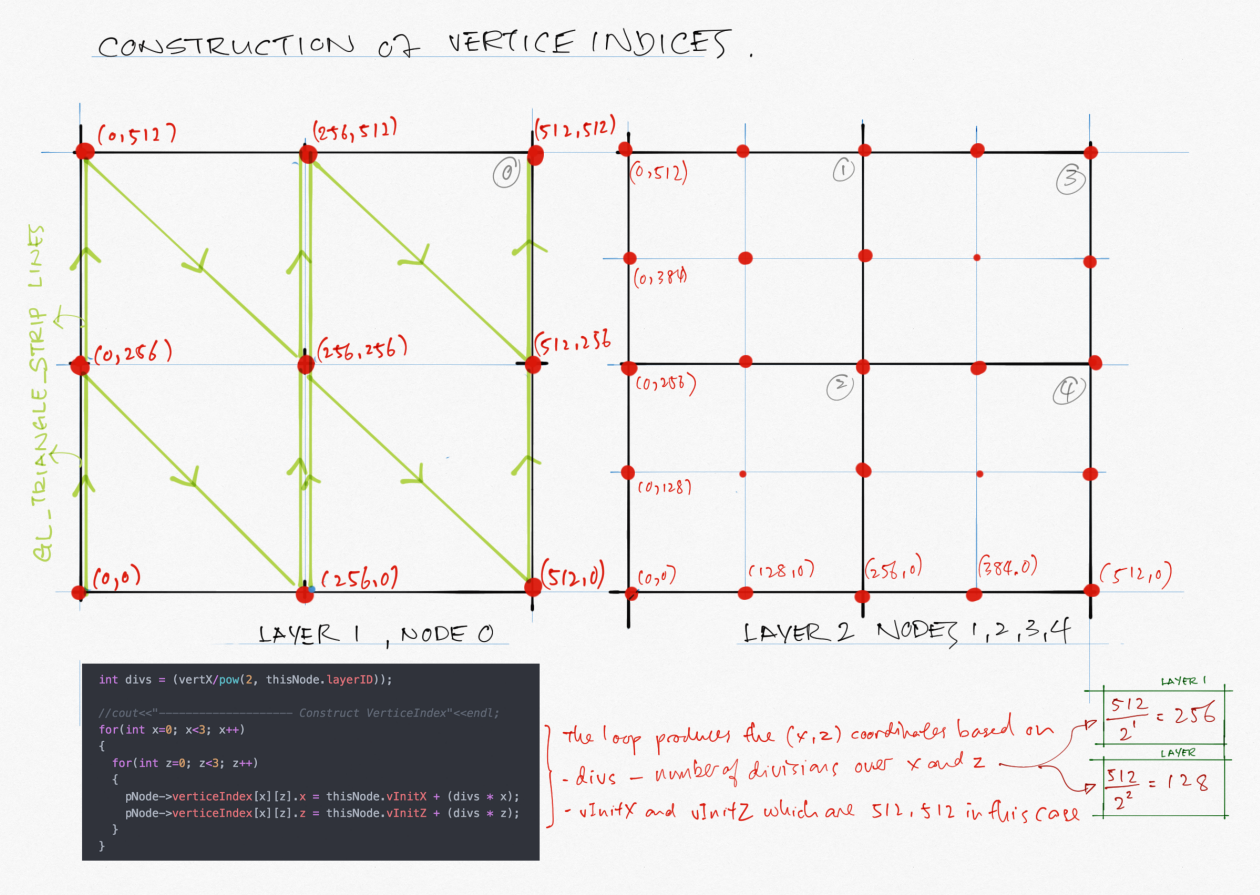
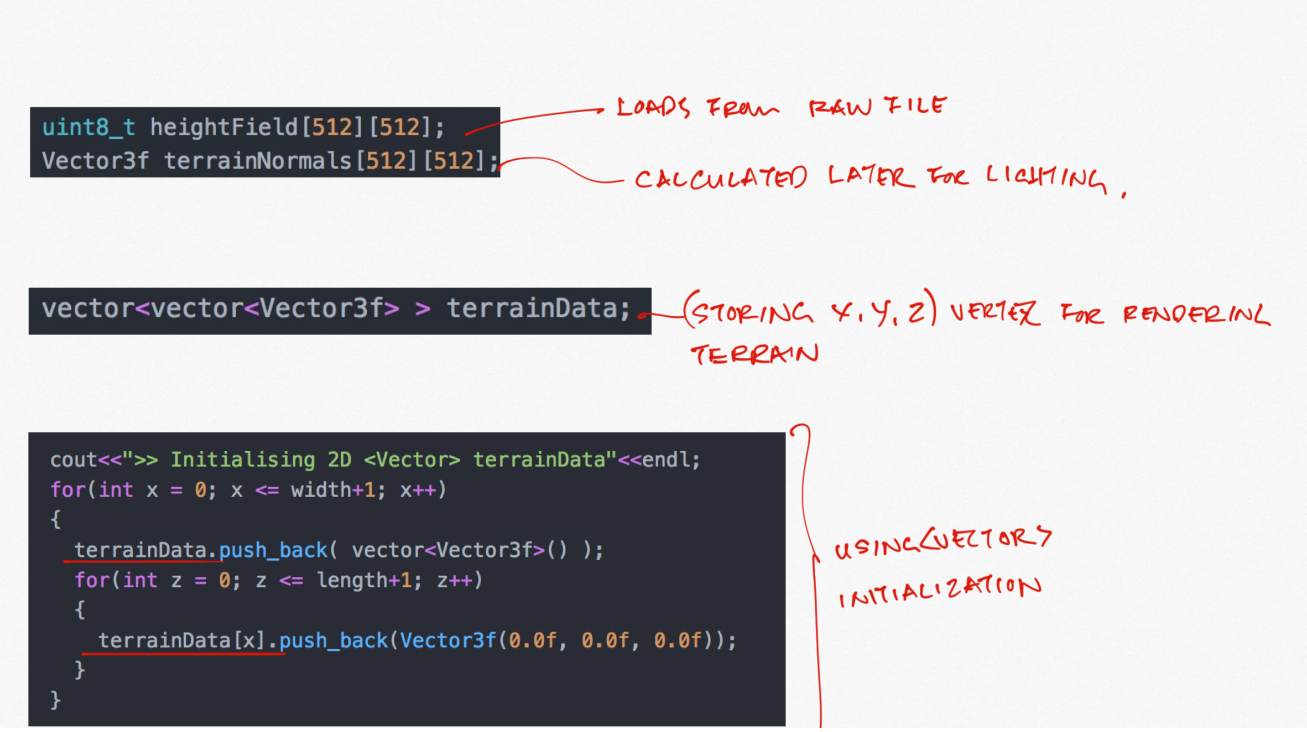


Figure 4. Construction of Vertice Indices. Red dots: vertice indices (x,z), green directional lines: OpenGL ***GL_TRIANGLE_STRIP*** using vertice index for each quad, constructing two triangles as surfaces.

QTTerrain.h: Constructing OpenGL Terrain Surface

A suitable OpenGL surface is the ***GL_TRIANGLE_STRIP***, 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.



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 Teh 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.

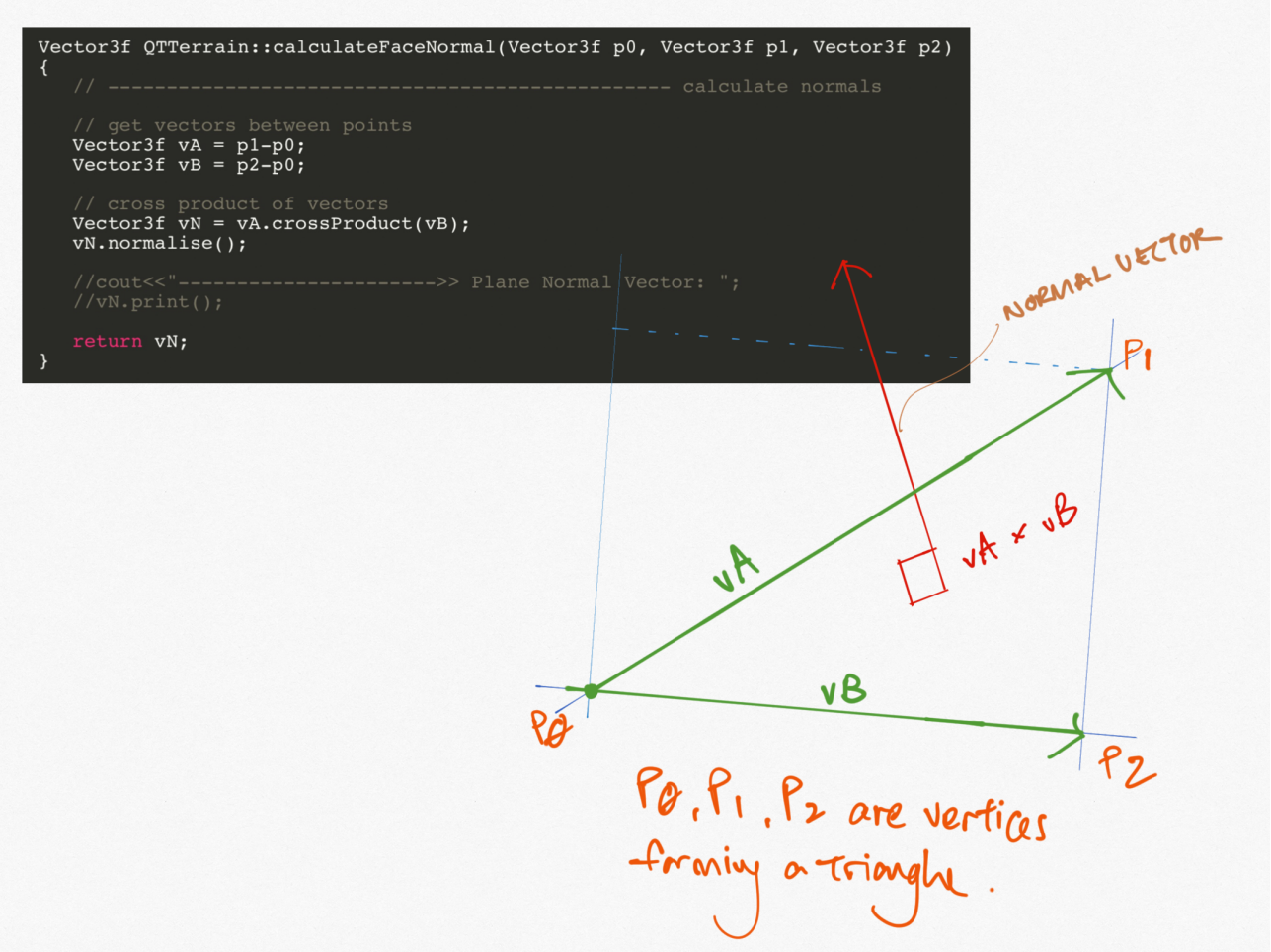


Figure 5. Calculating face normals

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 ***verticeIndex*** 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}.

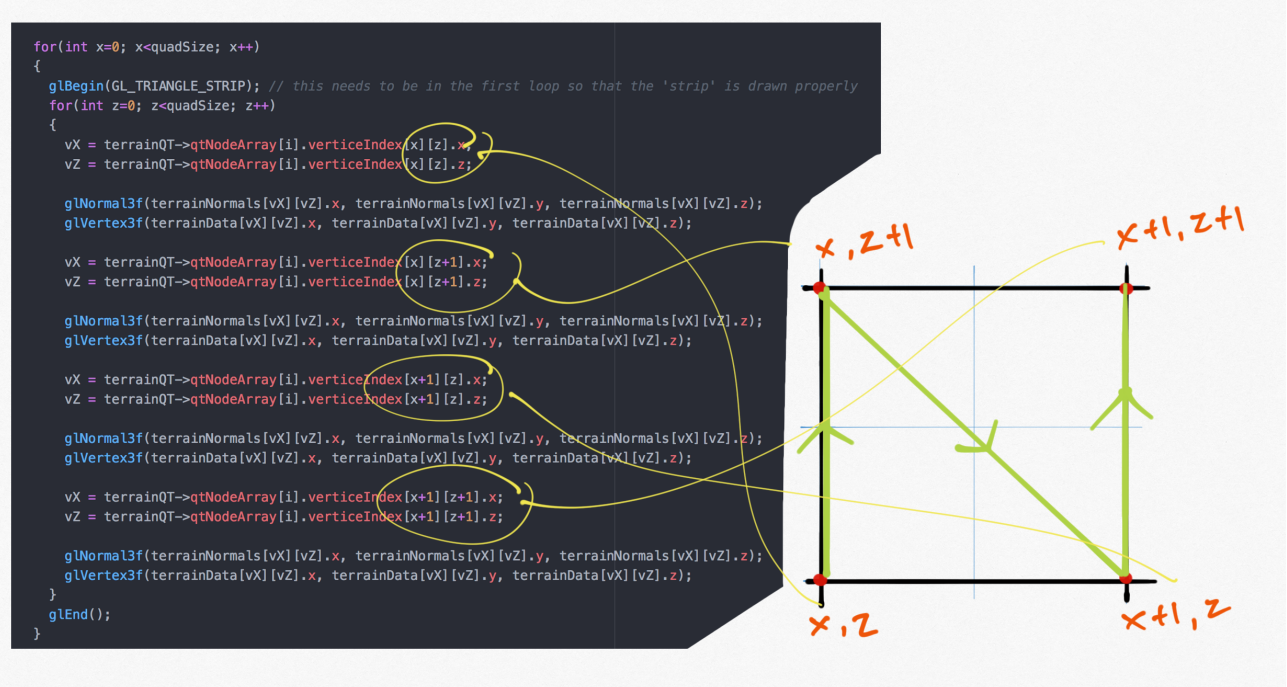
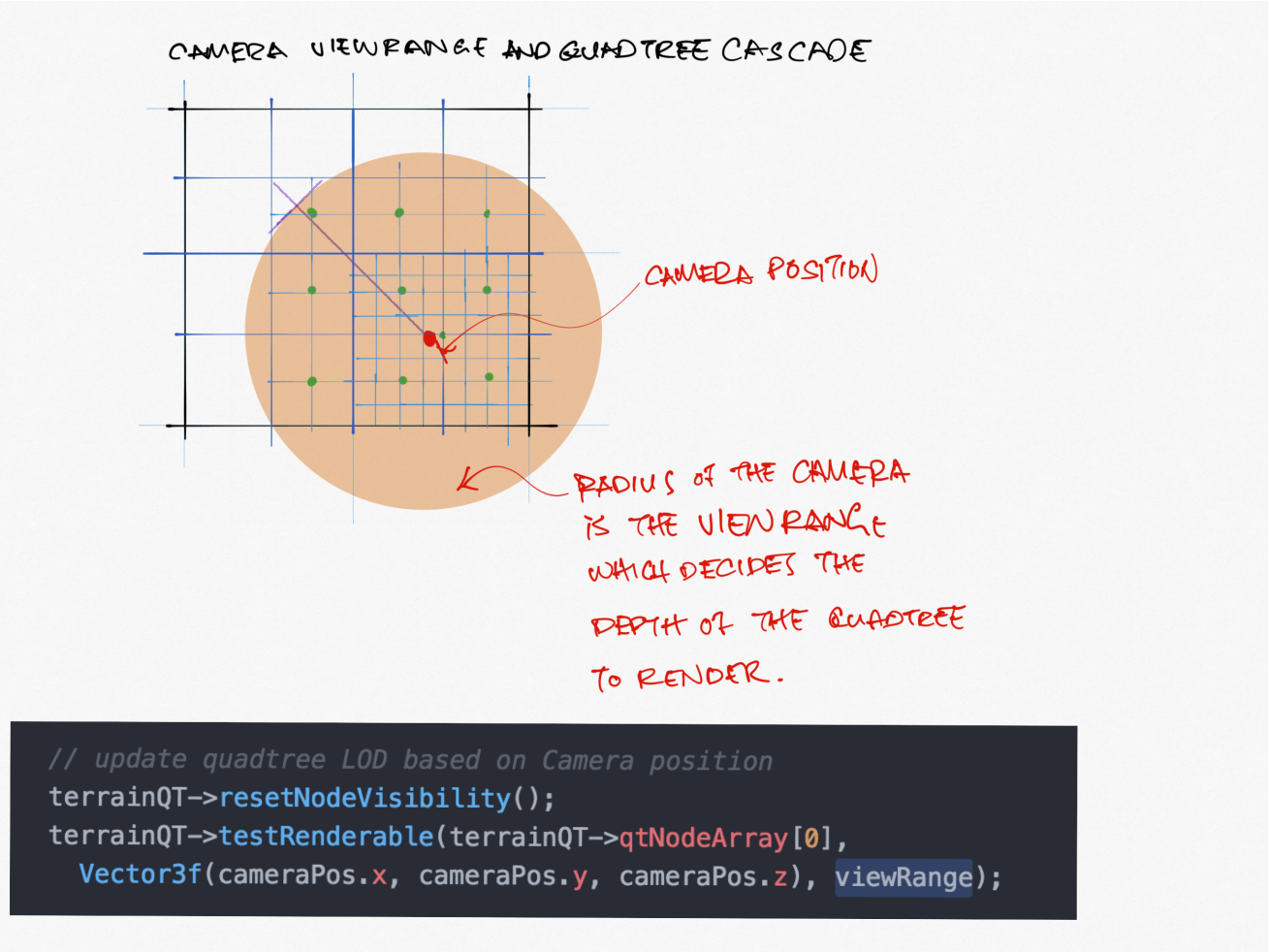


Figure 6. Each z loop constructs GL_TRIANGLE_STRIP using the four points in terrainData[x][z].

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

A Quadtree data structure build for managing agents on a terrain will need additional variables (arrays) for storing agents and environment emitters within each node, and therefore different from a Quadtree-based terrain system. Additional utility functions are also necessary for managing agents. Agents and environment emitters in a Quadtree terrain system cascades down into depth of the nodes based on certain programmatic rules, e.g., divide the node into more nodes if there are more than *n* number of agents, etc. The division and cascading of nodes will have different node IDs as the the Quadtree is dynamic, and is divided on demand.

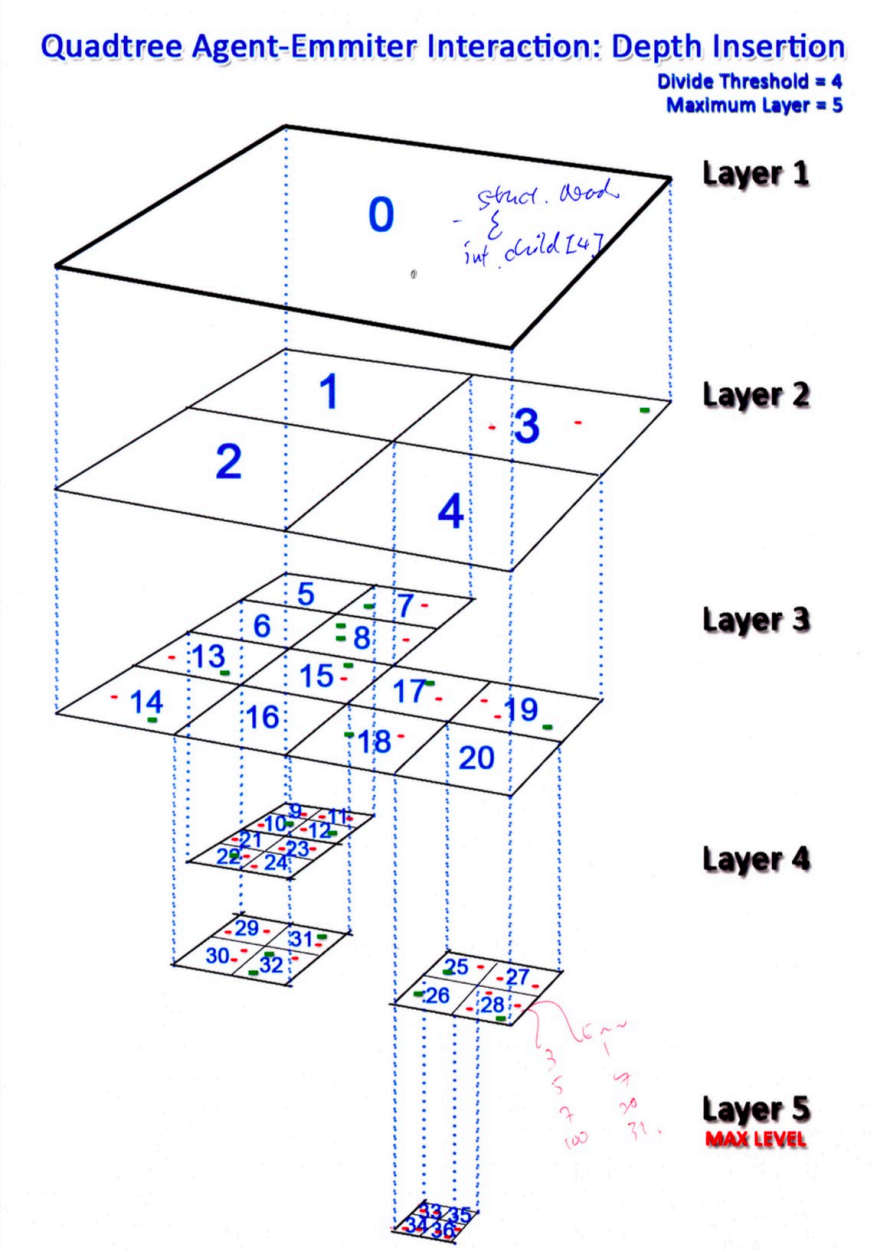


Figure 8. Agents and environment emitters cascading down into the depth of the tree.

Agent Quadtree Struct

The management of Quadtree nodes uses C++ vector:

vector<AGENTQUADTREENODE> qtNodes;

Each Quadtree node is a struct, and vectors are used for transferring agents and emitters between nodes as they move around.

```
struct AGENTQUADTREENODE
{
  int nodeType;                    // node or leaf enum
  int ID;                          // my unique ID
  int parentID;                    // for identifying parent
  float top, bottom, left, right; // boundary
  float width, height;             // width and height of the node (bounding box size)
  Vector3f position;               // a position for comparing distance between this node with camera pos
  bool visible;                    // is this node visible for drawing?

  // the division level: width/divs = LevelDivs
  // first level is 32/2^1 = 16, 16 is the division related to the index to terrainData[x][y]
  int layerID;

  // the nodes has 4 quadrants (branches), their array index is stored in this variable
  int childIndex[4];

  // For polymorphism
  // <Object*> stores pointers to derived class from globals: gAgents and gEmitters
  vector<Object*> agents;     // all agents are stored here
  vector<Object*> emitters;   // all local environment emitters are stored here
};
```

vector<Object*>agents and ***vector<Object*>emitters*** uses the updated base class Object as pointers because of the need for polymorphism with the virtual function update(). The classes Agent, Predator, Prey, Snack have all been updated to work with polymorphism. The hierarchy of the base and child classes is this (Figure 9):

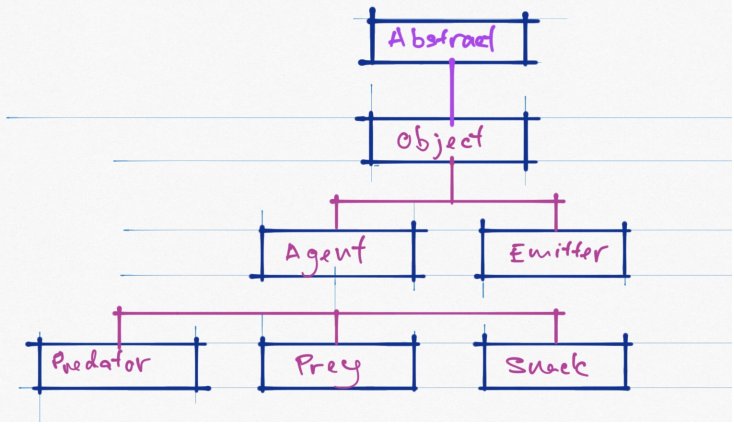


Figure 9. Object hierarchy within the agent-interaction Quadtree. Abstract class can't be instantiated and thus Object is the base class.

Agent Quadtree Utility Functions

Two code files define the Agent Interaction Quadtree structure: ***AgentQuadTree.cpp*** and **AgentQuadTree.h** of which additional utility functions are listed below:

- ***bool divideNode(AGENTQUADTREENODE *thisNode);***
- ***int placeObjectInNode(AGENTQUADTREENODE &parentNode, Object *obj);***
- ***int inWhichNodeBoundary(AGENTQUADTREENODE parentNode, Vector3f pos);***

Agent Quadtree Constructor

The constructor for ***AgentQuadTree*** is shown below. It takes the terrain width and height in QTTerrain and multiplies it by terrain_Scale for the size of the landscape. maxLevels set the maximum depth-level the Quadtree maybe divided into. The divideThreshold decides the division of a node into four more nodes (quads) based on the number of objects currently within the nodes ***(agents + emitters)***. If ***divideThreshold = 5***, then when ***(agents + emitters)>=5***, the node is divided.

```
AgentQuadTree(float terrainWidthHeight, float terrain_Scale, int maxLevels, int divideThreshold);
~AgentQuadTree();
```

The constructor calculates the ***nodeSize*** based on ***maxLevels***. ***adjFromOrig*** is calculated for setting the boundary of the terrain (top, bottom, left, right). Finally, ***minSizeOfQuad*** is calculated to determine when a node should be ***QT_LEAF***.

```
_divideThreshold = divideThreshold;
_maxLevels = maxLevels;

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

// adjustment for setting terrain centre at origin
float adjFromOrig = (terrain_Scale*terrainWidthHeight)/2;

// set terrain boundary
float _top = -adjFromOrig;
float _bottom = adjFromOrig;
float _left = -adjFromOrig;
float _right = adjFromOrig;

// 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, _maxLevels);
```

```
AGENTQUADTREENODE firstNode;

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

// getting the width and height up to the QTTerrain scale
firstNode.width = terrain_Scale*terrainWidthHeight;
firstNode.height = terrain_Scale*terrainWidthHeight;

// set node related IDs
firstNode.ID = 0;
firstNode.parentID = 0;
firstNode.layerID = 1;        // first layer (used as 4^layerID to divide the landscape)
firstNode.nodeType = QT_LEAF; // all nodes are initially a leaf

// the root node is initially a leaf node, so all child is -1
firstNode.childIndex[0] = -1;
firstNode.childIndex[1] = -1;
firstNode.childIndex[2] = -1;
firstNode.childIndex[3] = -1;

// add the first node to the array
qtNodes.push_back(firstNode);
// first divide the rootnode
divideNode(&(qtNodes[0]));
```

Next, also within the constructor, the very first node is created with all the information. A Quadtree is defined by four nodes so the root node is divided further at the end of the constructor.

Dept Insertion: Cascading Objects into Nodes

Once the root node has been divided into 4 other child nodes, the Quadtree is ready for agents to be inserted into the landscape with the ***placeObjectInNode(AGENTQUADTREENODE &parentNode, Object *obj)***. Cascading occurs at the root node and thus the parent node ***qtNodes[0]*** is provided here. *obj is the pointer to the agent or emitter to be inserted into the Quadtree. The three lines of code below work together to achieve the cascade of objects into the nodes.

```
int placeObjectInNode(AGENTQUADTREENODE &parentNode, Object *obj);
int inWhichNodeBoundary(AGENTQUADTREENODE parentNode, Vector3f pos);
bool divideNode(AGENTQUADTREENODE *thisNode);
```

inWhichNodeBoundary(...) checks the object's position (x,z) to see where the object is placed, within which node. ***divideNode(...)*** divides only the ***QT_LEAF*** node into 4 child nodes. The logic for ***placeObjectInNode(...)*** is below:

placeObjectInNode(AGENTQUADTREENODE &parentNode, Object *obj)

- ***IF nodeType is not QT_LEAF***
  - ***Check the position to see which childNode the obj is in, do placeObjectInNode in that childNode***
- ***IF nodeType is QT_LEAF***
  - ***Get combined number of agents+emitters***
  - ***IF over divideThreshold***
    - ***IF this is Quadtree level is at the maxLevel, put obj within this node***
    - ***IF this is not at maxLevel (and a QT_LEAF), divide this node into four child nodes***

- *Transfer this object and all objects in the parentNode into the child node based on inWhichNodeBoundary()*
- *Remove all agents and emitters from the parent node*
- *Set parentNode.visible to false*

- *IF not over divideThreshold*
  - *Place the object in this node*

A test of placeObjectInNode(...) with a maxLevel=5 and divideThreshold=4 below shows how the agents cascaded and divided the nodes into quads where node[24] and node[9] contains the two sets of agents from node[1] and node [4].

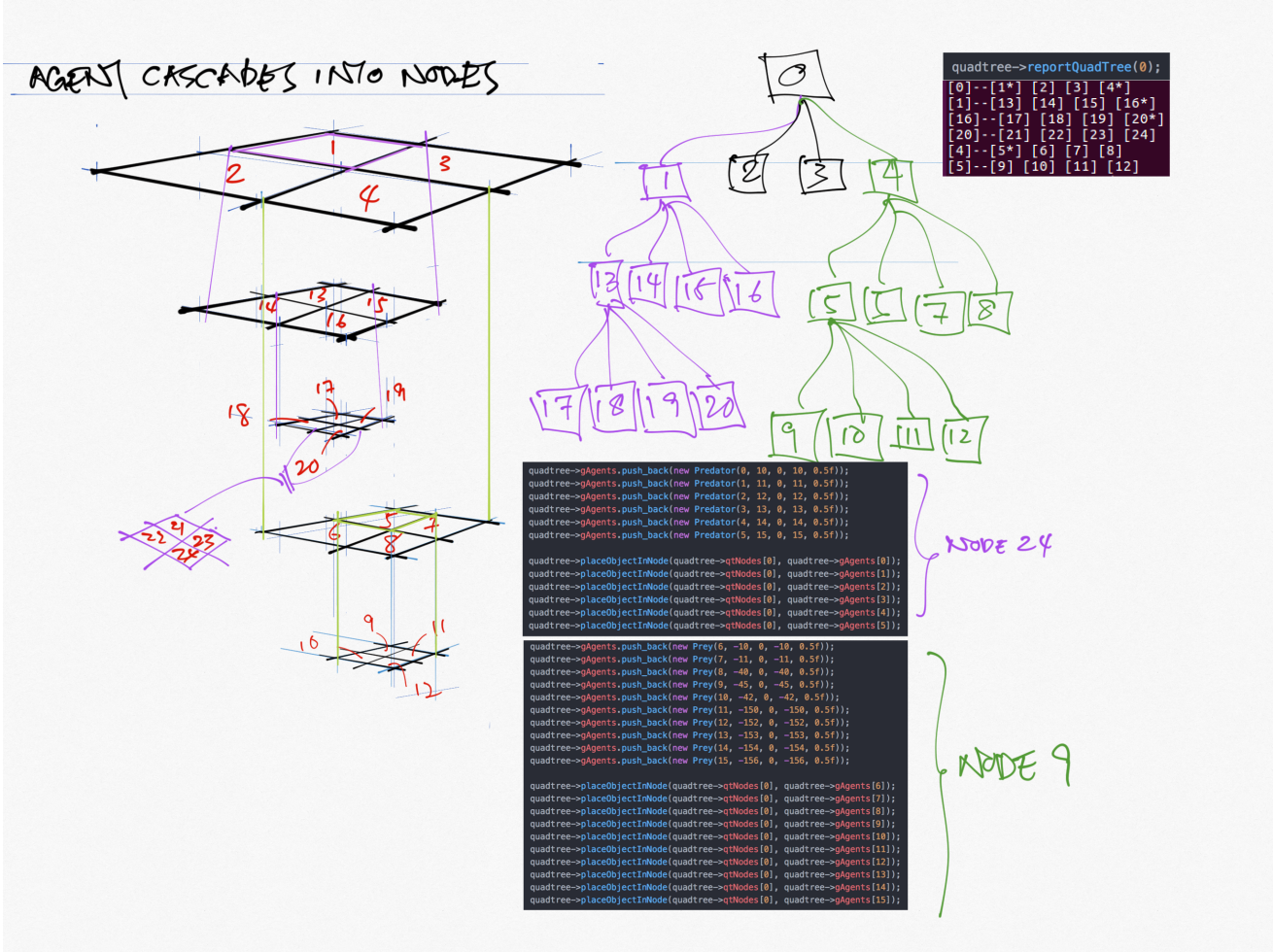


Figure 10. Agents put into the quadtree using placeObjectInNode(...) cascading down into the deepest level at maxLevel=5.

Cascading Example

Predator={0...5} gets down to node[9], and Prey={6...15} gets down to node[24]. The sequence for node[24] is as follows. Predator={0...3} is put into node 0 with the position={x,y,z} close to the axis at node[1], which cascades down through node[1] -> node[16] -> node[20] -> node[24] the moment Predator={4} is inserted into the landscape as the divideThreshold is breached. Predator={5} when inserted basically just cascades down to the deepest level at node[24].

Range Query

Range query traverses the breadth and depth of nodes collecting agents in the view range measured by the radius ***r***. If any of the centre node axis are within the ***viewRange***, the agents or emitters or both are collected in the vector ***rangeObjs*** for interacting with. ***rangeObjs*** takes three parameters, the object (an agent), the types of Objects to collect {agent, emitters, all} and the radius.

rangeQuery(Object *obj, OBJECTYPE objectType, float radius)

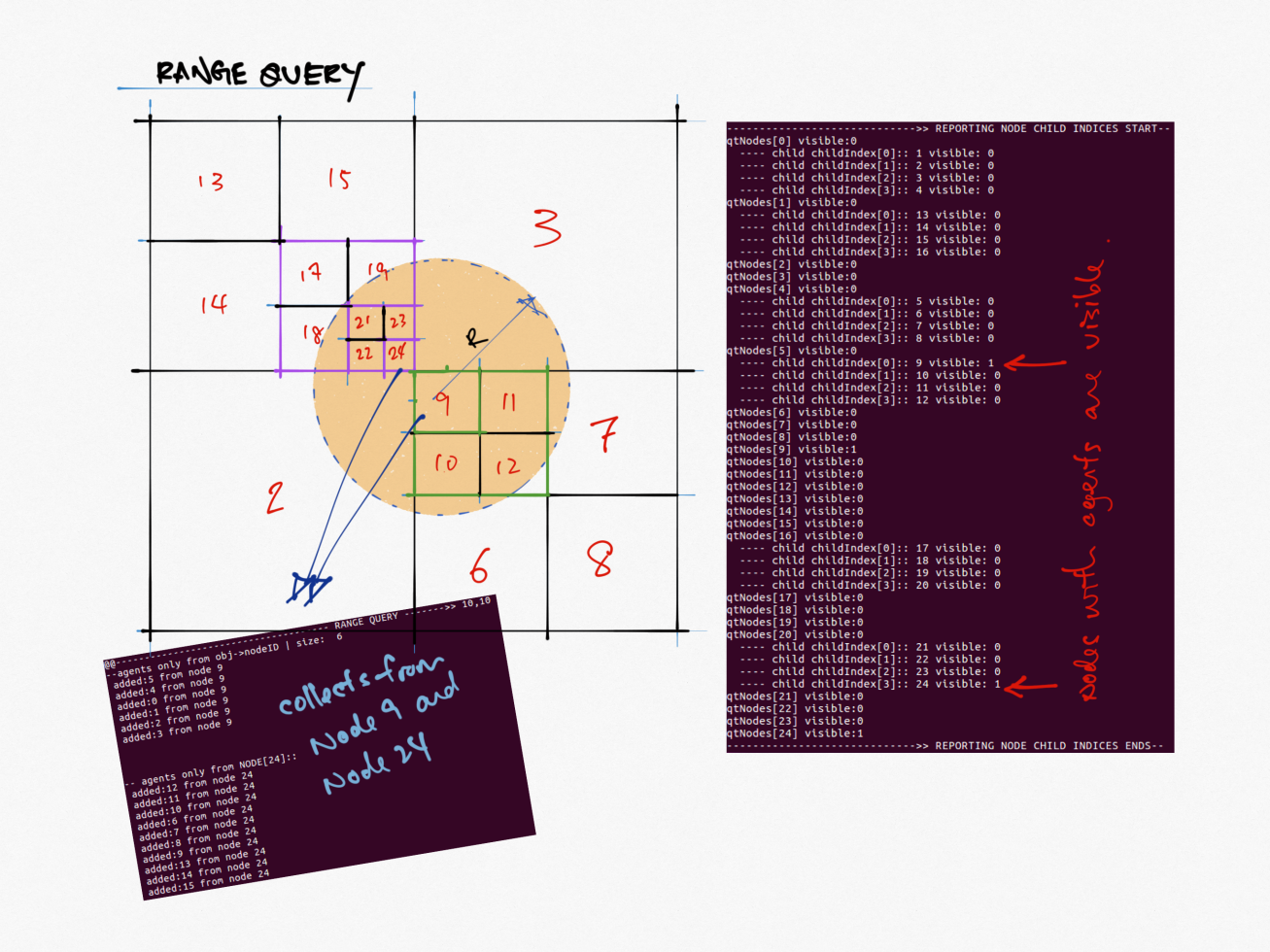


Figure 11. Range Query algorithm for the AgentQuadTree class.

Note that the radius could not reach the centre axis of quads which aren't divided. This can be resolved by initially dividing second level nodes {1, 2, 3, 4} into quads, prior to inserting agents into the landscape, such as in Figure 12. However, note that the undivided nodes would have 0 > i >= 4 agents, perhaps an insignificant number of agents, unless the agents are important to the simulation.

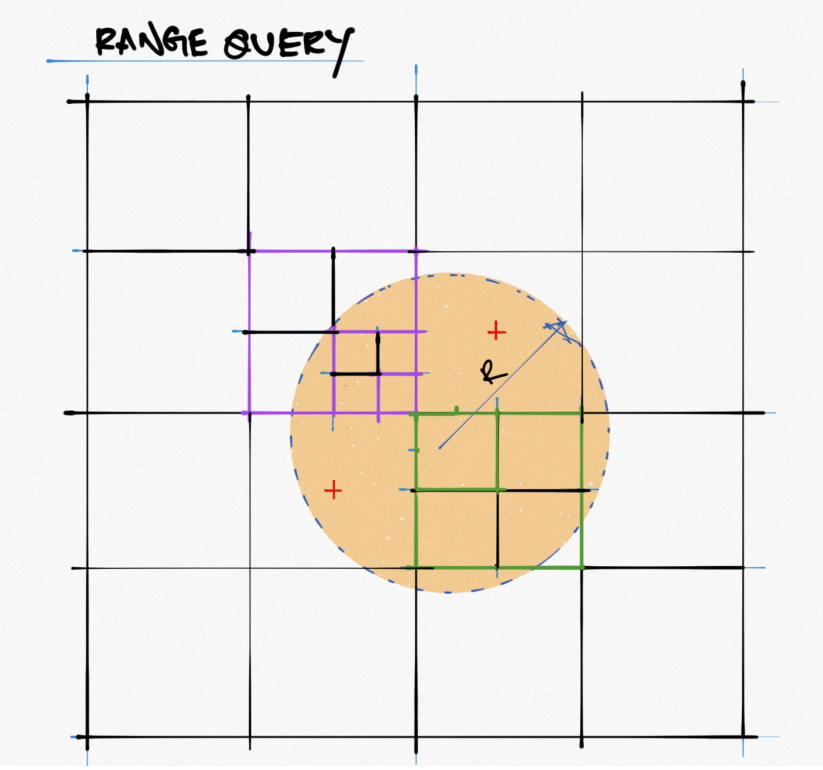


Figure 12. Initially dividing second level nodes {1, 2, 3, 4} into quads, prior to inserting agents into the landscape solved the issue of the ***viewRange*** not hitting the larger nodes' centre axis (red crosses).