VoxFox : A Voxel Library for C++ using NGL

Tom Collingwood*
National Centre for Computer Animation

Abstract

For this assignment I have created a data structure representation for voxels. It uses a sparse oct-tree that has a tree height of three. VoxFoxTree (root node) can hold 8^3 PrimaryNodes, PrimaryNode can hold 8^3 SecondaryNodes, SecondaryNode can hold 8^3 LeafNodes. Leaf nodes always hold the data for 8^3 Voxels. The tree is sparse which means that each node can store 1 to 8^3 nodes.

In terms of class diagrams see the diagram I submitted for initial design except combine RootNode and World class.

I'm going to go through the parts of the program in order of which their called. Explaining the tree structure, then importing meshes, applying set operations and finally rendering them.

1 Tree

1.1 Sparseness

Storing every single voxel data even if empty in a worldspace would be wasteful. So the sparse nature of the octtree stops this from happening. If a space where a PrimaryNode would be is empty then VoxTreeNode simply does not have this PrimaryNode. The same for SecondaryNodes within PrimaryNodes and so on. Even to the voxel level.

1.2 Leaf and Voxel

One difference between LeafNode and the other nodes is that it has an array of boolean values telling us which voxels are present and has data for. The other nodes do not have such a map telling us which are present. This is because we need isVoxel function for the drawing. This quickly checks the `m_VoxelMap` without accessing the data. No need to iterate through the data.

The Voxels that are preset have their own `Voxel` object inside `m_VoxelData`. The Voxel data structure (a struct) has an index which tells us which Voxel data it holds. They are also in order of index to allow for easy iterating when drawing.

1.3 `AddVoxel(..)` and `Accessors(..)`

When you call `addVoxel` from `VoxFoxTree` it first checks the accessors for the relevant LeafNode, then SecondaryNode then PrimaryNode. The accessors hold the last accessed LeafNode, SecondaryNode and PrimaryNode. If none of the accessors have the right bounding box for the voxel to be inserted it will search all PrimaryNodes for the right PrimaryNode. Once the right PrimaryNode is found you call this PrimaryNode's addVoxel function and so on.

Using `m_VoxelMap` and `m_VoxelData` together saves space in memory.

*e-mail:i7449874@bournemouth.ac.uk

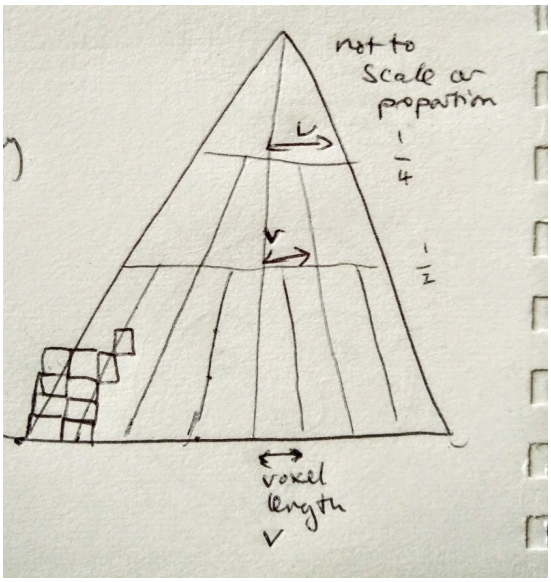

Figure 1: *Diagram on how I drew a tri in voxels.*

2 Importing Meshes

2.1 `importObj(..)`

First we take the mesh in `ngl::Obj` format. I iterate through the triangle faces of the mesh. Each mesh I pick an edge. I iterate down the edge. For each iteration I iterate down a line towards the vertex opposite the selected edge. The steps I iterate down the lines in halved voxel sized steps. I have halved them since in some cased holes appear.

At the final iteration we add voxels at each step. As we iterate through the two lines (the first edge and then the line) we interpolate the normals, and UV coordinates.

Now as you go down the line you notice that the lines get closer together and are getting closer than a voxel length. Intuitively, the gap between lines should half, half way along since the lines are going to same point. So half way down you can cull every other line out. Then you can half again counting from after this. And so on. You would stop when left with two lines of voxel apart.

Calculating distances between lines is hasslesome. So I halved them down until an 1/8 in the actual library. The polygons are never that large to bother. Although one of my demos has a crazy big mesh at one point, the algorithm holds together well even if it does take a full few minutes.

See figure 1 for a diagram. It came out of my own head. The poly to voxel converters I saw in papers looked horribly complicated and mathematical and the code I found looked even worse. My method is probably horribly in-efficient as it uses the length of a vector a lot (calculate the step). Which is why I used glm::vec3 instead of ngl::Vec3 as I assumed some sort of efficiency gain (sorry Jon but for what it's worth I didn't actually test this at all).

The interpolation aspect of the algorithm is inspired by Phong shading. In fact the normals would probably classify as Phong shading except we do it per voxel not per pixel (fragment).

One path of logic says that I would have had to interpolate the normals and UVs at some point so why not addVoxels while I'm at it instead of horrible mathematical plane equations.

There are four optional boolean values for the `importObj(..)` function.

1. `_normals` - Whether to calculate and shade normals.
2. `_interpnormals` - Whether to interpolate normals (true) or just use face normals (false). Only gets evaluated if `_normals` is set.

So if you were using the library without my demo you would most likely have `_normals` set. Otherwise it would invalidate all the normals generated. My demo specific shader checks the normals and if out of bounds it knows this flag to be set to false and will not use normals.

2.2 `importTexturedObj(..)`

There are two functions. `importObj(..)` takes obj, size and color and creates the obj in VoxFoxTree with that color. `importTexturedObj(..)` takes a `ngl::texture` instead of a color. It will actually grab the RGB values from the texture (using the interpolated UVs) and put them inside the voxel and not wait for the shader to do this. This allows us to have multiple objects with different textures within the same shader and VoxFoxTree.

3 Set Operations and Translation

3.1 Translation

I have included union, intersection and subtraction set operations between VoxFoxTree. Each node has its own respective operators. At LeafNode it ORs the m_VoxelMap and merges the m_VoxelData list (in correct order in respect to indexes).

Translation was tricky as originally I would move the origins of each node using offsets. If the origins moved out of the parent node then you change the parent node. This got tricky as when you move the node across you may iterate through that later. So I needed to store a copy of the VoxFoxTree to iterate through.

When you change the parent you need to delete the node from the old parent's list. But since I was iterating through the copy of the VoxFoxTree it was complicated to get that pointer and list. Especially since I was removing from vectors. The indexes would change etc.

At the leaf level, to shift by voxels it got even more confusing. I would have to shift them by the number in one axis, then the opposite direction by (8 - amountToShift) in another tree. Then translate this other tree by a leafUnit in the direction of translation. Then union the two trees. However it gets horrible since if that translation of a leaf goes over the SecondaryNode's bounds (the problem in the previous paragraph).

In the end I simply took all the voxels coordinates and data in two vectors. Then translated all the coordinates. Cleared the VoxFoxTree (including accesssors). Finally addVoxel for each voxel in the vector. This method is easier to read and write. The other method was like a super hard suduko when attempted in code.

3.2 Union

VoxFoxTree finds all PrimaryNodes that both trees have using idx, idy and idz of PrimaryNodes. Then we call the union operator on the PrimaryNodes. This happens for all nodes until we get to LeafNodes. Here we do the actual unioning. Simply bitwise & each `m_VoxelMap` char. Then we sort out the common `m_VoxelData` elements.

All the nodes that are not shared between the trees are also added to the result node.

3.3 Intersection

Beginning stages are very similar to union in that we find all the common nodes. At leaf node level we do: (LHS & RHS). Then we sort out the LHS voxel data that is shared.

3.4 Subtraction

We find all common nodes. At LeafNodes we do: LHS & - (LHS & RHS). Then we sort out voxel data so that it is LHS that is not included in RHS.

4 `calculatePolygons()`

This was one of the first functions I got off the ground. It converts the tree into polygons to render with OpenGL. The idea was to eventually use another method of rendering. A more efficient method more suited to voxels. Such as perhaps ray tracing or view aligned slicing. However I didn't have enough time or even technical know-how to get to this point in time.

Essentially it iterates through every single voxel. It checks if it has any immediate voxel neighbors. If it has an immediate neighbor on one side, it will not calculate polygons for that side. Since most of the time the neighbors are in the same leaf no complicated lookup is required. When it does fall outside the LeafNode we use `isVoxel(..)` which makes use of accessors to speed it up. Calculate polygons should really be called calculate vertices. Since it plops all the data for each vertex. All vertexes for a given voxel shares the same data (color, normal, UV) except for `blockCol` voxels.

These `blockCol` Voxels were created by my shape functions. Since the shapes are full (unlike the imported meshes grrr...) I could not have normals for the voxels inside. The normals change depending on the surface. So if an object is cut open by a subtraction operator the normals would change depending on the cut.

So I faked shading for the polygons. I assigned different colours for each face, some darker and lighter than the others. This gives the shapes generated a more voxel-ly appearance than pixel-ly (compared to the imported meshes which use normals per voxel not polygon - each voxel is completely coloured one colour).

The output are the four vectors of floats contained within every VoxFoxTree.

- `m_vertexes` - three floats per vertex
- `m_normals` - three floats per vertex
- `m_textureCoords` - two floats per vertex
- `m_colors` - three floats per vertex

Normals between -1.0f and 1.0f, texture and colors between 0.0f and 1.0f, and vertex coords are in world-space.

5 Demo

The demo included uses the Examples.h class in the library. It simply runs one of nine examples. Just change the number on line 153 in demos/NoGUI/NGLScene.cpp.

6 Testing

There is one test project that tests the entire library. This is because most of attributes of LeafNode, SecondaryNode and PrimaryNode are public. Since VoxFoxTree needs access to most of it's nodes contents.

Unfortunately I could not load in ngl::Obj objects without crashing the project. When I tried to initialize NGL using NGL::NGLInit it crashed also. However I believe I tested all functions apart from the importObj functions appropriately. The set operators can be tested very well with the shape functions (createSphere, createCylinder etc). Some would argue more so since the shapes are full and not empty obj imports; you are operating on more voxels.

7 Conclusion

I know this pdf isn't necessary but I find that you get a better overview this way rather than shifting through some hastily written comments in code.

I learned in full force the C++ Golden Rule of Three. At one point I failed to implement an assignment operator when implementing copy constructors and came across many errors. So I am now more learned.

I am pleased with the aesthetic quality of imported objs. Especially having left the metallic rendering quality in the shader. It makes more use of those delicious normals. I also do like the simplicity of using the set operators. If I had more time I would implement a GUI using QT and perhaps keyboard shortcuts to switch between trees.

Figure 2: *Large deer, no normals.*

Figure 3: *Large deer, interpolated normals.*

Figure 4: *Small deer, no normals.*

Figure 5: *A tiny shiny deer.*