Quixel Terrain Engine Documentation

Last Updated 4/14/2014

Table of Contents

1. Introduction

Hello! Welcome to the Quixel Documentation. My name is Gerrit Jamerson and I'll be walking through the source code here. Please note the project is open source under the MIT license. As noted in the readme.txt, use for commercial projects is of course allowed, though I'd love to hear about any projects being worked on, and I'd be awfully humbled if my name appeared in the credits.

Quixel started as a hobby project in the summer of 2013. The initial post on the Unity forums can be found here, when the engine was still infantile. After a few months of off-and-on work with QT, I became aware that it had become something more than just a simple terrain engine, and that I might be able to do something with it. After a few more months of work and heavy deliberation, I finally (today, on 4/14/2014) made the decision to go completely open source and release it on the Unity forums.

Though I truly love the idea of open source and contributing to the community, the real reason for going open source like this is because of the sudden impact of real life, and now I have more important things to keep my intention on than what started as a hobby project over summer between school years. Since I no longer have the time to continue working full time on this, nor will I in the close future, I've decided that there's no reason to keep a (mostly) working voxel terrain engine in the closet where nobody can use it. I will be able to provide some bug fixes however, as well as answer questions on the forum.

If anyone out there is interested in continuing work on this, I would definitely love that. I hate leaving something with potential unfinished, and if someone else is willing to take on the work then I'd be more than willing to collaborate. The only downside to that is that I want to keep the work open source. Other than that, if someone has a bug fix they're willing to share, I'd deeply appreciate that too.

Well anyway, thanks for reading my boring life story. I hope you enjoy the blood tears and sweat I poured into this project, and I look forward to seeing anything cool made with it.

Gerrit Jamerson

2. Setup and Use

Setting up Quixel in Unity:

1. Download source from git: https://github.com/Chippington/Quixel
2. (Optional) Create folder to hold the Quixel Terrain source files.
3. Create new script, attach it to an object that will become the terrain parent.
4. Use the following as a template:

```
using UnityEngine;
using System.Collections;

public class TerrainController : MonoBehaviour {
        public Material[] materialArray;
        public GameObject player;

        // Use this for initialization
        void Start () {
                //Set the smallest voxel size to 1 unit, and the largest LOD to 8.
                //Must be called before QuixelEngine.init(...);
                QuixelEngine.setVoxelSize(1, 8);

                //Set the 'camera' object to the player. This is what the engine uses
                //To determine with nodes to load.
                QuixelEngine.setCameraObj(player);

                //Initialize Quixel with the material array and use this game object as
                //the 'parent' terrain object. All nodes for the terrain are created
                //under the parent.
                QuixelEngine.init(materialArray, this.gameObject, "testWorld");
        }

        // Update is called once per frame
        void Update () {
                //Updates all components of the Quixel Engine.
                QuixelEngine.update();
        }

        void OnApplicationQuit() {
                //Terminate Quixel. This must be used or the threads would have `
               //no quit flag.
               QuixelEngine.terminate();
        }
}
```

5. Create materials and player object.

   You can use up to 255 materials with Quixel. When using the NodeEditor, the material ID corresponds to the material's location in the material array. Only material with index 0 will be used for terrain generation (with the default terrain generator).

   The player object can be anything. It is used as a reference for what nodes the node manager should load. There can only be one 'player' object at the moment.

6. Compile and run!

Depending on the settings you chose for Quixel (LOD size and max LOD), the player object may move too fast to keep up with generation. In this case you have two options:

> 1. Increase the initial LOD size -- Something like 3 or 4 is good for use with Unity's default FPS character controller. However this will decrease the quality of the terrain.
>
> 2. Decrease the player's movement speed/size. Tweak the player's settings so that the movement will be slow enough to allow the Quixel engine to keep up.

Remember to change the render distance in the settings if you want a particularly large setting. The level of detail system allows for some large and beautiful scenery.

Editing the terrain with NodeEditor:
Note: At the time of writing this, editing terrain that doesn't have a default lod size of 1 will not work, and will throw a lot of debug errors at runtime when you attempt to edit the terrain. The problem lies somewhere in the NodeManager and it's search functions.

Note 2: All the editing functions can be found in NodeEditor.cs.

I tried to make terrain editing simple when making this engine. As a result of this and lack of work on terrain editing, it's not terribly optimized. Here's an example of using the box brush:

```
//Set material to paint
NodeEditor.setMaterial(materialIndex);

//Brush using a box of width 3, with hard density.
//Applies the material to the working area as well.
//lookatPos is assumed to be where the player is looking.
NodeEditor.applyBrush(NodeEditor.BrushType.BOX, 3, lookatPos,
       NodeEditor.DENSITY_SOLIDHARD);
```

The following NodeEditor functions are available for editing terrain:

`setMaterial(byte matID)`

Sets the material ID to be used when applying changes to the terrain.
matID corresponds to the material's index in the material array passed in at init.

`applyHeal(BrushType type, int size, Vector3 pos)`

Resamples the terrain in the brush area. Essentially removes any changes applied.

`applyBrush(BrushType type, int size, Vector3 pos, float val)`

Apply the brush to the terrain, with given width/radius and density.
There are static densities available for use:

DENSITY_SOLIDHARD - Solid with hard edges
DENSITY_SOLIDSOFT - Solid with soft edges
DENSITY_EMPTY - Empty space

`applyPaint(BrushType type, int size, Vector3 pos, bool regen)`

Paints the area with the set material.

That's pretty much all it takes to get started with Quixel. Any further tweaking with the engine will have to be done by editing the source.

3. Source Files

This is to be used as a quick reference to find something you're looking for. Each function (and some classes) has a <summary> tag that explains what it does. Any further specification will have to be done by you or by asking in the thread/contacting me.

a. QuixelEngine.cs

Centerpoint of the Quixel terrain engine.

b. NodeEditor.cs

Tool used for editing the terrain.

c. Nodes.cs

NodeManager Class

Handles the top 27 (3x3x3) parent nodes, as well as searching for nodes.
Also handles the procedural creation and deletion of nodes via the camera obj.

Node Class

Contains density and material information.
saveChanges and loadChanges is where save files are created/handled.

d. MeshFactory.cs

Centerpoint of thread control.

generateMeshData(...) (Called by build thread)

Where the marching cubes implementation is begins.
Three loops are used to calculate densities, normals, and triangles in order.
The mesh's data is put into a MeshData object and passed to the node.

generateTriangles(...) and the rest of the functions

Marching cubes implementation. The lookup tables can be found here.

e. ITerrainGenerator.cs

Interface used for creating your own terrain generator.
Implement the calculateDensity function to change how terrain is generated.
See section (5) for how terrain generation works.

f. Threading.cs

GeneratorThread Class

Continuous thread that checks the MeshFactory for nodes that are waiting to be generated. Uses a queue for mesh requests.

DensityThread Class

Currently unused due to bugs.

Originally used to create and destroy density arrays in a way that wouldn't affect the fps as much. Unused due to a bug in which the same density array is passed to more than one node.

FileThread Class

Continuous thread used as the paging system. Checks if nodes are waiting to have changes saved/loaded and does so in the thread to reduce fps impact.

g. Pooling.cs

ChunkPool Class

A gameobject pool created for the purpose of reusing chunks that have been removed. Saves fps since Unity doesn't have to keep instantiating and deleting objects.

DensityPool Class

Currently unused due to bugs (see DensityThread)

h. Utilities.cs

DensityData Class

Contains density and material information for a node.
Also has methods used for RLE compression used when saving node data.

Triangle Class

Triangle data created for specific use with Quixel.
The first three points are location, and the last three points are normals.

Vector3I Class

Integer vector created for specific use with Quixel.

MeshData Class

Contains the data that Unity uses for creating a mesh. Passed to the node that requested a chunk generation. Since mesh creation cannot happen in a thread, I tried to do as much work as possible in the thread before passing the data to Unity, which hangs the game until completion.

4. Terrain Generators

I'll do my best to explain how terrain is generated, but NVidia has a cool GPU Gem article written here that explains it really well, and it just so happens to be the article of inspiration that started this whole project.

When you implement IGenerator, only one function is required: calculateDensity(...). The world coordinates are passed through (as a Vector3) for the single point that is being sampled, and an instance of VoxelData is returned as a result.

VoxelData has two fields:

(float) Density - Density of the point
(byte) Material- Material of the point

The density is, obviously, the solidity of a single point. This value is used in the marching cubes algorithm to determine where polygons are created. Since polygon vertices are created along a line between two points (aka an edge of a voxel), the vertex position is interpolated between a point below the isosurface value (5f by default) and a point above. A density that is above the iso value is considered empty space, while a density below is considered solid space.

So how do you use this to generate an entire 3d world? Why, use a noise generator of course! Any will do, but I suggest you use something fast or else raise your initial LOD value to compensate. I personally enjoyed using this Simplex noise generator as a very fast alternative to perlin noise generators.

The default terrain generator (found in ITerrainGenerator.cs) will simply create a flat plane along the y axis at -50f. You can easily add hills and mountains by adding perlin noise. For example, adding:

```
d += Mathf.PerlinNoise(xx / 300f, zz / 300f) * 300f;
```

After `flaot d = yy - (-50f)` will form rolling hills (though with an ugly seam that cannot be avoided with the use of Unity's perlin noise). Change it around to:

```
d += Mathf.PerlinNoise(xx / 300f, zz / 300f) * 200f;
d += Mathf.PerlinNoise(xx / 2000f, zz / 2000f) * 1900f;
```

And now we have some mountain-looking things with the hills added to it. Essentially, the perlin noise works like so:

```
d += Mathf.PerlinNoise(xx / stretch, zz / stretch) * amplitude;
```

Where ‘stretch’ is how much the perlin noise map is stretched, and amplitude is how much it affects the terrain’s height. A large stretch with high amplitude will produce large scale mountain ranges, while a small stretch with high amplitude will produce a rocky/spikey landscape.

A neat mountainous terrain generator can be found here, though you’ll need to download SimplexNoise to use it properly.

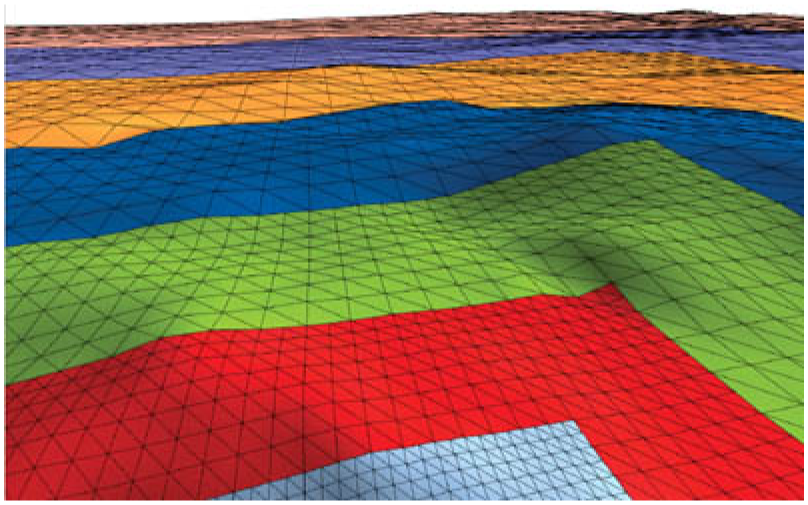
---

5. Level of Detail Overview

The entire world is stored as nodes. Every node contains a 16x16x16 array of voxels, which is used to create the node's mesh. The nodes are handled in memory as an octree. Starting with the top level nodes (A 3x3x3 array of nodes that have no parent), each node contains an array of 8 (2x2x2) subnodes, which in turn may contain more subnodes. This repeats for however many times until the max LOD value is reached.

The smallest node (LOD 0) will have a width of the initial LOD width passed through when QuixelEngine.init(...) was called. Each successive level of detail is twice as large as the previous, and so for each time you go up a level in LOD the quality of the terrain decreases by half.

This is essentially what the terrain mesh looks like with different levels of detail:



Because of this system, there are noticeable seams that form between two differing levels of detail. I have no way to solve this issue as of now, and so I'm currently using a cheap trick of lowering distant terrain nodes to hide the seams. As a tradeoff, there is a somewhat more noticeable popping effect when more detailed nodes are created, replacing parent nodes.

5. Contact Information

I’m always willing to talk if you have questions or concerns. Hit me up on skype or send an email and I’ll get back to you as soon as possible.

Tumblr: http://sirchippington.tumblr.com/
Email: Scythix@hotmail.com
Skype: scythix