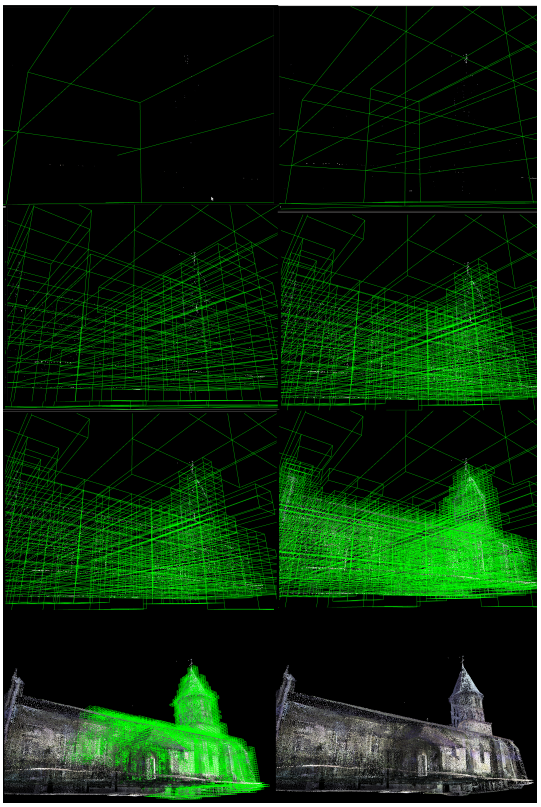
PointMan: Pointcloud Rendering

Nathan Marshak and Uriah Baalke

December 12, 2013



1 Introduction

Large point clouds are becoming an increasingly common representation of 3D range scan data. At the same time, there is great demand for users to access data and applications via the web. WebGL point cloud rendering of large data sets necessitates the development of data structures like octrees that can be transmitted from server to client. Furthermore, it is desirable for the octree to stream from the top down, rather than force the user to wait for the entire tree to load.

2 Approach

2.1 Octree Builder

The octree is first built using a C++ application on the server. The "vanilla" definition of an octree stipulates that there is at most one point (i.e. a point in the point cloud) at each leaf. In order to reduce the size of the tree, we allowed for there to be at most 50 points in one leaf. If the number of points exceeded 50, we split the leaf recursively into more leaves, and distribute the points among each. Finally, we store points in the interior nodes that represent some sampling of the leaves, rather than just in the leaves. Again, this is in contrast with the "vanilla" definition of an octree. We do this in order for points to appear, even if we haven't reached the leaves, and to facilitate level-of detail. The points inside an interior node i are determined as follows: for each child c of i, randomly sample $\frac{n_c}{k}$ points from c and store them in i, where n_c is the number of point in child and k is the number of children. Therefore each interior node holds roughly the average number of points that each of its children have. This ensures that the upper levels of the tree contain a reasonable number of points.

2.2 Python Server

The octree and data points are serialized, and then read by a Python server based on the Tornado framework. Octree nodes are indexed in order of BFS traversal, where the root has index 0. Parents are related to children using the

following formula:

$$Child(i) = 8k + i + 1 \quad (1)$$

Where k is the index of the parent, and i is the octant the child is in (ranges from 0 to 7). By giving each octree node a unique index, and by relating parents to children via indexing operations, the octree can be stored in a hash map, and links between parents and children do not need to be stored. The server receives requests for children from the client as a list of indices. Then the server looks the relevant nodes up in a hash map, and sends them to the client as a list of JSON objects. The server is thus stateless, and does not need to keep track of where the client/clients are currently in the tree.

2.3 JavaScript Client

The client is written in JavaScript and WebGL. The client keeps track of a "front", which contains (roughly) the current level of the octree that needs to be rendered. The client only stores the current front and not the entire octree - when the client needs to access children, it queries the server, receives the children, then throws out nodes that are not currently being rendered. This should reduce the memory footprint of the octree. Finally, the client automatically traverses the octree in a series of exchanges with the server - the client renders while it is requesting additional nodes from the server, and the client can be operated interactively while waiting on a response from the server. Once the additional nodes are loaded in the client, the client may make another request to the server, and walk down the tree one more level, until the end is reached.

2.4 Performance Comparison

The biggest performance problem we ran into was the octree being so large that it crashed the client! We solved this by making it so that we stored 50 nodes at each leaf, rather than 1 node at each leaf. Then, at each interior node, we randomly sampled the children in order to populate the interior nodes of the octree, facilitating progressive refinement while loading. This drastically reduced the octree's size. See the table below, which lists how many total nodes there are in the tree, versus the max number of nodes per leaf.

Nodes/Leaf	small chappes	chappes
1	38025	1917363
50	1867	79067