Adaptive Point Cloud Rendering

Project Plan

Final

Group: May13-11
Christopher Jeffers
Eric Jensen
Joel Rausch

Client: Siemens PLM Software
Client Contact: Michael Carter
Adviser: Simanta Mitra

4/29/13

Table of Contents

1 Problem/Need Statement

Siemens PLM creates software that allows companies to import models of various components and locations to analyze various details such as collision and dimensions of an assembled object. Most models used in these programs have the data that allow them to be rendered as polygons. Some hardware used in the industry, however, creates models that are created of entirely points. Siemens PLM does not currently have software that allows them to render entire models made entirely of hundreds of millions of points, models also known as point clouds. While they have the capability to render points if they are part of a model, if a model consists of more than a few thousand points the rendering program will not perform at desired levels.

The purpose of our project is to create a medium-scale (100s of millions - billions of points) point cloud rendering kernel that will render a displayed section of the point cloud efficiently. This project is about feasibility as much as it is functionality. We are not sure if we will be able to complete the project to the desired requirements, and much research must be done to figure out the most efficient methods to complete our task. Testing will also be a large part of this project, to confirm that our methods work correctly and efficiently. The two main components of this program will be the data structure we choose to represent the point cloud data and the algorithm we choose to decide which points are rendered.

2 Implementation

2.1 Architecture

Before a point cloud can be used by the application, it needs to be converted into the QSplat tree structure format. This is done by the preprocessing tool.

The QSplat PCR application performs two major tasks: traversal and rendering. Each of these occurs in a separate thread of execution. Communication between them is confined to thread-safe mechanisms. The following diagram depicts components of the application, and what they are used for. Components which are in-between tasks are shared between tasks. The components in the diagram will be explained in detail in the subsequent sections.

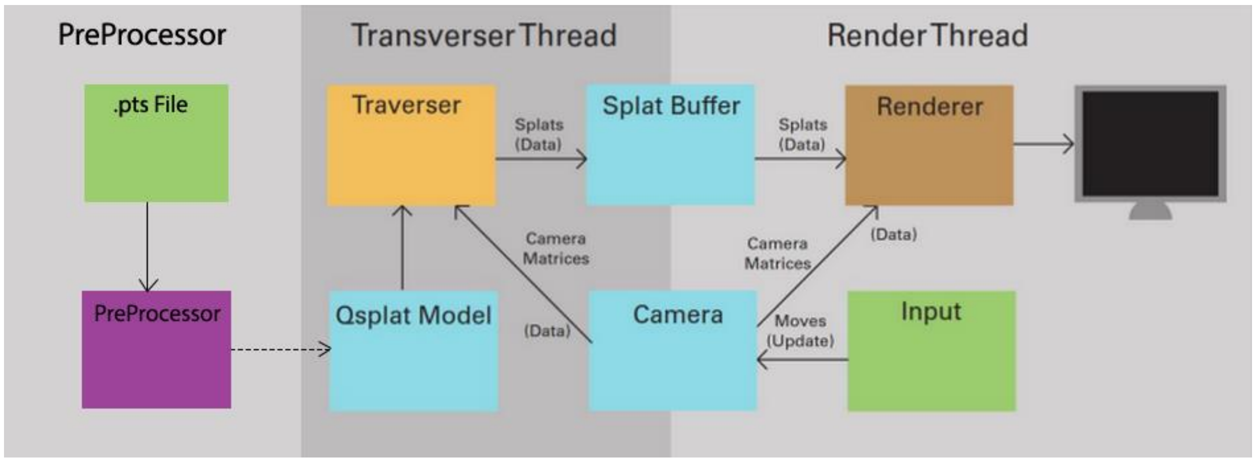

Figure 1: Architecture Diagram

2.2 Preprocessing

The goal of the preprocessor is to load a point cloud file (in the .pts format for our project) and output a .qs file that contains the information of the QTree Structure. This preprocessing only needs to be done once for a point cloud that we wish to render. Once we have the .qs file, we can load that into our main rendering application.

The process begins by parsing the number of leaves and loading the information for each point into nodes that will eventually be the leaves in the tree structure. After all points are loaded the tree is created by grouping close points into spheres. These spheres are also grouped, until there is one single sphere representing the entire point cloud. After the tree is completely constructed, we write out the .qs file. A header containing some general information about the tree structure is written, then the information on each sphere is written in the following format.

Position, radius	Tree structure	Normal vector	Normal cone width	Color (optional)
13-bits	3-bits	14-bits	2-bits	16-bits

- Position and radius are encoded as offsets relative to the parent and are quantized down to 13-bits.
- The first two bits of the tree structure tell the number of children of the node, the last bit tells whether or not all children are leaf nodes.
- Normals are quantized down to 14-bits.
- The width of the normal cone is quantized down to four possible values: Cones whose half-angles have sines of 1/16, 4/16, 9/16, or 16/16.
- Color is stored using 16 bits (RGB as 5-6-5).

2.3 Traversal

Traversal describes the process of selecting a set of visible points from a point cloud model. Once the set of visible points is selected, they are handed to the renderer to be drawn. This is done by finding a rough approximation, and incrementally refining it to get the final result.

The point cloud model is a file in the QSplat tree format which is mapped into memory. This allows the operating system to dynamically page parts of the tree in and out of memory. Memory mapping reduces the memory footprint of reading the file considerably compared to loading the file into main memory. The savings become significant given the size of the files the application works with.
The traversal and rendering tasks communicate through a splat buffer, and the virtual camera. The camera tells the traverser when it needs to do something (camera moves), and the buffer is used to transfer visible points to the renderer. Multiple transfers may be made while the traverser is refining the image. Each subsequent transfer contains more detailed information about what is visible to the camera.

Refinements can be noticeable to the user; it may appear that the image quickly sharpens after interacting with the model. To provide a smooth transition between the different levels of detail, the tree is traversed taking the size of the approximations into account. This allows features of the model which are close to the camera to improve more quickly than distant features.

If the camera moves, the traversal process needs to be restarted. This may occur when the traverser has or has not completed the current traversal. If it is in the middle of a traversal, it needs to bail and begin again given the new camera position.

A number of techniques are employed to determine which points are visible which includes frustum, cutoff, and occlusion culling. The recursive nature of QSplat’s nested sphere hierarchy allows entire branches of the tree to be culled, making these techniques very powerful.

The virtual camera “sees” a three-dimensional area which is shaped like a pyramid with the top chopped off. Frustum culling checks if points fall within this area. If they do not, they are not visible to the camera and can be discarded. This is a standard optimization in graphics processing.

QSplat’s hierarchy supports another type of optimization: cutoff culling. This is used to stop traversing a branch of the tree when the approximations become smaller than a pixel. At this point, there is no further benefit in continuing to traverse the branch. Distant objects in the scene can be represented using far fewer splats without any noticeable difference in image quality.

Occasionally in a scene, features will eclipse more distance features of the scene. The more distant features are hidden behind closer features. Thus, the distant features are occluded. This can be detected by keeping track of the closest object visible at a pixel using a depth buffer. Occlusions can be detected, and culled by comparing the distance of an approximation to the screen to the depth buffer.

These optimizations allow the traverser to quickly determine what is visible in the scene. The process is incremental, with periodic updates sent to the renderer while the image is refined.

2.4 Rendering

In rendering our point cloud model to the screen, there are several options available. By limiting ourselves to only point data, we can eliminate most mesh options. This would leave real-time mesh construction methods (for example, several terrain generation methods use height value and a two dimension grid to create 3D terrain meshes) or methods that render each point independently. Choosing to focus on the latter, we selected splatting as the method that most matched our needs.

Splatting is based off of the idea of throwing paintballs against the canvas and with enough density, the result can be seen as an approximately solid model. Splatting in itself means that points are represented with polygons called splats. (Technically speaking, anything can be used and still be called a splat, even if it is a point or some complicated 3D model.) For our needs, we wanted our splatting method to be able to create a more accurate representation of the surface than just using a uniform splat. Therefore, each point is treated as a sample of the surface, giving us a position, a color, a normal vector that represents the surface's orientation, and a radius that represents the size of the sample. By having the normal vector, we can perform lighting calculation (which depend on the normal, look up Phong lighting model) and are able to orient the splat in 3D space. By having the radius, we can have non-uniform splats, which are required for the QSplat implementation to have its levels of detail.

We designed two splat shape generating methods, regular polygons (using triangle fans, choosing any polygon, draw point at the center, connect that point to the vertices with lines, and you will have a fan of triangles) and area defined with a polar equation. The former is a standard graphics primitive. We used hexagons as the best balance of performance and quality. The latter is based off of computational textures. Basically, you define a square (quad), and define one corner to be (-1, -1) and the opposite corner to be (1, 1). When the quad is processed and broken down into possible pixels (fragments), each fragment has an attribute that represents its interpolated value on this 2x2 grid. For our uses, this attribute is treated like a vector and the fragment is discarded (not drawn) if the vector has a magnitude greater than one. Thus, drawing a circle.

We designed and completed three ways to splat these shapes: depth correct, billboard, and perspective correct billboard. Depth correct is a basic method, where we draw a splat in its appropriate orientation and location in space. This method can cause splats to cross each other. For billboards, we force the splats to be parallel to the view-plane, so splats will never cross and the splats will appear to reorder themselves when changing the point of view. Perspective correct billboards, do the same things as the standard billboard, but correctly handle interpolating values with different depths. We discovered that perspective correctness provides little to no visual benefit for a processing cost.

For blending, we are using order-independent adaptive blending. We started to develop a method that maintained a shorter sort-list of just the closest fragments, but were unable to complete this, because of time. Atomic functions do not allow 64-bit exchanges, so GLSL software mutexes are required for this. Another idea for blending is to use the GPU to sort the splats (or use indexing and have the CPU sort it) and then use hardware blending. It must be noted the blending requires billboard, but depth-correct leads to artifacts caused by the changes in order when the splats cross each other.

The graphics shader pipeline that we created for drawing a splat using GL_POINTS:

- Vertex Array: Position, Color, Normal Vector, Normal Cone (not used), Radius
- Vertex Shader – Passed vertices to geometry without modification, excluding the blending vertex shader which multiplied the radius by two.
- Geometry Shader
    1. Compute i and j unit vectors for the plane using the normal vector.
    2. Create a splat of inputted radius using i, j, and the position as the origin.
        a. For hexagons, create six triangles.
        b. For quads, create a two-triangle triangle strip.
    3. For billboards, multiply the vertices by 1/w and set the z-value to the center's z/w-value.
    4. Output primitives, with view-space normal, light direction, and eye vectors, color, and, for quads, texture coordinate. (Note: We used a single point light, so light direction is not constant across the splat. The normal is constant, because the normal cone's 2-bit resolution does not provide enough information to approximate details.)
- Fragment Shader – Without blending, apply standard Phong lighting (no emissive component, with white light, and diffuse color for the specular color, for effect) to fragment. For quads, test the vector length of the texture coordinate, if greater than one discard.
- Blending Fragment Shader – Requires OpenGL 4.2+, after coloring the fragment, pack it into an unsigned integer (we are using LDR color only), and then add it to a per-pixel atomic linked-listed data structure (We used load/store images for compatibility, but shader storage buffers want also work.) Formally called, order-independent, adaptive transparency.
- Resolve Fragment Shader – Per-pixel, load the first set of stored fragments into arrays (an array per store attributes, we stored depth and color), sort the arrays by depth, calculate the color by applying the standard blending equation back-to-front.

4 Operating Environment

The operating environment for this application will be on a Windows computer with the hardware required to render a point cloud in the latest version of OpenGL (3.3 or higher). The programming language our program will be written in is C++ using OpenGL libraries.

3 User Interface Description

The user will be able to load a point set and interact with it in the application's primary use case. A single rendering window and a menu bar will allow this. The user may load a point cloud using the file menu. Keyboard and mouse input will be captured to allow the user to navigate the point cloud one the point cloud has been loaded.

4 Requirements

4.1 Functional Requirements

- Load a billion-point point cloud provided by Siemens
- Allow the user to rotate, zoom, and pan on a point cloud model

4.2 Non-Functional Requirements

1. Maintain an interactive frame rate of no less than 10 fps at 1080p
2. Render the highest quality representation of the point cloud
3. The rendering kernel shall be written in the C++ programming language.
4. The rendering kernel shall use the OpenGL library.
5. All data must fit into a few gigabytes of main memory.

4.3 Inverse Requirements

1. The application shall not divide a point cloud into discrete objects.
2. The rendering kernel shall not convert a point cloud into triangle meshes.
3. The rendering kernel shall not perform dynamic lighting on a point cloud.

5 Deliverables

This project is a feasibility study as much as it is a development project. Multiple algorithms are being evaluated using common metrics. Performance analysis of the different algorithms, along with the implementation of at least one algorithm in our application, will serve as the deliverables for our project.

6 Planning

6.1 Risks

- Using either a CPU-side or GPU-side approach will not perform at the required frame rate in the final product.

Strategy: Prototype possible methods to gain a greater understanding of their performance and possibly implement multiple methodologies for the final product.

- Obtaining real world data may not possible or practical during our development. Strategy: Define a common format that most real world sources could be easily converted into.

6.2 Schedule & WBS

See attached file.