Adaptive Point Cloud Rendering

Design Document

Final Version

Group: May13-11
Christopher Jeffers
Eric Jensen
Joel Rausch

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

4/29/2013

Table of Contents

1 Introduction 3
1.1 Problem Statement 3
1.2 Deliverables 3
2 Feasibility Study 3
2.1 Overview 3
2.3 PCL to Render Using Octrees 3
2.4 Ray Tracing 4
2.5 QSplat 8
2.6 Analysis 10
3 Implementation 10
3.1 Architecture 10
3.2 Preprocessing 10
3.3 Traversal 11
3.4 Rendering 12
3.5 Testing and Performance 14
3.6 Coding Standards 15
4 Appendix 15
4.1 QSplat PCR User Manual 15
4.2 QSplat Preprocessor User Manual 17

1 Introduction

1.1 Problem Statement

Digital 3D models are used in industry during the design process. Our client, Siemens PLM, creates software to allow businesses to view and compare 3D models. One type of 3D model a business may obtain is called a point cloud, which is a large group of points in (x,y,z) space with no connections between the points. Most hardware and software is optimized to handle the far more common polygon-based 3D models, and Siemens PLM currently has no program capable of rendering large models made completely of points.

1.2 Deliverables

This project is a feasibility study as well as a development project. Multiple algorithms were evaluated using common metrics. Performance analysis of the different algorithms, along with the implementation of our chosen algorithm, QSplat, will serve as the deliverables for our project.

2 Feasibility Study

2.1 Overview

There are several papers available covering the topic of point cloud rendering. Of these, using the Point Cloud Library (PCL) to Render using Octrees, Ray Tracing, and QSplat were selected for further analysis. Each group member was responsible for prototyping one of these. After comparing the results, QSplat was selected for the final implementation.

2.3 PCL to Render Using Octrees

This method for rendering the point cloud involves recursively traversing the data structure, an octree, to see if a point on the screen should be rendered or not.The open-source Point Cloud Library has an octree implementation so we decided to do preliminary testing using that. An octree is a tree structure in which each node has exactly eight children. If we start at the finest level of detail and combine 2 x 2 x 2 blocks of points in space, then continue to combine the 2 x 2 x 2 blocks into single nodes, we can create a tree structure that eventually leads up to one node. Traversing down the tree from this node will result in following branch nodes, which will all have eight children, until a leaf node is hit, which has no children. This structure allows the data to be compressed into a format that is comprised of bytecodes that represent which areas in space contain points or not. Reconstructing the point cloud is just a matter of traversing the octree and rendering points in the locations that correspond to the leaf nodes at the end of the structure. It is possible to accomplish different levels of detail by setting a maximum level of depth to traverse in the octree. The algorithm will stop at this maximum depth and render points

for leaf nodes and branch nodes that exist at that depth. Branch nodes that are rendered are approximations of their children nodes.

The implementation of the octree structure will require a recursive algorithm to construct the tree structure with the proper bytecodes. Another recursive algorithm is needed to traverse the octree and actually draw the splats. Both a depth-first and breadth-first traversal are possible, but the depth-first traversal is less complex and uses less memory space.

The following is pseudo-code of the depth-first traversal of the octree structure.

decode(transformed center position $\mathbf{p}'$, recursion level i)

    if $i = 0$
        read normal index n, **material index** c
        set pixel($\mathbf{p}'$, **shading**(n,c))
    else
        read byte code $[\chi_1 \cdots \chi_8]$
        for $j = 1, \ldots, 8$
            if (χ_j) **decode**($\mathbf{p}' + \mathbf{d}'_{i,j}$, $i - 1$)

2.4 Ray Tracing

Ray tracing in standard graphics study is a method of following the path of a photon from the light source to the camera, as it bounces off various surfaces. Ray tracing generates its path using ray casting. Ray casting is the method of detecting when a straight path from a point hit an object. Ray tracing uses the surface that the casted ray hits in order to calculate the refracted and reflected light rays. One ray grows exponentially until the path terminates at the camera or leaves the scene. Ray tracing is the primary method for simulating real world lighting without a heuristic in a simulation.

For a point cloud representation of a model, ray tracing is impossible because surface is unknown (and taking the time to calculate the surface, renders using a point cloud model useless, as you will have made the faster geometric model pre-computable). However, ray casting is possible, as long as we define when a ray hits a point (see Ray Casting subsection). So let us set aside the need for accurate lighting and focus on detecting what the camera sees without shading. In others word, cast a single ray and see what hits, but do not follow the path past that one ray.

In ray tracing one can start from the light source or the camera and still produce the same results (think of it as finding all paths between two nodes in an undirected graph, any path from A to B is a path from B to A, therefore the set of paths is the same from either direction). Starting from the light source in point cloud gives us every point that is lit by the light source. But we have stated that we do not care about shading, so this result is unhelpful. So let us start from

the camera, this would give us every point that can be seen of the camera. In other words, by casting rays from the camera, we can filter out any point not seen by the camera, because there is no visible difference between not rendering a point and rendering over that point (we are ignore transparency at this time, but the method for including it should be apparent).

To summarize what we have so far, we have a method for filtering all points that are obscured by other points, by defining the position of the camera in three dimensions. We can further filter out points by defining where the camera is looking and its field of view, namely we only cast rays in the viewing frustum.

For our final filtering step, we use our knowledge of how a scene is displayed by the GPU, namely every object is rasterized into pixel-aligned fragments. For a point object that is only one fragment is generated. Without transparency, there can only be one fragment per pixel that can become a pixel. So we have two choices for heuristics. Option one, cast one ray per pixel. Or, cast multiple rays per pixel and blend the results into one object. The latter would give a high quality antialiasing result at a significant cost (potentially greater than standard anti aliasing methods). The former is the option most often used in practice.

To summarize this basic method that we have described so far, for our scene made of only point clouds we will cast one ray per pixel to select the points that will be rendered. So regardless of the point cloud size, we will only render a maximum of the number of pixels of screen resolution (e.g. 1920x1080p has 2,073,600 pixels) points. Based on connection types available in 2012, namely HDMI 1.4b's maximum resolution, we will never render more than 8,847,360 points (ignoring antialiasing methods), which is orders of magnitude under the billions of points in a point cloud that we are trying to render.

The usage of the word "select" needs to be emphasized. This method does not render the model. It only filters the points. By associating a ray with a pixel as a permanent structure, we can have a list of rays that will be used over and over again to filter the scene. Moreover, if we store the ray's result, we only need to calculated the result when the camera changes. This means that we can have two independent threads, one that renders and one that updates. Therefore, rendering and update are concurrent tasks and the visual frame rate is determined by the length of this ray list.

Now, to avoiding giving the wrong impression, it must be noted that rendering one point at a time is inefficient. Each node of this list must be a set of rays, with the results being stored as vertex buffer object data (hopefully, stored on GPU memory). Also, use of the term point in rendering, does not have to mean a point primitive, but would probably be a splat. In addition, in order to have shading one must have or approximate the normal of that point, which is out of the scope of this methodology.

Let us assume we have normal vectors and discuss quality. Because we are just filtering points, we do not introduce resolution. We can view the model from any scaled and all detail is preserved. Because no information is lost or approximated the image generated is a near-

perfect, consistent representation of the point cloud scene. Interactivity is maintained at all times (with a reasonable computer) by the fact that rendering and updating are independent. Even if the update lags, the scene will still render at the high frame rate, just from a different viewpoint with needed points just missing until the update catches up.

Based on our research, this method produces the highest quality of any other. The downside of this method is that it is a simplified version of the slowest, highest quality method for shading in computer graphics.

In ray casting there are two standard structures: static k-d trees using boundary limits and dynamic bounding box structures. For the scope of this project we will not be considering dynamic models and so the need for dynamic structures is not present and therefore we shall pass over the dynamic structures. (As an aside, using a static structure is faster than using dynamic structure statically, because dynamic structures are built to be changed quickly, whereas static structures are built to hold to a spatial invariant. This spatial invariant speeds things up significantly.)

In static points clouds the primary structures are k-d trees and octrees. Now, because we are using point clouds, the k-d tree using boundary limits reduces to the point cloud k-d tree. Now since we have already discussed octrees independently, we need to explain what a k-d tree is and then focus on the comparing the two for ray casting.

A k-d tree in its most basic form is binary search tree (BST) in k-dimension. In one dimension they are equivalent. In adding dimensions, every tier of the tree defines an axis for comparisons (e.g. Tier 0 compares x components, Tier 1 compares y components, and Tier 2 compares z components). Which tier uses which axis and the order of axes need not be deterministic, but for our uses we shall use the repeating sequence of X, Y, Z.

Because of this splitting of the tree invariant across multiple tiers, unlike the BST, a k-d tree cannot by rebalance or self-balancing. In BST, one can rotate nodes. However, k-d tree this does not work, because the tiers that share a common axis are no longer next to each other, change a node’s tier breaks the tree invariant. With the basic structure describe, we shall continue on with the comparing to the octree and deal with greater detail as needs.

For starters, we must address the resolution feature of the octree. By defining the maximum level of detail for the scene one can reduce the number of points being stored, thus saving space at the cost of detail. It is both good and bad. Because of this duality, we shall assume for our comparison that no points are lost and that the virtual point and real point are the same, so both k-d tree and octree are losslessly equivalent for this comparison. This leaves us with backend complexity and spatial uniformity.

Starting with the latter, the octree clearly wins with it representing a uniform three dimensional grid. One can clearly picture the scene at design-time and therefore can easily optimize cache structures and the casting algorithm based on intersections of cubes and lines. However in

terms of traversal complexity, k-d tree wins hands down. The binary tree of the k-d makes traversing it simple, with four choices: go left, right, both, or none (only with at a leaf) calculated from one line-plane intersection. The traversing the octree requires the line-cube intersection of ray with each child voxel to be calculated and may require four children to be traversed.

In terms of space complexity, the k-d tree also wins. If our scene has n number of points, the k-d tree will n number of nodes. Whereas, with an octree all points are stored at the leaves and the depth, d of the tree is a user settable value, so the upper bound of nodes would be O(d*n) nodes for a loosely filled tree.

In terms of time complexity from the perspective of tree depth, the octree wins. All leafs in an octree are same depth, even if the tree is unbalanced. A k-d tree shares its worst case for an unbalanced tree with its binary search tree brethren, namely a depth of n. A k-d tree suffers even worst, because it cannot be rebalanced. However, with some care in building the tree, one can avoid (but not prevent) this case.

In deciding which structure to use we must take in account where we are implement their algorithm. On the GPU, within the graphics pipeline, caching information spatially and limiting the length of loops are important factors, so octree is needed, despite the added complexity. For a CPU-side approach, uniformity in space is unnecessary, so we can use the faster k-d tree (faster, in terms of the complexity of choosing to traverse a node and the number of nodes) and remember that like quicksort, there are cases that will suffer in performance.

Since our group is currently developing CPU-side rendering approaches for testing, from hereon we shall be discussing this methodology using a k-d tree backend.

Create a list of rays and result sets associated with the camera perspective settings and the number of pixels in resolution of the viewport in length. Concurrently, update and render this list. An update will consist for all rays of updating the ray to the new camera position, look, and up vector and casting that ray. Rendering shall walk the list and draw the result sets to the frame.

Pros:

- Independently updating and rendering
- No resolution effects
- Consistent number of points being rendered that are independent of the point cloud
- O(n) memory space requirements
- O(R lg(n)) average-case update time, where R is pixels of resolution (unproven)
- Depth Buffer rendered is consistently with standard geometry.

Cons:

- No filtering can be done to reduce memory footprint.
- Worst-case ray cast is O(n) using recursion
- Resolution changes are costly or become a scaled image of the original resolution.

Using a k-d tree that records the bounding limits of ranges and that records if there is an equal-axis child in front and if there is an equal-axis child in back, relative to Cartesian axis directions.

Steps:

1. Test if the ray is on the node splitting plane and store.
2. If Step 1 is false, traverse the child closest to the origin. Else, if there is an equal-axis child closer to the origin, traverse the or-equal child.
3. If Step 2's child returns a point, return that point.
4. Test if the ray hit this node's point.

   Where d is the dot product of the direction of the ray with itself, f is the dot product of the origin ray relative to the node's point with direction of the ray, e is the dot product of the origin ray relative to the node's point with itself, and r is the square of the radius of the spherical represent of all points:

$$\big((f \geq 0) \land (d(e\text{-}r) < 0)\big) \lor \Big((f < 0) \land \big(f^2 \geq d(e\text{-}r)\big)\Big)$$

5. If Step 4 is true, return this node's point.
6. If Step 1 is false, return empty. Else, Test if the ray intersection with the node's axis.
7. If Step 6 is true, traverse remaining child.
8. If Step 7's child returns a point, return that point.
9. Return empty.

2.5 QSplat

The QSplat algorithm is a recursive splatting algorithm. In this type of algorithm, groups of points in a point cloud are approximated by a single rendering primitive. Ellipses are used because they allow adjacent splats to be blended together to create the illusion of a smooth surface. QSplat is capable of producing high quality renderings. It also, however, has level of detail scaling which is a useful property in an interactive application such as this.

The quality of renderings can be refined when the camera is stationary. Inversely, QSplat is also capable of creating unrefined renderings quickly. A rough approximation of the point cloud is first rendered. Higher quality features may be added during subsequent frames if the viewport remains stationary. This process of adding more detail is called refinement by QSplat.

The data structure was designed with the goals of being dense, allowing level-of-detail scaling through refinement, and having reasonable pre-processing times. The data structure is implemented as a bounding sphere hierarchy. Bounding spheres are arranged in the tree such that rough details are always above their refinements. This is similar to the Min-Heap property in binary heaps. The property is used for level of detail selection. While rendering, start at the top of the tree and bail if it takes too long. This gives us an image quality / time tradeoff.

QSplat uses a bounding sphere hierarchy data structure for its compactness, and easy level of detail selection. The data structure used by QSplat is created in a preprocessing operation. Preprocessing takes a point cloud or a mesh and builds the bounding sphere hierarchy from it. This process has an algorithmic complexity of O(n log(n)).

The tree layout allows the data to be very compact. Bounding sphere nodes in the tree are 32-bits without color, and 48-bits with color. This requires a massive, contiguous, block of memory. Memory mapped files can be used for this.

QSplat trees have an upper bound on the number of nodes they can contain, because the child offset fields are fixed at 32-bits. To

The bounding sphere hierarchy is arranged breadth-first as an array in memory. The child or parent of a given node can be found using the following relations:

Parent = nodes[floor(index / 2)]
Children = { nodes[2 * index + 1], nodes[2 * index + 2] }

Information is tightly packed in the bounding sphere nodes by utilizing bit packing and quantization. Quantization is way of compressing a number by mapping bit sequences to a finite number of values. For example, the radius of the sphere can take one of 13 values in the range 1/13 to 13/13.

The table below depicts the bit packing in bounding sphere nodes.

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

The tree is constructed from the point cloud during a preprocessing pass. Normal vectors are also computed at this time, and added to the bounding sphere nodes of the tree. The normal vectors are used for backface culling and shading.

The rendering algorithm will draw splats with increasing levels of detail, or abort if rendering took too much time. An implementation of QSplat for the Digital Michelangelo Project uses a feedback mechanism to maintain a target frame-rate by adjusting the level-of-detail.

The rendering algorithm uses a traversal function which does the following:

1. If node not visible, return.
2. If node is a leaf, draw a splat for it and return.
3. If the benefit of making another recursive call is too low, draw a splat and return.
4. Recursively draw each child node.

2.6 Analysis

After comparing the runtime performance and memory footprint of Octrees using PCL, Ray Tracing, and QSplat, QSplat was chosen for the final implementation.

3 Implementation

3.1 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.

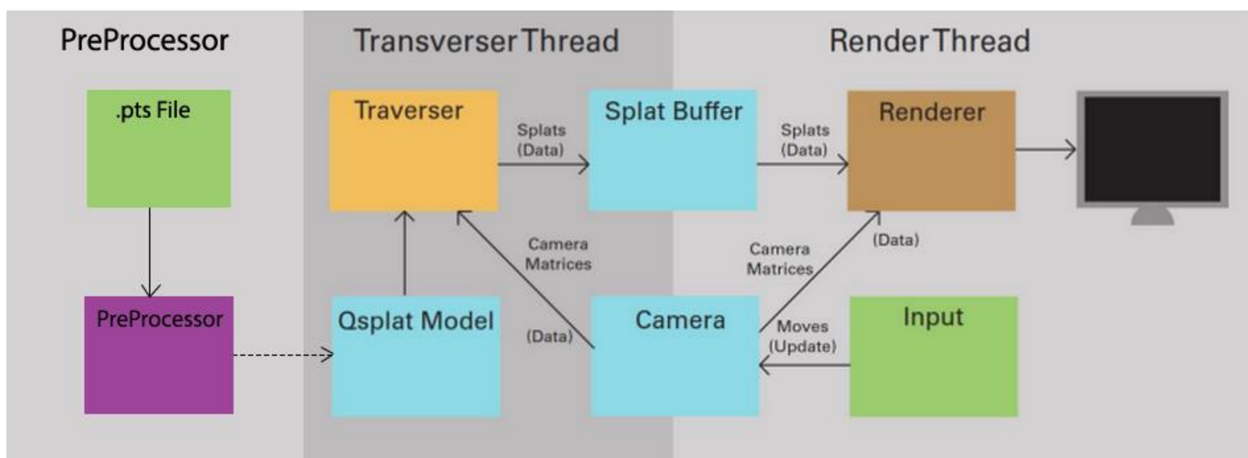

Figure 1: Architecture Diagram

3.2 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).

3.3 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.

3.4 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.

3.5 Testing and Performance

Because we introduced the buffer between traversing and rendering, we decoupled the size of the point cloud from the frame rate. This means that rendering is relative to the size of this buffer. A pixel splat is the smallest splat that can be displayed and we introduced occlusion filtering, so there may only be one pixel-sized splat per pixel. Therefore, the buffer size is equal to the number of pixels being rendered.

We tested our splatting methods against the baseline GL_POINT primitive at 1920x1080. Figure 2 shows the more important results for the frame rate for when the buffer is being updated every frame. It must be noted that the frame rate was capped to 29.4 for these tests.

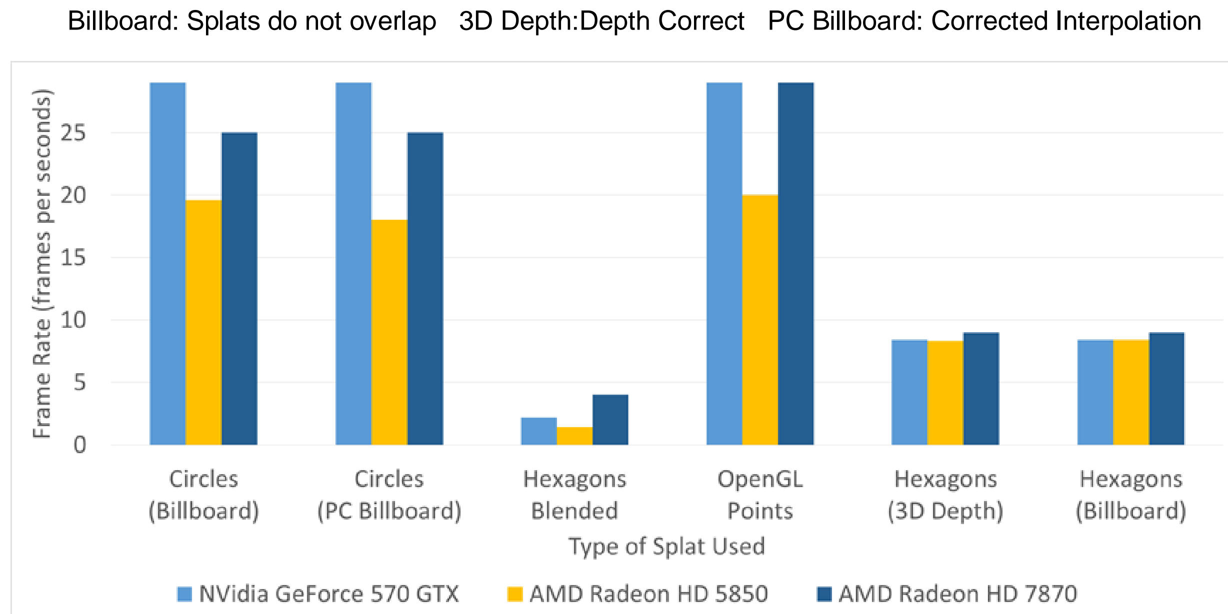

Figure 2. Frame Rate versus Type of Splat for rendering one billion-point point cloud at a resolution of 1920x1080

With our results, we showed that our quad splatting method was able to perform at about a third of the OpenGL points, which for even 60 fps (the refresh rate of most monitors and all televisions) is not an issue. In fact, what is not shown in Figure 2 is that when the buffer is not being updated, all of our benchmark computers were able to max-out the frame rate.

Blending is a major performance hit. It does provide a significant quality improvement over using normal splats when the splats are large. Figure 2 shows the results for blending at a worst case, with blending being applied to splats that are too fine to need blending. These results suggest that there may be a solution to improve performance that uses blending only when necessary.

Regardless of blending, we succeeded in developing a modern QSplat implementation using quad splats that met our performance and quality goals.

3.6 Coding Standards

Source code will be consistent with the GNU Coding Style, with the exception that nested scopes are indented four spaces instead of two.

4 Appendix

4.1 QSplat PCR User Manual

Required Files: QSplat PCR.exe, freeglut.dll, glew32.dll, “Shader” directory.
Required Files under “Shader” directory: axis_f.glsl, axis_v.glsl, basic_f.glsl, basic_g.glsl, basic_v.glsl, blend_linked_f.glsl, blend_quad_linked_f.glsl, blend_quad_sort_g.glsl, blend_resolve_f.glsl, blend_resolve_v.glsl, blend_v.glsl, mock_f.glsl, mock_g.glsl, mock_v.glsl, point_f.glsl, point_v.glsl, quad_f.glsl, quad_g.glsl, quad_pc_f.glsl, quad_pc_g.glsl

The point cloud rendering application is a command-line application which is configured using command line arguments, and manipulated using a mouse and keyboard. The application can be launched with an optional command line argument for the input file. If this is not used to specify the input file, a sphere will be used as the default. The exact form of the command is shown below.

“QSplat PCR.exe” [-model <qsplat-file.qs>]

All arguments are listed in the table below.

Arguments	**Function**
-sphere	Loads a low-point sphere (default)
-stress	Loads a extreme high-point sphere
-lion	Loads model at "Model\lion_color.qs"
-model <qsplat-file.qs>	Loads model at <qsplat-file.qs>
<qsplat-file.qs>	Loads model at <qsplat-file.qs> (Note: any argument starting with "-" and not listed as a command is ignored)
-<combine of two of the following: x, y, z, X, Y, Z>	The first character is used for the look position and the second is used for the up direction. Capital Letter are negative (i.e. Z = -z = (0,0,-1)) (e.g. -xy, means the camera will by looking down the x-axis at the origin, with the y-axis being up)
-height <value>	Sets initial window height to <value> (default is 500)
-width <value>	Sets initial window width to <value> (default is 500)
-max_height <value>	Sets the maximum supported window height. This affects primarily affects blending (default is 1200)
-max_width <value>	Sets the maximum supported window width. This affects primarily affects blending (default is 1920)
-frame_rate <frame_rate>	Sets the maximum frame rate for the application in frame per second. (Default is 30 fps)
-update_rate <multiplier>	Sets the rate that the traverser updates the buffer as a multiple of the frame time (default is 1)
-blend_3D	Blending will use depth correct hexagons splats instead of billboards
-blend_Quad	Blending will use circles

User Interface

Button	Function
Left-click	Hold and drag to rotate the scene. In Light Mode, rotates the direction or the light.
Right-click	Hold and drag to control the zoom. Up to zoom in. Down to zoom out.
Spacebar (Left-click Lower-Right Corner)	Cycle through rendering modes.
Escape	Quits application.
F (Double left-click)	Toggle between fullscreen and windowed.
R (Right-click Upper-Right Corner)	Toggles model rotation on/off.
L (Left-click Lower-Left Corner)	Toggles Light Mode on/off.
+ (Left-click Upper-Right Corner)	Increases the size of a point
- (Left-click Upper-Right Corner)	Decreases the size of a point

Note: This interface is Windows 8 touch compatible.

4.2 QSplat Preprocessor User Manual

The application is launched using two command line arguments, one for the input file and one for the output file. If the command line arguments are not input, the program will close with an error. The exact form for the command is shown below.

“QSplat Preprocessor.exe” <input-file.pts> <output-file.qs>