Point Cloud Filtering using Ray Casting by Eric Jensen 2012

The Basic Methodology

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 lite 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 antialiasing 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's maximum resolution, we will never render more than 8,847,360 points[1] (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.

Method Summary

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 (un-proven)
- 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.

[1] http://en.wikipedia.org/wiki/List_of_video_connectors

Ray Casting

Data Structure

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 past 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. Figure 1.1 shows this issue in a 2D plane.

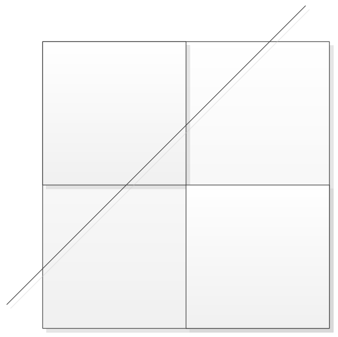
Figure 1.1 – A ray intersecting a 2x2 grid can have 75% coverage. A 2x2x2 cubic grid can have 50%.

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 there 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.

Method (1st Pass)

Before now, we have discussed a ray as a line, but the probability a line intersecting a point is very nearly zero. Therefore, we must redefine this geometry. We have to choose whether to keep the line or the point. We shall ignore the third option of change both, because calculating intersection of volumes adds unnecessary complexity.

In change points into volumes, say into a sphere, one now has a measurable probably of an intersection. However, how does one know if a line enter a sphere and even more important how does one know if this set of sphere has at least one sphere that the line enters? One must sort the sphere into ranges, calculate the lines intersection with that range, and calculate the intersection with every sphere within a candidate range. This is the standard method of ray casting in a geometrical scene using a volume bounding k-d tree. It also can be overkill, if one chooses a volume at whim.

So let us look at convert the line into an object, say a square or circular prism (more on this later). Now the question is: is this point in this range? Our point cloud k-d tree (hereafter, shorten to just k-d tree, unless noted) sorts into ranges already, so just need the figure out the shared ranges and test if the ray volume contains the points.

It should be noted that for certain levels of thought and mathematics these are approaches are equivalent, and we shall use the view that is easiest to understand when discuss how to visualize certain concepts.

The following the basic steps are what the ray casting algorithm will do to perform the hit test of a ray:

Step 1: Test point and return it if it mean the collision criteria.

Step 2: Test whether the node's axis crosses the ray line and traverse the necessary children.

Step 3: If no child exists return no collision or return the nearest (to the camera) non-empty result.

Testing a Point

This is the simplest step. Test. If true, return the node's point. The test is always false for virtual points. For non-virtual points we must choose a volumetric represent to test against.

In terms of how a standard display LCD monitor displays a pixel, a square prism volume ray will represent the square grid the best. View frustum culling tells us that to test this we must calculate the point distance from each side of the prism and test for a negative distance. So for each ray we must compute the four normal vectors based off the up vector of the camera. Then for each point we must do several cross products.

Let us step back for moment and remember that we want speed. Any finite polygon prism will be more costly (except the triangular prism), but what about a circular prism. The question is: is point C a distance or radius R from the line ray, defining by a direction D and an origin O? Well, this can be answer by the point being a sphere and the ray being a line. A ray will intersect a sphere if at least one point on the shell is shared by the ray.

Point P is on the sphere's shell with a radius R must be a distance of from the center C of the sphere, therefore

$$(P-C)\cdot(P-C)=R^2$$

A point P on the ray is describe as

$$P=ray(t)=D\cdot t+O \text{, when } t\geq 0$$

Solving together:

$$(D\cdot D)t^2+2E\cdot Dt+E\cdot E-R^2=0 \text{, where } E=O-C$$

In quadratic form:

$$at^2+bt+c=0 \text{, where}$$
$$a=d=D\cdot D$$
$$b=2f=2E\cdot D$$
$$c=e-r=E\cdot E-R^2$$

Notice that a, d, e, and r are all non-negative values.

The solution for this is:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Combining with our condition on t and noting that result is a Boolean OR on the plus or minus cases:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \geq 0$$
$$-b \pm \sqrt{b^2 - 4ac} \geq 0$$
$$b \mp \sqrt{b^2 - 4ac} \leq 0$$

Now let us reframe this into three cases:

$$\text{Case 1: } b \mp bx = b(1 \mp x) \leq 0, \text{ where } 0 < x \leq 1$$
$$\text{Case 2: } b(1 + x) \leq 0, \text{ where } 1 < x$$
$$\text{Case 3: } b(1 - x) \leq 0, \text{ where } 1 < x$$

Case 1 is true, only when b is negative, c is non-negative, and the part under the square root is non-negative.

Case 2 and 3 occur when c is negative. Case 2 is true, only will b is negative. Case 3 is true, only will b is non-negative.

Combing these cases into a single Boolean form:

$$\Big((b < 0) \wedge (c \geq 0) \wedge \big(b^2 - 4ac \geq 0\big)\Big) \vee \Big((c < 0) \wedge \big((b < 0) \vee (b \geq 0)\big)\Big)$$

Simplified,

$$\Big((f < 0) \wedge \big(f^2 \geq d(e - r)\big)\Big) \vee \Big((f \geq 0) \wedge \big(d(e - r) < 0\big)\Big)$$

By using a circular prism, we need three dot product and three comparison compared to four cross products and four comparisons.

Traversing the tree

For traversing the k-d tree, we just care about the step, one node to its children. So from a structural perspective we have four cases: left child, right child, both children, no child. The last case is the trivial case when we have reached a leaf. Figure 1.2 shows the three remaining case from a ray's perspective.

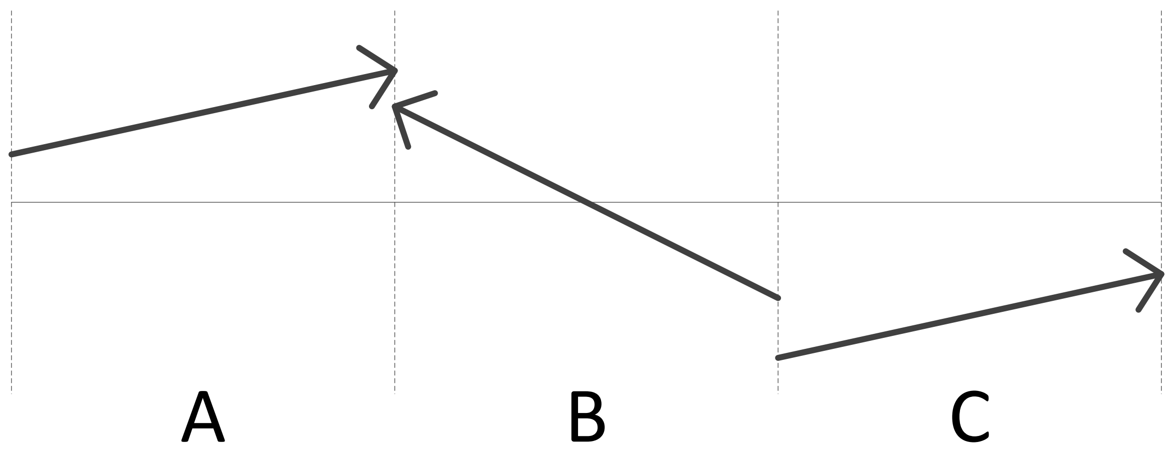

Figure 1.2 – The three cases of a ray interacting with a boundary. Cases A and C are one child traversals and Case B is when both children need to be traversed.

Figure 1.2 also highlights the fact that the plane of the k-d node cannot always have infinite length (a single dimension because the other is defined by an ancestor). We must track the limits of the axis. This can be done during traversal or at built time. We shall take the less time consuming path for now and assume the k-d node has maximum and minimum values as a part of this structure. Additionally, we know our first question: does the splitting plane segment intersect the ray? Our second question: is which child is closer to the ray's origin (a.k.a. the camera's position)?

Our algorithm therefore is simply: If no intersect occurs, traverse the child closest to the origin. If intersect occurs, traverse the child closest to the origin, then traverse the other child.

Now combining Step 3 with this statement and adding some intelligence it becomes:

Traverse the child closest to the origin. If that child returns a point, return child's point. Hit test on this node. If true, return this node's point. Test for intersection. If true, traverse remaining child. If that child returns a point, return that point, else return empty.

The two advantages of this ordering are that points closest to the ray's origin are tested first and the first hit stops the chain. (If one uses an iterative loop instead recursion, you can avoid passing up the returned value.)

Now, before we can continue we must address the special case of the ray being on the splitting plan. When this occurs the closest child cannot be tested and point intersection is undefined. Now we could assumed this this node is the closest and return. But the k-d tree pasts equal-plane nodes to one of the child, so there could be a child that is closer. So we could then just traverse the or-equal child. But then a node that is behind this node could be returned, if there are no nodes in front. So to solve this issue, when building the k-d tree, we shall record if there is an equal-plane child in front and if there is an equal-plane child in back, relative to Cartesian axis directions.

Intersection Testing

To be written up later.

Casting Method Summary

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.

S1: Test if the ray is on the node splitting plane and store.

S2: If S1 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.

S3: If S2's child returns a point, return that point.

S4: 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:

$$\left(\left(f \geq 0\right) \wedge \left(d\left(e-r\right) < 0\right)\right) \vee \left(\left(f < 0\right) \wedge \left(f^{2} \geq d\left(e-r\right)\right)\right)$$

S5: If S4 is true, return this node's point.

S6: If S1 is false, return empty. Else, Test if the ray intersection with the node's axis.

S7: If S6 is true, traverse remaining child.

S8: If S7's child returns a point, return that point.

S9: Return empty.