AFRL-RI-RS-TR-2017-086

ADVANCED VISUALIZATION AND INTERACTIVE DISPLAY RAPID INNOVATION AND DISCOVERY EVALUATION RESEARCH (VISRIDER) PROGRAM TASK 6: POINT CLOUD VISUALIZATION TECHNIQUES FOR DESKTOP AND WEB PLATFORMS

APRIL 2017

INTERIM TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-086 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
ASHER D. SINCLAIR
Chief, Resilient Synchronized Systems Branch
Information Systems Division

/ S /
JULIE BRICHACEK
Chief, Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2017	INTERIM TECHNICAL REPORT	OCT 2013 – SEP 2014

4. TITLE AND SUBTITLE

ADVANCED VISUALIZATION AND INTERACTIVE DISPLAY RAPID INNOVATION AND DISCOVERY EVALUATION RESEARCH (VISRIDER) PROGRAM TASK 6: POINT CLOUD VISUALIZATION TECHNIQUES FOR DESKTOP AND WEB PLATFORMS

5a. CONTRACT NUMBER
IN-HOUSE / R12L

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F / 625318

6. AUTHOR(S)

David E. Kaveney

5d. PROJECT NUMBER
PAVZ

5e. TASK NUMBER
IH

5f. WORK UNIT NUMBER
00

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISB
525 Brooks Road
Rome NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-086

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2017-1880
Date Cleared: 21 Apr 2017

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Design and implement various point cloud visualization techniques for viewing large scale LiDAR datasets. Evaluate their potential use for thick client desktop platforms or thin client web platforms and adapt their implementation when appropriate.

15. SUBJECT TERMS
VISRIDER, Point Clouds, LiDAR, HDR Banding

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**DAVID E. KAVENEY**
U	U	U	UU	44	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1.0 Introduction

This task explored visualization techniques that could be applied to Air Force point cloud data; specifically, LiDAR scans with various categories of embedded data. These techniques were tested and evaluated using desktop and web development techniques to see how much parity can be achieved between the two platforms.

1.1. What are Point Clouds

Point Cloud is a general term used to describe any data set that is a collection of spatial coordinates. Normally these coordinates represent some three dimensional objects or scene. As opposed to three dimensional objects that are made of polygons; point clouds have no topological information which would allow for the formation of polygons. This lack of structural information can make it hard to understand visualized point clouds. With polygonal data, the data more closely mimics the properties of the real world. Even without any other information contained in a polygonal dataset, the surfaces can be easily visualized through simple lighting techniques to give structure and context to what is being represented. Point clouds lack this inherent structure requiring more creative means to visualize and understand them.

1.2. LiDAR

LiDAR is the most common source for point clouds for the Air Force. Lidar (also written LIDAR, LiDAR or LADAR) is a remote sensing technology that measures distance by illuminating a target with a laser and analyzing the reflected light. (NOAA, 2013) This type of sensor can be mounted on a vehicle to scan down streets or fly over a region, or simply used on a tripod to scan a room. The laser's global position and orientation is necessary so the laser returns can be converted into a point that is globally placed in a three dimensional scene. LiDAR scans vary in collection density, denser point clouds are often easier to interpret when viewing them so that the points appear to form a continuous surface. These higher densities allow for easier interpretation of the object or scene being viewed; when viewing a set of points your mind will try to fill in any of the blank space and interpret point connectivity and relations to make sense of the object.

The density of points can vary wildly from multiple meters between points to centimeter or lower density depending on the sensor used, the type of vehicle used, and the number of passes of the collection platform. Scans can be many square miles in size and result in billions of points and many terabytes in size.

1.3. Hardware Description and Limitations

This task deals deeply with computer graphics and a basic understanding of the computer graphics pipeline is important for evaluating the techniques that were explored. The pipeline stages that are most important will be exposed by stepping through the general process of what is involved in drawing a point cloud.

The computer has two main processors: the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). The main application runs on the CPU and performs the needed to setup for the scene to be drawn. The CPU then issues various commands to GPU for it to: copy data from main memory to the graphics card, setup the scene state (colors, viewpoint, data buffer formats), and to start drawing. The GPU then processes the data available to the graphics card and presents the result to the viewer.

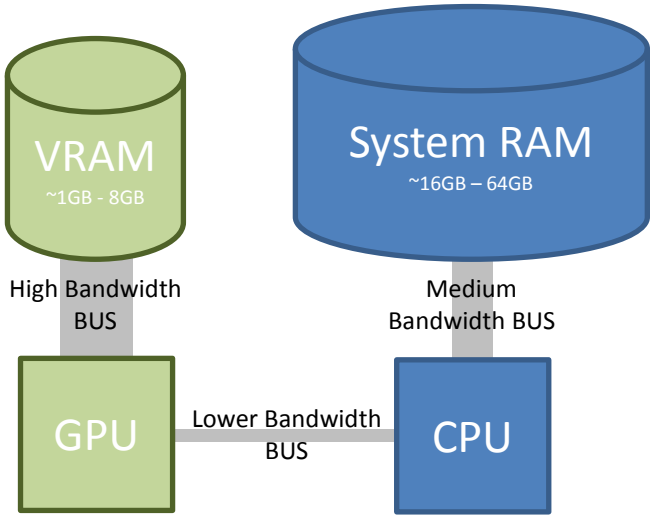

Figure 1 – Typical processor/memory diagram for PC's

The graphics card has a limited amount of memory available to store the point clouds (commonly referred to as video RAM or VRAM), which for graphics cards circa 2015 range from 1GB to 8GB in size. This is a significant limitation when rendering large point clouds, as they can easily exceed the available memory. Management of this memory pool then becomes necessary for rendering large point clouds; moving data in and out of this limited resource intelligently depends on is the points in view or points that will be in view shortly.

1.4. Point Cloud Visualization

Due to the limitations of both the video hardware and the format of LiDAR scans; the goal of this task was to create reasonable representations of point clouds that allow a user to understand a scan through interactive navigation and viewing. There are numerous methods for visualizing LiDAR scans which are just as varied as the diverse data types contained within LiDAR scans. This task explored a promising subset of visualization options that fit within the limited resources of current real-time rendering to provide interactive viewing of point clouds. These visualization methods were explored using traditional desktop application development and web-based development platforms.

2.0 Methods, Assumptions, and Procedures

Within this task there are many assumptions being made that drove our experimentation procedures. The focus was to visualize LiDAR data that was primarily collected from the Air Force. These data sets had certain properties that were very common amongst most collections. This could be they commonly contain or omit certain data types, values were in certain ranges, densities were of a particular degree,

or collection methods were done by similar sensors from similar vantage points. These shaped how the visualization methods were approached, focusing on common characteristics first, and then testing edge cases and possibly augmenting the approach later.

2.1. LiDAR Data Available

Data completeness contained within the LiDAR scan files that we had available was not consistent. There are many possible values that can be stored within a processed LiDAR file; the most common are enumerated here:

- X, Y, Z Position
- Altitude from sea level
- Normal direction
- Intensity
- Color
- Classification
- Collection time
- Global time
- Return number

Even among these categories, most are not present in almost all of the LiDAR data that we have currently encountered (24.5 TB in 580,811 files). Position, altitude and intensity are the most common to appear, followed by collection time and color. With position data and intensity being the only reliable data categories present in the data; those were the ones focused on when developing visualization options. Some ideas and significantly less effort was used on the data that could be, but wasn't, present.

2.1.1. Missing Useful Information

The missing information within LiDAR data such as normals or color could be very useful in visualizing the data. Normals provide a surface direction at the sample location. This sort of information gives the visualization designer more information about the sample of the surface that this point represents. Normals could have easily allowed for interpreting the side of the shape that this point exists upon. Techniques described later in this document highlight the ability to remove sample points that should be occluded due to the structure of the object, but without information about the general structure of the object being viewed, there is no perfect way to remove "unseen" points. Normals would have allowed for a starting place for making intelligent guesses as to what points should and shouldn't be drawn depending on view; not drawing points that are facing away from the viewer as on the real object since they would be impossible to see.

Color is the other very useful parameter normally missing from LiDAR data. Its usefulness is almost obvious, as it would allow for automatic coloring of a point cloud to give the viewer an immediate and accurate representation of the scene. Without some kind of coloring, point clouds quickly fall into unintelligible messes on the screen when rendered, see Figure 2. Point clouds naturally lack structure to help the viewer understand what they are seeing and proper coloring guides their viewing. Using the

color of the object taken right from the scan when point clouds are fairly dense results in an object that looks and acts like a seamless three dimensional picture of the area scanned. Techniques explored in this task focused on using false coloring techniques to bring out details of the objects within the scene, with intensity being the best candidate to exploit as it highlights material differences within objects, bringing out their prominent details.

Figure 2 - Left, a point cloud colored only in white; right, the same point cloud colored with its intensity values. Coloring and shading makes all the difference in understanding the structure of dense point clouds.

2.2. Measuring Point Rendering Performance

While drawing points is the single fastest graphical primitive that graphics hardware can render, using them to render an entire system exposes significant issues. Understanding the theoretical maximum number of points that can be rendered while maintaining a usable framerate was important to understand. The data sets that were being considered contained multiple millions of points and knowing the current hardware limitations greatly fed into the solutions explored.

The majority of tests were written in Java where some were then adapted for JavaScript. Identifying the video card's limitations was not as simple as just drawing points and finding an upper limit. Instead it was quickly found that the video card could draw many more points than there were pixels on the monitor. And while technically we only need one point per pixel to generate an image, the data controls the distribution of points and drawing only one point per pixel is not feasible to identify. Instead, for dense point clouds many points will overlap on the same pixel location. Even though only one point per pixel is needed for the final render, all the points need to be processed to determine which will be the point closest to the viewer at that pixel location.

When conducting early point cloud tests, an unexpected observation occurred; a fairly complex point cloud would achieve a higher interactive frame rate when rendered at a higher resolution. This goes against normal rendering convention; higher resolution normally means that the video card has to do more work to complete the render and take longer. After some analysis of the frame rate when compared to the current view, it was discovered and reasoned that a limitation of point rendering is directly related to the video cards fill-rate and specifically how depth tests are performed. When rendering a very dense point cloud at a small resolution, many more points are competing to be

rendered at a given pixel. Due to how computer graphics handles this problem, each point is checked against the value at that particular pixel and determined if it is in front or behind. If in front of the current value it overwrites that value, else it is discarded. Each point must take their turn doing this comparison; leading to a bottleneck in the rendering as they are all competing to complete their check against the current value. The higher resolutions effectively distributed the points out further which resulted in less of these conflicts and therefore actually rendered faster.

A test was then designed which reduces the number of points rendering conflicts in order to measure the maximum number of points able to be rendered on screen. This test essentially rendered point clouds that contained view-aligned sheets of points that were aligned to the screen pixels in layers. The tester could request any number of sheets resulting in the total number of points rendered would be the screen resolution multiplied by the number of sheets. Tests were run on GeForce 770 GTX with 2GB of VRAM connected to a 30" monitor with a resolution of 2560x1600; after Windows GUI components were taken into account, the rendering surface was 2560x1524, resulting in 3,901,440 pixels per layer. Results are shown in Table 1.

Table 1 - Measured frame rate from point cloud benchmark

Layers	Points	Frame Rate
18	70,225,920	60 fps
19	74,127,360	57 fps
20	78,028,800	54 fps
21	81,930,240	52 fps
22	85,831,680	50 fps
23	89,733,120	48 fps
24	93,634,560	46 fps
25	97,536,000	45 fps
26	101,437,440	43 fps
27	105,338,880	42 fps
28	109,240,320	40 fps
29	113,141,760	1 fps

The table shows how a maximum of approximately 110 million points were achieved before the frame rate stopped degrading gracefully. This is due to the data being much larger than the video RAM, requiring system RAM to be used to compensate. This added latency of fetching data from system RAM eventually became overwhelming to the system resulting in the severe drop-off in frame rate. While this is a theoretical maximum for this system, in practice there is little need to render this many points simply because typical screen resolutions have no more than 8 million pixels.

Testing a theoretical maximum within a web client was significantly more complicated. The WebGL client prototype quickly exposed that even "small" point cloud sets were difficult to load. A region of 14 million points that would load fine on the Java client would not on the WebGL client. This appears to be a buffer size limitation, as breaking the area into smaller chunks allowed for similar performance to the desktop version. The OpenGL binding to the graphics card in WebGL, OpenGL ES 2.0, is a subset of the desktop profile, OpenGL 4, and under this limited profile it is possible to experience different limitations.

Expanding on the knowledge learned from the Java client, and from initial WebGL tests, a rendering limit would be reached within the WebGL client well before the limit found within the Java client. This is due to extra overhead needed within the WebGL client imposed by the web browser, such as a maximum memory allocation limit, and the extra limitations of WebGL itself. Additionally, the web browser environment varies greatly between browsers and between versions of the same browser. Due to this added complexity and reasonable rendering performance in early tests, a definitive rendering limit was not sought, but instead would be addressed should the implemented techniques hit any particular barrier.

Testing per-frame rendering performance for WebGL is particular difficult due to the techniques used by browsers for ensuring security and avoiding dead-lock in the user interface. For instance, the Chrome browser builds up a command buffer for OpenGL operations and will only permit a single web page to issue so many commands before it forces a repaint. This means that the OpenGL commands issued that would normally cause the current thread to block do not, and therefore the standard timing techniques are unavailable.

This simple set of tests were sufficient to determine that raw point rendering performance will not be the dominant factor in point decimation algorithm development.

2.3. Point Visualization Techniques

All point cloud visualization techniques were originally tested in a heavyweight Java environment due to the ease in prototyping and evaluation. The Java environment took advantage of pre-existing tools for building small visualization tests and using the desktop profile of OpenGL allowed for the desired visual output to be the driving requirement not some other limitation. Each of the visualization techniques are primarily designed for accurately displaying point clouds with real time execution and compatibility with the web being a secondary concern. Should any technique prove to be useful; it would then be analyzed for performance enhancements as well as how well it can be incorporated into a web based viewer.

3.0 Results and Discussion

This section outlines the various techniques that were researched for visualizing point clouds. Their reasoning, development and results are discussed below. Many of the techniques below assume the worst case scenario for the point cloud data where the only information is location, other features that could be useful like normals and surface material information is not available. The prevailing thought is that techniques need to work in the anticipated absence of these features and that by solving the harder problem, the better the solution becomes when in the presence of these other point-cloud features.

3.1. Visualization Techniques in Relation to Point Sampling

Display resolution can have dramatic effects on the visual interpretation of a point cloud. For instance, even though points have no volume, when rasterizing a point cloud, a single pixel ends up being the smallest element that can be used to represent any particular point. The higher the resolution of the display, and therefore a higher resolution rasterization, the more distinct points can be drawn while still maintaining a visual coherency with the drawn object. However, depending on the density of the point

cloud, the particular viewpoint, the resolution of the output render and the technique for color the points, it is possible for there to be too many points for the given resolution to represent. This means that the output render will appear to be a contiguous blob and not a set of individual points simply because there was not enough resolution to represent the space between the points, i.e. the points cloud density is such that there are more points to be rendered than pixels on the display. This is shown clearly in Figure 3.

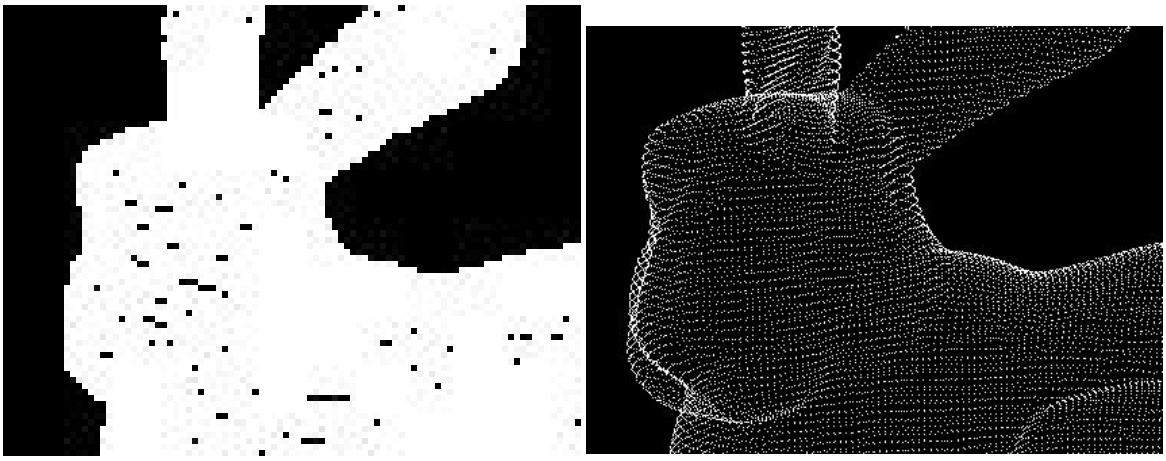

Figure 3 - Resolution effecting point cloud perception: the same point cloud rendered lower resolution (left) and higher resolution (right)

This conceptually bins point cloud visualizations into two categories, under sampled and over sampled. Under sampled point clouds act like you think they should, where individual points are recognizable on screen and it looks like a group of points being viewed. Over sampled point clouds fall into the case described above, where based on properties like sampling density, current view and screen resolution, the point cloud looks to be a solid surface on the screen. Taking this into consideration when designing visualizations was determined to be very important,; certain techniques working in one scenario broke in the other.

It is important to note that these categories cannot be applied to any given point cloud, but only in a particular instance of viewing a point cloud. Since increasing the resolution of the visualization or changing the view to be closer to the points can both switch the visualization from being considered to be over sampled to being under sampled. To make the problem even more pathological, the oversampled or undersampled situation can exist within a single scene where a portion of the point cloud is low enough density to have space between and other portions of the point cloud overlap so much that significant portions all map to the same pixel on the screen.

3.1.1. Point Cloud Transparency

One of the first visualization tests for improving point cloud interpretation was to try using a previously developed algorithm for transparency, using a technique called depth peeling (Everitt, 2001). Point clouds, when uncolored and very dense, tend to look like a non-descript blob on the screen. They are as described above, oversampled. This is shown in Figure 4 where the object, the Stanford dragon, is

approaching that tipping point of having more points in the model than there are pixels of resolution to accurately be able to draw it. As a result most of the model looks white and uninterpretable.

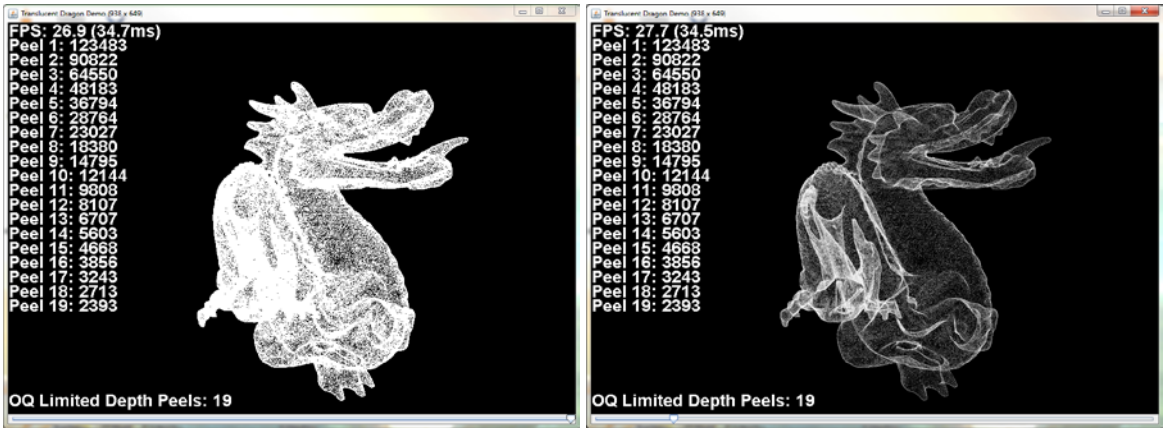

Figure 4 - Dragon point cloud without transparency shading (left) and with transparency shading (right)

Knowing this, the depth peeling transparency algorithm was used as a shading mechanism for the point cloud. Each point was given a uniform transparency and the algorithm naturally will blend the points together. This dramatically improves the ability to see features of the dragon that are hidden in the non-transparent rendered dragon, as seen in the right image. It is much easier to see the local definition throughout the model along with its entire structure. Before transparency was turned on it was much more difficult to make out any of the structures throughout the model other than its major features and defining outer silhouette. With the transparency enabled the model's detail starts to become salient, as the internal structures can be more finely defined with the added shading the transparency effect produces.

This technique does have its problems though. While it does a great job showing off the complexities of the point cloud it does not well preserve the viewer's orientation with respect to the model. What this means is that because point clouds are scans of real world objects, those objects would normally have a natural occlusion that happens where you see the front of the object and the back is obviously occluded. By using this technique and others like it, the viewer can see all the samples of the object regardless of view perspective. This problem is later addressed in the hidden point removal section.

This transparency method could be improved upon, either both technique and implementation. The algorithm uses same colored and same opacity valued points to blend. The method used to combine the over-lapping points is generically defined in the depth peel transparency algorithm as it is supposed to support any object color with any opacity. However, since they are all the same color and opacity this could be significantly improved and made more simply. A simple count of points per pixel and then doing a scaled shading bound by the maximum and minimum counts of points per pixel would be much simpler and faster than executing the entire depth peel transparency algorithm.

Alternatively, the use of normals (either supplied from the data or calculated) could be used to alter the transparency or color of the points. Points that are oriented toward the view could be colored brighter and be made more opaque, while point facing away from the view would be color darker and more

transparent. This is to highlight the sides of the object that are actually facing the view as in real life, while the parts of the point cloud that faces away would be reduced in prominence. This technique would be usable within the depth peeling transparency algorithm, but would require normals to be present among the data.

3.2. HDR (Banding)

An early concept for improving the visualization of point clouds was to color the points based on the distance of the point from the viewer. OpenGL has a built in capability of a depth buffer. The depth buffer is an off-screen buffer that is the resolution of the rendered image. As points and polygons are being rendered, the color of the pixel is stored in something called the color-buffer and the current depth of each pixel is stored in the depth buffer. This depth value can then be referenced and used for various purposes. In this case it was used to alter the color of the rendered pixel.

It was necessary to determine the minimum point and maximum point of the object(s) being viewed so that the maximum amount of color could be applied. While this could have been done per point, the technique applied was to calculate a general bounding box of the point cloud that was being rendered, and the closest and farthest distances were evaluated and saved. These values, along with the depth value of the given pixel, were used to create a gradient of white to black along the view vector of scene. The idea of this visualization was to accentuate the depth of the points in the scene, allowing for the viewer to better understand the normally nondescript point cloud.

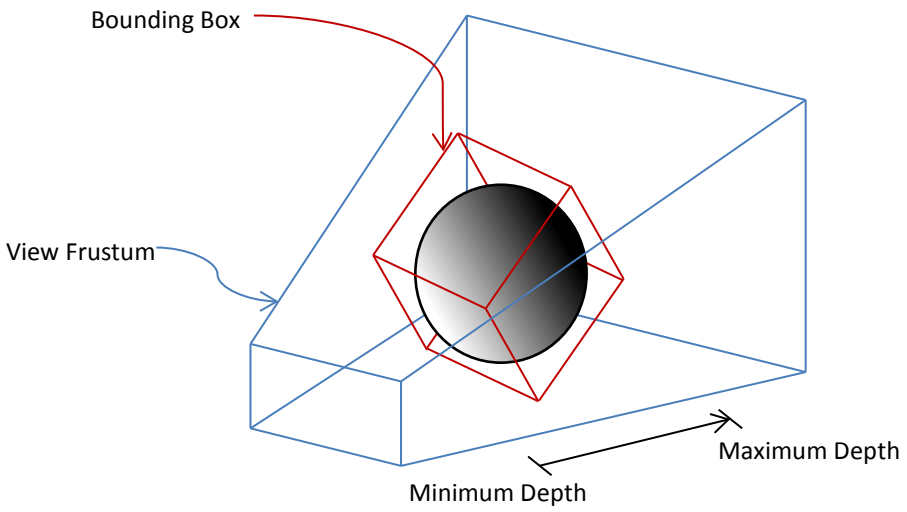

Figure 5 - Depth rendering diagram

This technique worked well, especially for the oversampled point clouds that were tested. Figure 6 shows the application of this visualization. The first picture shows a single colored oversampled point cloud where all of the details except for the silhouette are obscured due to its oversampled nature and no color variation within the object. The top right image shows adding a single gradient to the scene, where the point cloud is colored along the view vector from white for the closest point and black bound to the farthest point in the point cloud. This technique vastly made the point cloud easier to interpret,

with many features becoming readily apparent in the visualization. This gradient only has the ability to show 256 various levels of contrast, resulting in points with similar depths having very similar colors and therefore it remains difficult to determine if they are actually different. The 256 levels of contrast is a limit imposed by the use of standard 24bit color monitors. 8 bits are reserved for each of the red, green and blue channels per pixel. Grey scale gradients are achieved when all the channels have the same value, resulting in 8 bits total worth of data to define the color; $2^8 = 256$, defining the 256 values of contrast.

To increase this perception of values between points, the same gradient was run over the depth twice, running from black to white and back to black again. The idea is that since we were trying to highlight the local changes within the point cloud, a higher rate of change in the gradient should be allocated to a smaller change in the depth values of the points. This approach is essentially allowing for the twice the visual sensitivity to the depths in the visualization versus a single gradient over the depth in the original version. This can be seen in the third, lower left picture below, where smaller details start to become more apparent such as features of the face and foot definition.

Repeatedly applying this concept isn't limited to one pass, so a test was performed where the alternating gradient was applied ten times over the same depth range, alternating black to white and white to black to create smooth transitions over the point cloud. This worked very well in bringing out the features of the point cloud as can be seen in the fourth, lower right image. The face is much easier to see, the clothe patterns are easily understood, and even details within the base are able to be seen. However, this creates a banding artifact across the whole point cloud that is visually displeasing. It is even more unsettling when in motion, as the model is moving through the set gradient field and not directly applied to the points. This means that as the point cloud is manipulated, each individual point is changing color based on where they are in respect to the view vector and depth within the scene. This has a slightly unsettling effect in itself, as the gradient lines tend to warp over the point cloud as it is moved and rotated. Needless to say, this method proved to be effective but too distracting and other means of pulling out details were sought.

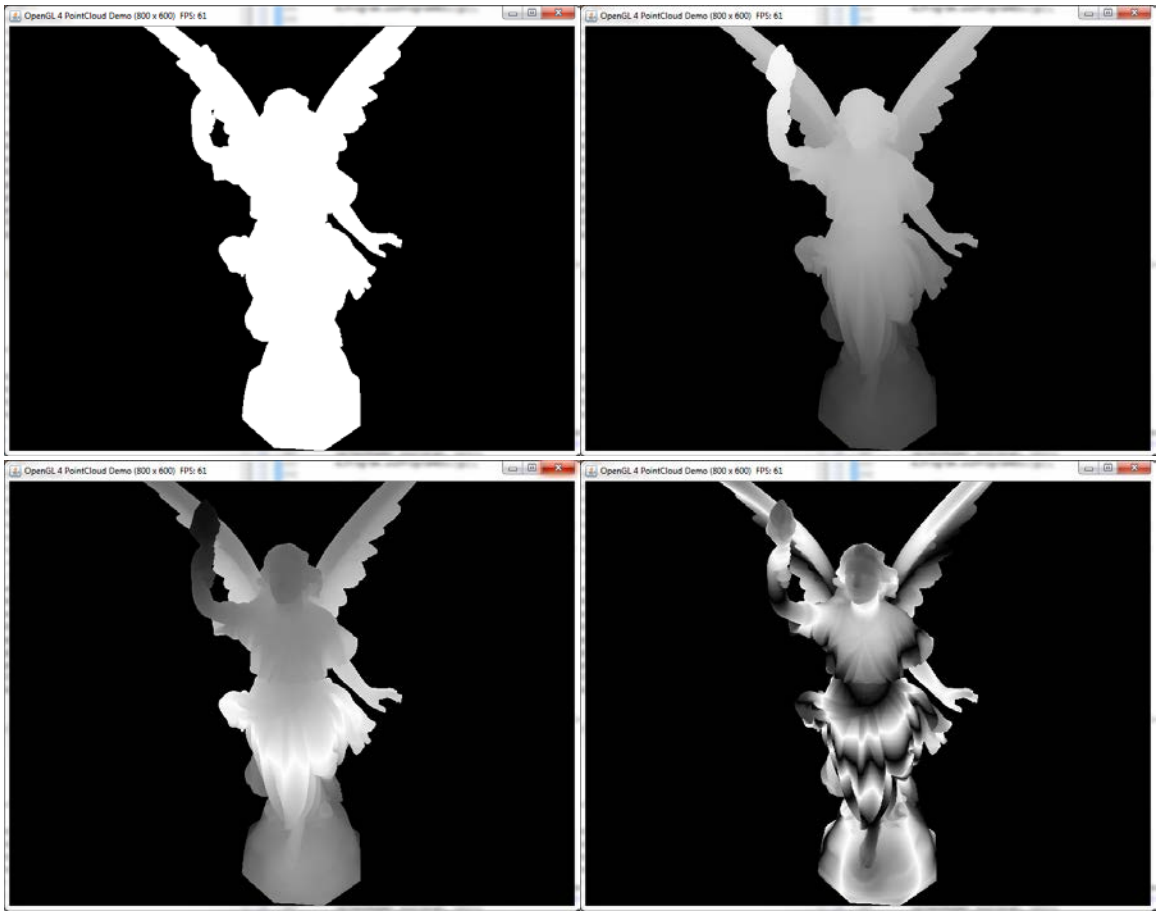

Figure 6 - Various views of Depth Rendering visualization

3.3. Screen Space Ambient Occlusion (SSAO)

Global Illumination is a general rendering term used to describe the realistic propagation of light through a scene. Traditional rendering does not actually model light as it works in the real world; it is an approximation that allows for quick construction of a scene that has enough realism to be understood and useful to the viewer. Global Illumination tries to model the more complex features of light, such as ray bounce and color diffusion. Screen Space Ambient Occlusion (SSAO) is a technique for efficiently achieving Global Illumination that specifically deals with the simulation of light falloff that occurs in areas such as corners. This can be seen in Figure 7 with the effect present and then omitted to show the difference.

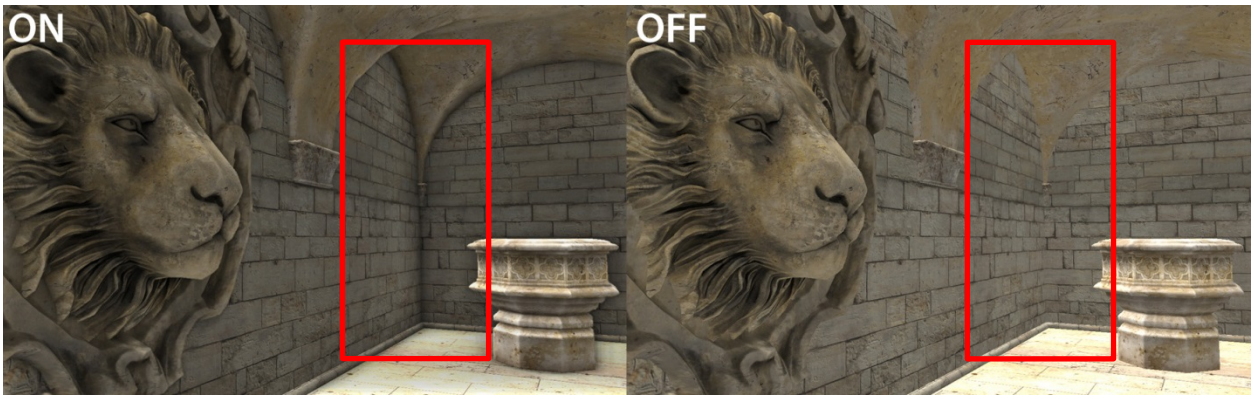

Figure 7 - Sample of SSAO on (left) and SSAO off (right) (Datasystems, 2007). The red highlighted area shows how SSAO can accentuate the geometric structure of a scene through shading.

Notice the added definition in the scene due to the additional shadows, allowing the viewer to see more variation in the scene. These shadows in real life occur due to there being less light reaching that area due to objects occluding the incoming or outgoing rays. This is what is meant by Ambient Occlusion.

The technique Screen Space Ambient Occlusion is an implementation of this effect in screen space; that is to say that it works off the final image that is rendered, approximating the effect based on the limited information available in the final render.

Testing this approach with point clouds was a reaction to the results achieved with the depth rendering test. With the depth rendering test, adding extra gradients to the depth field drew out smaller details within the model. However, this added the undesirable effect of all the bands. Thinking about the problem farther, the aspect of that technique that was most desirable is the ability to highlight the areas within the model that showed important features while ignoring those that were fairly unchanging. While the depth rendering technique was able to bring out these small details, it indiscriminately used the color space along the model. The desired, but unachieved effect in the depth rendering approach was actually very close to the effect that happens with SSAO.

Implementation of this technique was actually abandoned half way through. While the technique is fairly simple to describe, it is difficult to implement correctly. There are many variations of SSAO, and choosing one that would highlight the desired features was tricky. However, with further discussion, it was realized that maybe this wasn't a worthwhile endeavor as the algorithms often rely heavily on connected features which don't align well to the sparse nature of point clouds.

Up to this point, sample objects were used with a very high density of points making them over sampled when rendered; resulting in objects that actually appear as solid objects and not their constituent points. While SSAO will work fine in this situation, when the object is rendered in such a way that the individual points become apparent this technique would break down. This is simply a feature of how SSAO works. For speed purposes, the technique considers only locally adjacent pixels when applying its shading. When the point cloud fills enough pixels to be seen as continuous it would work fine, but once it breaks into individual points due to zooming or drawing on a higher resolution display, there is nothing for the SSAO algorithm to work with in terms of neighboring pixels; causing the technique to

fail. The technique could not be extended to work with neighboring points either, as there is no connectivity information available, and the methods for reconstruction make assumptions that are unacceptable in general.

3.4. LiDAR Render

Next the effort moved from generic point cloud visualization with only location data, to something more directly applicable to Air Force LiDAR. The focus of the LiDAR rendering technique was to generically draw LiDAR point clouds such that a reasonable picture would be produced independent of the LiDAR data source. This is important so that for any LiDAR data, a first pass understanding of the data could be gained. LiDAR typically provides more data categories than just position, which can be used to create richer visualizations. They have the ability to hold many categories of data, but only position is guaranteed. Every scan will have an x position, y position, and z position but additionally, it's possible to have intensity, return number, collection time, normal, or color.

From experience, there have been no LiDAR datasets found that do not contain intensity and altitude. While not guaranteed parameters, they were present in every dataset analyzed, which cannot be said about every other additional data category. While using these other data categories could be useful for visualizing LiDAR, their uncertain inclusion with any particular scan defeats the purpose of creating a generic visualization for all LiDAR scans. The generic LiDAR visualization only utilizes the now assumed data categories; x, y, z-position, intensity and altitude.

Figure 8 - Point cloud of Ottowa, ON shown drawn with only white points

The points were first drawn white, shown in Figure 8, in order to simply a baseline comparison for the other techniques. As expected, when drawing the points with any view encompassing a reasonable sized region, the view becomes a single white mass when rendered. To begin in assisting the user in differentiating points, multiple techniques were layered.

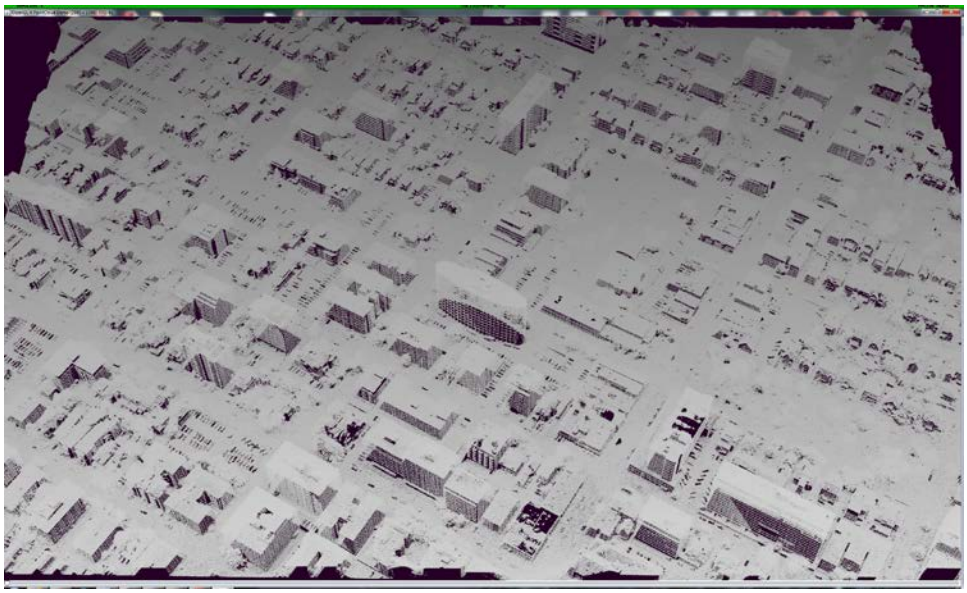

Figure 9 - Urban point cloud shown drawn depth shaded

The first technique was to apply a depth function to the LiDAR to produce the image in Figure 9. The depth function applies a gradient to the points. This gradient occurs over the view space from the perspective of the viewer; shading from white at the closest point to the viewer, to a medium grey. This simple technique added significant contrast to the points, allowing viewers to more easily determine the buildings within the scene versus their surrounding background. Grey, not black as in the depth technique described earlier, was chosen such that points in the distance do not completely vanish from view and any color that will eventually be layered won't be as dark. It was thought that possibly a different color should be chosen, something like blue, in order to mimic the natural atmospheric haze that occurs in real life when viewing something far away, termed blue-shift. This proved to be undesirable to look at, as the blue tended to wash out far away objects. Other colors were attempted with no real preference benefit or better choice found, so white to grey was kept. However, the scene was still fairly hard to distinguish, so other additional renders passes were pursued.

Figure 10 - Urban point cloud shown drawn with intensity values

Further enhancing contrast between objects in the scene was performed by blending the LiDAR provided intensity values. The intensity values, a range of 0-65535, were scaled between white and black bound by the overall maximum and minimum intensity values, resulting in Figure 10. This exposed more of the fine details of the scanned area, vastly improving the understandability of what was being visualized. However this representation still had a flat feel to it where prominent features such as the buildings were still being lost within the visualization. Bringing these features out was the next step.

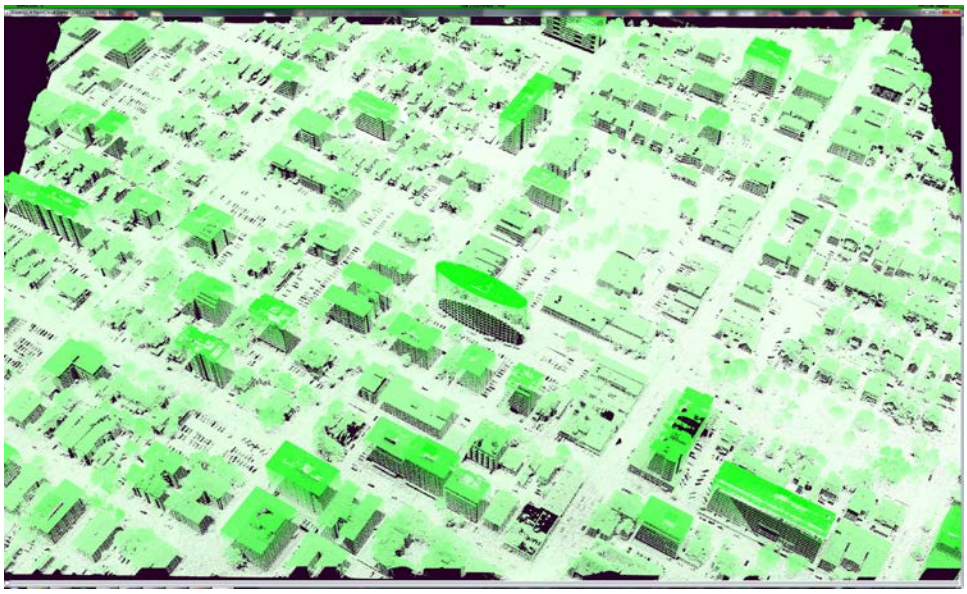

Figure 11 - Urban point cloud shown drawn with altitude gradient

Finally, highlighting or accentuating objects such as buildings, trees, cars, or anything above the surface within the scene was a desired feature of the generic visualization. The concept was to utilize the

altitude component for highlighting those objects, with the assumption being that interesting or important objects would protrude from the surface. Figure 11 is a screen shot of the altitude being drawn from white to green depending on its height from the ground. The altitude component is not drawn based on the entire dataset; it was scaled between the highest and lowest values for the points that are visible within a given view. This is done to show the largest amount of change to the viewer with the limited range of displays and color. However, the altitude component is a measurement of the point's height from sea level, so in scenes where the terrain varies greatly, this somewhat fails to accentuate the objects in the scene, but does preserve subtle variations in the heights of the scene.

It would be possible, although not attempted in this effort, to quickly alter this portion of the visualization so that it more easily shows the objects above ground level. This could be accomplished easily by utilizing DTED (Digital Terrain Elevation Data) for the region. DTED is a model of the ground for the area measured, with various levels of precision measured at set intervals. While not as detailed and fine-grained as LiDAR point clouds, the height values could be interpolated between measured posts to give a close approximation for the ground height of the area. Taking the difference between the altitude (sea level) and DTED (ground level) would give you the approximate height of the object being scanned by LiDAR. This would allow the visualization to color the object appropriately within the scene for the object's height above the ground, and not just height within the scene.

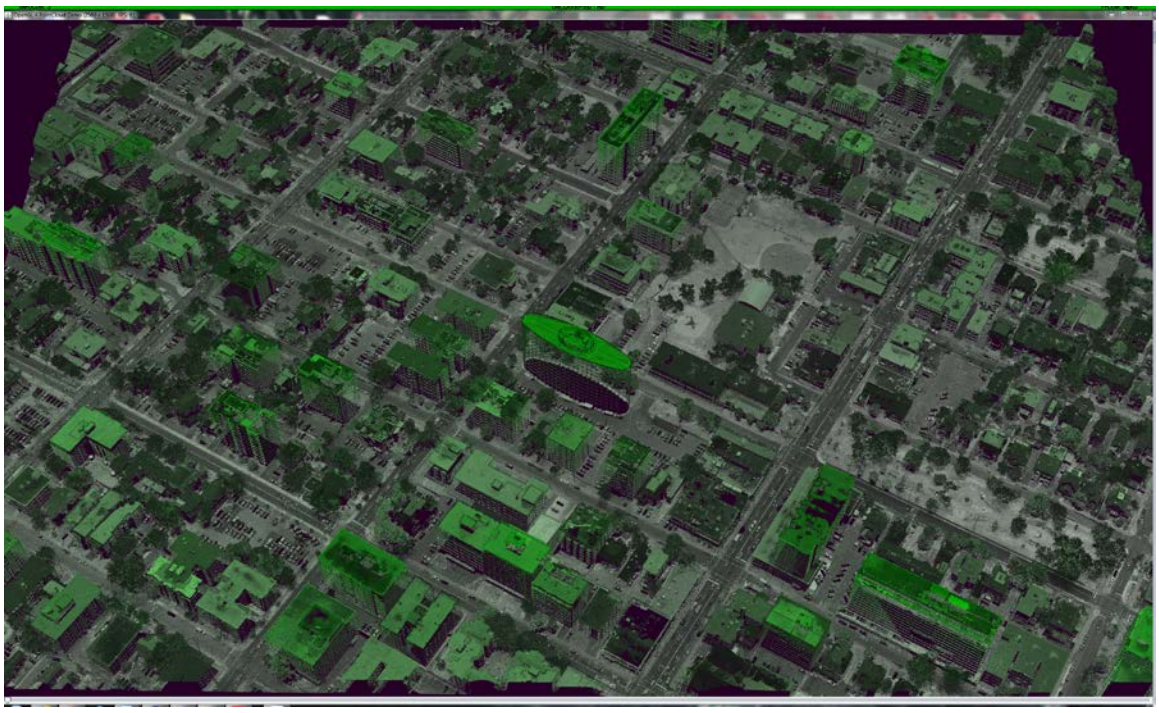

Figure 12 - Composited final visualizing LiDAR in a generic fashion

A composite was then made, with the visualization components being combined into a final render pass, where all the components were calculated independently of each other, but blended together to give the viewer a very good understanding of the scene. Figure 12 is a sample image of all the visualization components combined. It is easy to see how each component aids in the understandability of the

scene. The most immediate thing noticeable in the scene is that buildings and trees are differentiated from the ground with an unassuming green shade while all parts of the scene are enhanced through the use of intensity as the predominant color of the points. Additionally, the subtle fall off from the depth shading keeps the viewer grounded in the distance being observed.

3.5. Hidden Point Removal

When displaying a point cloud, there are structures from the object or scene that the cloud cannot represent, but users infer when they are viewed. Point clouds are sets of object sample points and do not include any kind of connectivity or other structural information. This presents a problem when rendering since a very simple function of real life objects is lost, occlusion. Occlusion is the simple property that when viewing an object the part facing you is visible, while the part in back is not. Point clouds don't exhibit this, as they are just a group of points. The difference between the two images in Figure 13 shows how important this feature is to preserve. The image on the left is a normal point cloud render where the points that are part of the rear of the object should be occluded but are not. The object on the right is the same object with a function applied to logically remove the back facing points that should be occluded (Basri, Ayellet, & Katz, 2007).

Figure 13 - Sample point cloud showing full point cloud on left and hidden points removed on right (Basri, Ayellet, & Katz, 2007)

Notice how difficult it is to make out the direction and orientation of certain features within the face. The ear, for example, in the middle of the head clearly looks as if it is in front, while in reality it is not. The actual ear that is in front becomes difficult to distinguish, as it is confused with features from the back side of the head. The point cloud on the right with the hidden points removed ends up being a much easier point cloud to understand.

3.5.1. Splatting

Splatting is a common technique to add coherency to the view of a point cloud. The technique is accomplished by drawing the points as larger representative spots. These are commonly referred to as splats. These splats are often considerably larger than single pixel points and provide for a simple

means of creating a contiguous surface along neighboring points while also creating a natural occlusion to points along the backside of objects that shouldn't be visible. Figure 14 shows the effect in full force, with the top image drawing points as single pixels, and the bottom image adjusted to have its splat size form a contiguous surface along the ground.

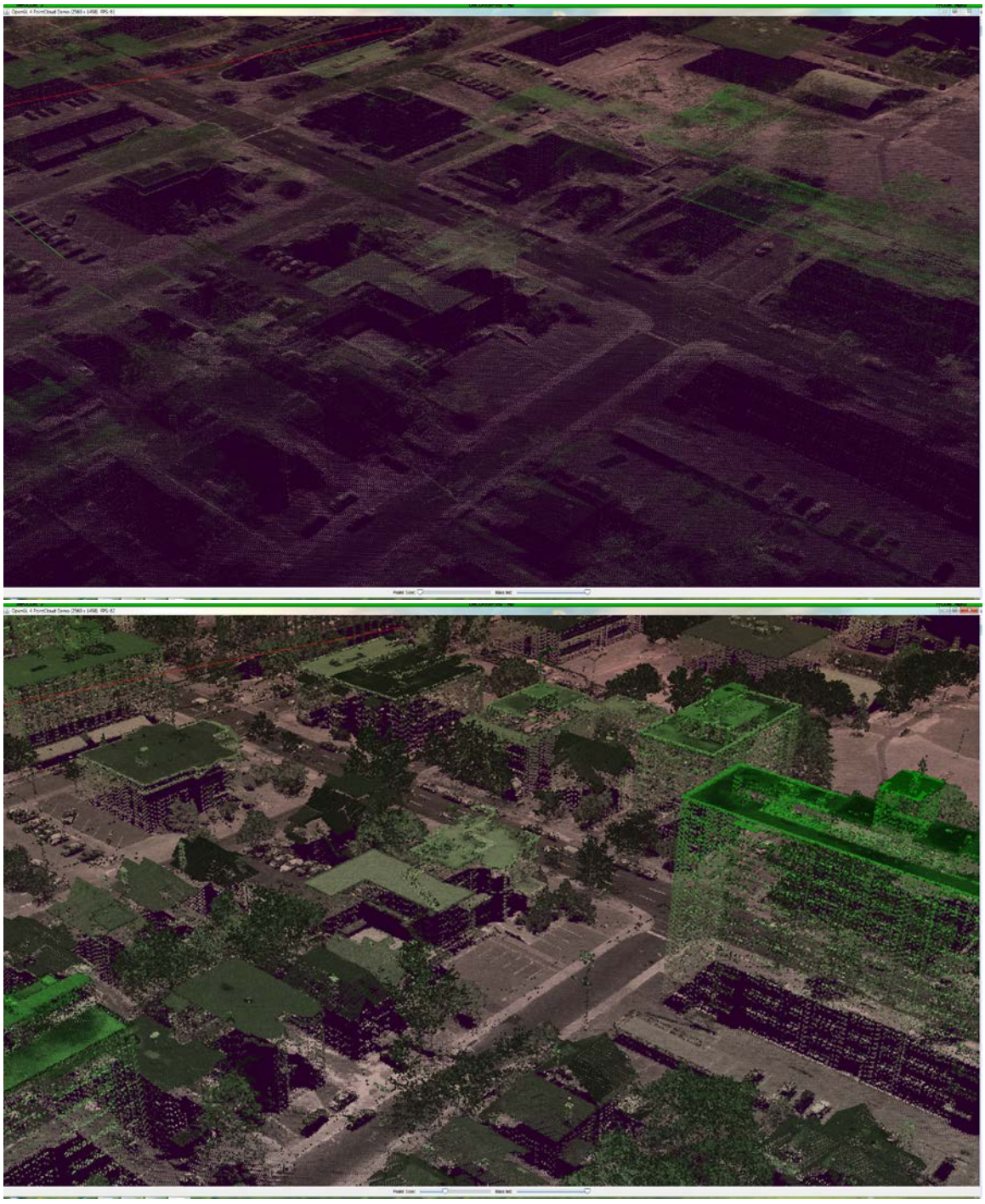

Figure 14 - Point Cloud without splatting (above) and with splatting (below)

Ultimately this effect can be quite useful to quickly gain a better grasp at the structure of the point cloud should the point cloud be undersampled for the display resolution and view location. Splatting is useful, but can easily hide smaller details as the splats tend to make objects appear messy when the splats overlap. The Java client allowed for on-the-fly adjustment of the splat size since optimal splat size varies over the scene and determining it automatically is an open research area. This made it easy for the user to adjust the points making the scene more solid, raising it high to really fill in the scene, or turn the effect off completely; all with just a quick slider adjustment.

3.5.1.1. Rendering Speed

Splatting points has a significant impact on the performance and results in fewer points able to be rendered at interactive rates. While single pixel points are extremely efficient at being drawn as they are the simplest objects possible to render and require less hardware operations to complete, rendering a point size larger than "1" requires additional application commands and graphics hardware.

The first stage impacted is the rasterizer. The rasterizer is the hardware that determines which pixels are to be altered when drawing any primitive to the screen. When a point has more than a size of "1," the rasterizer identifies all the extra pixels that will be used to draw the larger sized point. These pixels are emitted to the next stage of the graphics pipeline, and are referred to as fragments. For not anti-aliased renders, a single fragment is emitted per pixel, where multiple fragments are emitted during anti-aliased renders; the exact number of fragments per pixel is determined by the anti-alias type and sample number specified by the driver or selected by the application developer.

The second stage impacted is the depth test. Normally a single point would generate a single pixel and emit a single fragment (or a small number of fragments for anti-aliased renders). During splatting, every point generates a many fragments per point. All these extra fragments have to be processed by the Z-buffer culling and have a fragment shader operation run on them to finalize their final color and depth. The Z-buffer check is a comparison with the current state of the depth buffer to check if the fragment currently in question is in front or behind what has already been completed for the next frame of the scene. If it is behind the current depth value, the fragment is immediately discarded, but if it is in front then the fragment continues to be "shaded" by the fragment shader. The fragment shader is a small bit of code which executes directly on the graphics card and accepts many parameters from the rasterizer and outputs a single final color for that fragment. Fragment shaders often accept many parameters to create complex and powerful results. Fragment shader operations can be trivial or extremely expensive depending on the shader code defined to render the point and are often the limiting factor in scene interactivity. However, fragment shader operations in the examples for this project were not expensive, with the added depth test operations having more of an impact on performance than the actual shading calculations.

All these operations, regardless of what stage they are directly processing, put extra strain on the video RAM. While the overhead of the extra operations per point can be fairly well quantified, the effects of the video RAM cannot. Fragments competing within the same pixel can cause stalls in the pixel pipeline; and are affected by point clustering, overlap, and draw order. This results in the graphics pipeline being stalled more or less often based highly on scene details that change per frame and per dataset.

It is the sum of all these burdens which bound the number of points that can be drawn while maintain an interactive framerate.

3.5.2. Convex Hull Removal

The purpose of the convex hull algorithm is to detect points that would be occluded if the surface was solid and use this information to effectively hide the points on the back facing surfaces. Sagi Katz, Ayellet Tal and Ronen Basri introduced this concept in a publication title, “Direct Visibility of Point Sets”. The convex hull point removal algorithm works by transforming the points and then taking a convex hull operation on that transformed set. The points identified from this convex hull are then matched with their original untransformed points to identify the points in view. The following example is provided to show the stages of this algorithm which will be explained in 2D for clarity.

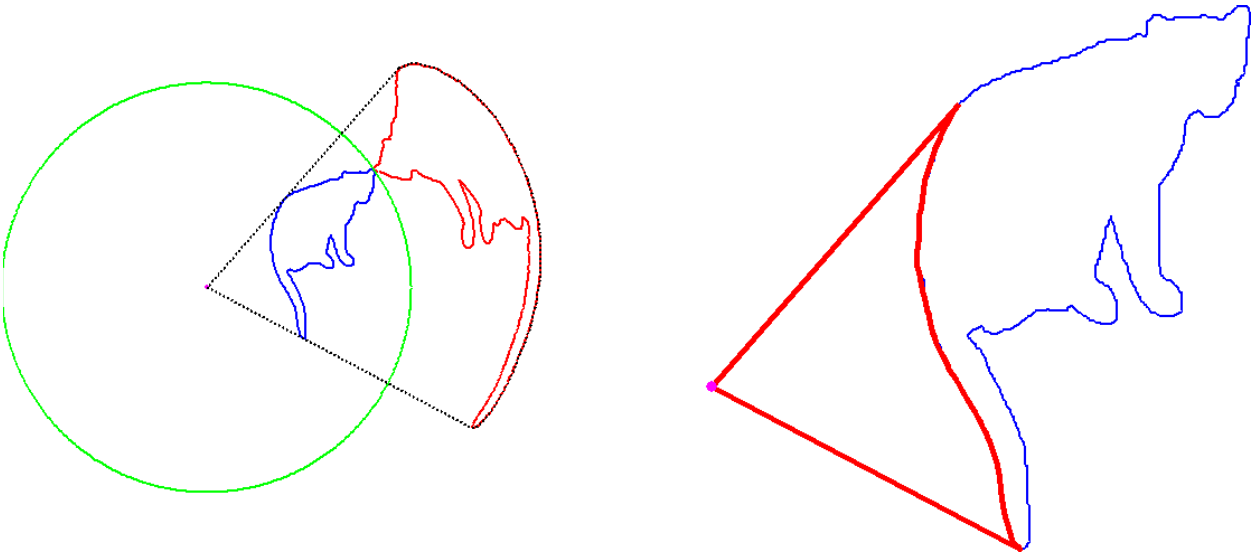

Figure 15 - Hidden point removal process diagrams

Figure 15 (left) shows the stages of the algorithm in three unique colors: purple is the center point of the viewpoint of the scene, green is a circle that encompasses the entire scene, blue for the points within the dataset, and red for the points after transformation.

The first step to the algorithm is to find an encompassing circle (or sphere for 3D) that fully engulfs the object, shown as the green circle. The points of the object are mirrored with respect to the circle. Simple vector math is done to apply this transformation, where the difference between the center and sample point is added to the sphere radius and that resulting magnitude is applied to the direction vector of the center point to the sample point. This essentially flips the object over the sphere and distorts the object as seen by the red points in Figure 15 (left). The next step will determine the points in view by performing a convex hull calculation over the transformed point cloud (the red) which also includes the center viewpoint (purple). This step is computationally expensive and will result in the set represented by the black outline. This set (minus the center point) are the points considered to be in the view, and all other points can be eliminated or deemphasized in the render to better show the resulting object. Figure 15 (right) shows the untransformed version of this convex hull operation.

Notice it appears as if the only included points are the ones that should be the visible points given the current perspective from the purple point. The object is then rendered, with the additional data of which points are in view allowing for the point cloud to be better represented.

While the results are impressive, there is an issue in implementing this for global point cloud datasets. It has to do with the spherical flip transformation process. The encompassing sphere diameter is a variable in the construction of the transformed point cloud. Its size alters the amount of points that are considered "visible" by affecting the positional relationship the transformed points have to one another. The smaller the encompassing sphere, the fewer points are included, the larger the sphere the more points are included. This is a byproduct of the transformation. The hidden point operator retains the points which are included in the convex hull calculation and the sphere size alters the number of points that will be included in the convex hull. There is no single sphere size that will result in a perfect transformation that guarantees that all hidden points are not drawn and that all visible points are drawn. Rather, the size of the encompassing sphere is some-what subjective and can result in having too many or too few points. Because the sphere size is completely variable, it is possible to have the set of visible point underrepresented, overrepresented or visibly just right.

Adjusting the size of the sphere drastically changes the number of included points. Katz et al. describes a heuristic process of choosing a correct sphere size. It begins with finding the centroid of the point cloud an establishing an opposing view point along the axis from the view point to the centroid, equidistant to the centroid. The hidden and visible points are then compared from these two view-points, choosing the sphere size that includes the most points overall, but the least amount of common visible points between the two view-points. The idea is that for the two opposing view-points they shouldn't share many common visible points, as points on one side should be the back for the other, and vice versa.

However, their method is not appropriate for real-time rendering of a point cloud as it requires the hidden point operator to be run many times to eventually find an optimal sphere size to use for a given view. This is unacceptable given how long the operation takes to execute. However, it was found that an encompassing sphere of around 5000 times the radius of the tightly encompassing sphere produced good results in all tests. Applied to large scenes however this may cause precision problems in the transformed points which would lead to unacceptably erroneous results.

3.5.2.1. Speed

While removing hidden points greatly improves the users' ability to understand the point clouds, this task is only interesting in interactive techniques. Since this algorithm is view-dependent, the set of points that is included changes constantly. Meaning that finding the hidden points must be efficient and fast enough to be applied to the points as the view point changes interactively. While no hard framerate exists for stating a system as interactive, the visualization community generally considers 8 to 12 frames per second as a minimum bound.

A straight-forward implementation of this algorithm, and applying it to the Stanford Bunny (Stanford University, 2014), proved to be too slow for interactive use. This model has approximately 35,000

vertices and has a comparatively small amount of points; the number of points in many of the Air Force datasets is in the billions. Even when applied to the 35,000 points, the algorithm was still too slow to provide interactive navigation of the scene. The setup and transformation of the points over the sphere (step 1) was actually very fast, but the convex hull calculation proved to be too computationally intensive. This was expected to be the slowest part of the hidden point remover, with an expected computation time complexity of O(n log n), and timing the performance validated this assumption. Knowing this, the convex hull calculation was not manually implemented; instead a library with a known optimized convex hull algorithm was integrated to guarantee that the most efficient implementation of the hidden point removal algorithm was being tested.

With the Stanford bunny model, the algorithm ran at approximately 2 fps, but was fast enough to evaluate the results. Some straight forward alterations to the interaction made the hidden point remover more usable. First, the hidden point remover was not calculated every frame; it was changed so that only when the view was static were the hidden points found and removed. This caused a little hiccup in interaction as the hidden points were determined. However, this only took a split second, meaning that overall the application remained responsive and useable, but not as aesthetically pleasing as a truly fast implementation of a hidden point remover.

3.5.2.2. Visualization of Hidden Points

A few tests on how to use the result of the hidden points were then performed which used the interaction paradigm where the hidden points were not determined until the view was stable. The first test was the default naïve case where the system drew the visible points and removed those that were occluded. Due to the interaction method when moving the camera, the viewer was only able to see those that were determined to be visible in the previous static view. This causes the object to look very incomplete which could lead to problems with orientation.

To address the orientation issue, it was decided to that while moving all the points would be restored. Then, upon a static view, the hidden points would be removed. This proved to keep coherency while moving the scene much better, however, was slightly distracting to members of the team that used the prototype as points snapped in and out of the view, based on whether the model was being manipulated or not. Also, the hidden points were completely removed from the view. While this seems counterintuitive, they are extra data that could be shown in the scene, and finding a way to represent them, but represent them as not a front facing point, would possibly be desirable.

The next iteration of the hidden points again used the static view calculation paradigm to encourage fast interaction. However, instead of removing those points that were determined to be occluded points, they were drawn in purple. Purple was chosen as it tends to stand out against a black background on LCD monitors, while also being dark like the black background. The dual purpose allows for occluded points to be distinct but still subdued compared to the purposely prominent in-view points, see Figure 16.

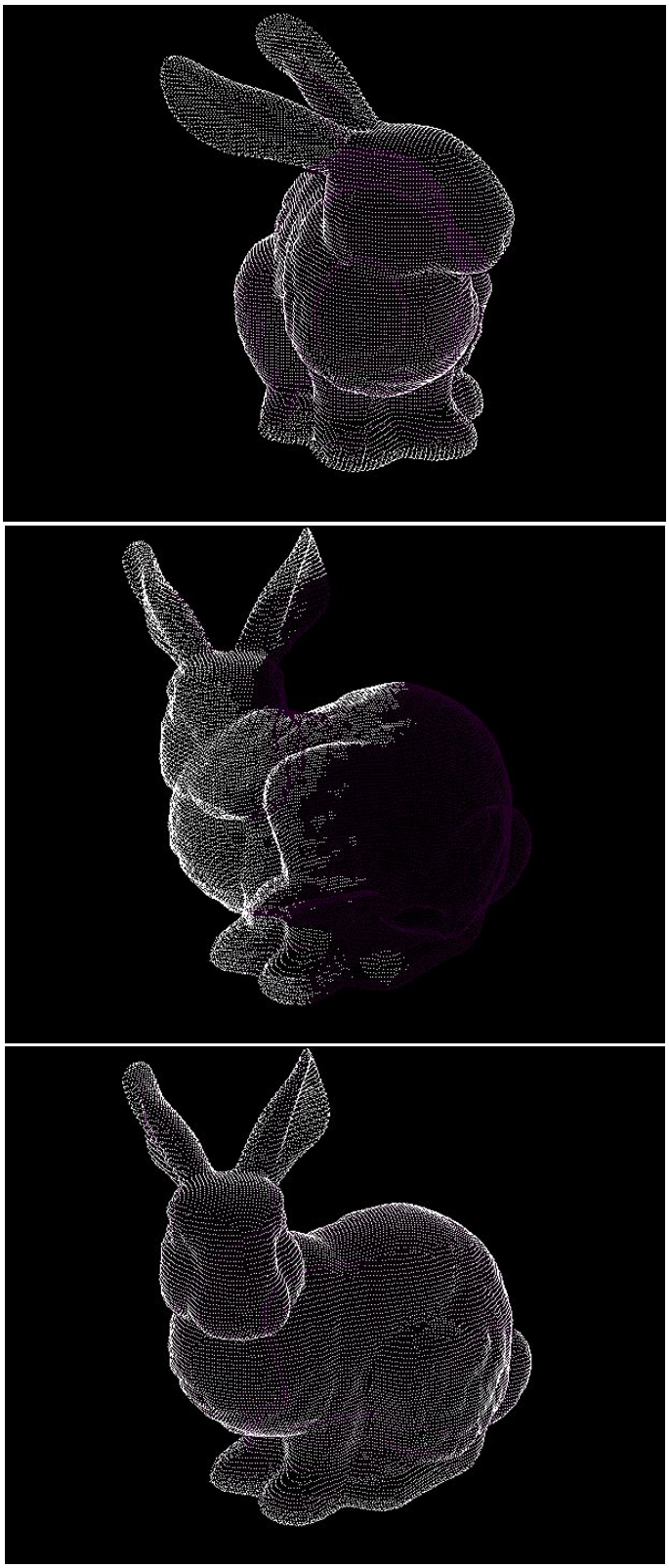

Figure 16 - Stanford bunny model in a static view (top), being oriented into new view retaining its previous view's hidden and non-hidden points (middle), and after new hidden point calculation is complete for the static view (bottom)

3.5.2.3. Failure in LiDAR

The hidden point removal algorithm was tested against LiDAR data with fairly poor results. Katz et al. (Basri, Ayellet, & Katz, 2007) applied this algorithm to typical normal solid (watertight) 3D objects, never to a point cloud scene with non-uniform sampling and sensor based occlusions like what would be present in LiDAR. It was unknown whether or not it could be effectively applied to LiDAR. Outside of the expected long processing time, the results were mixed.

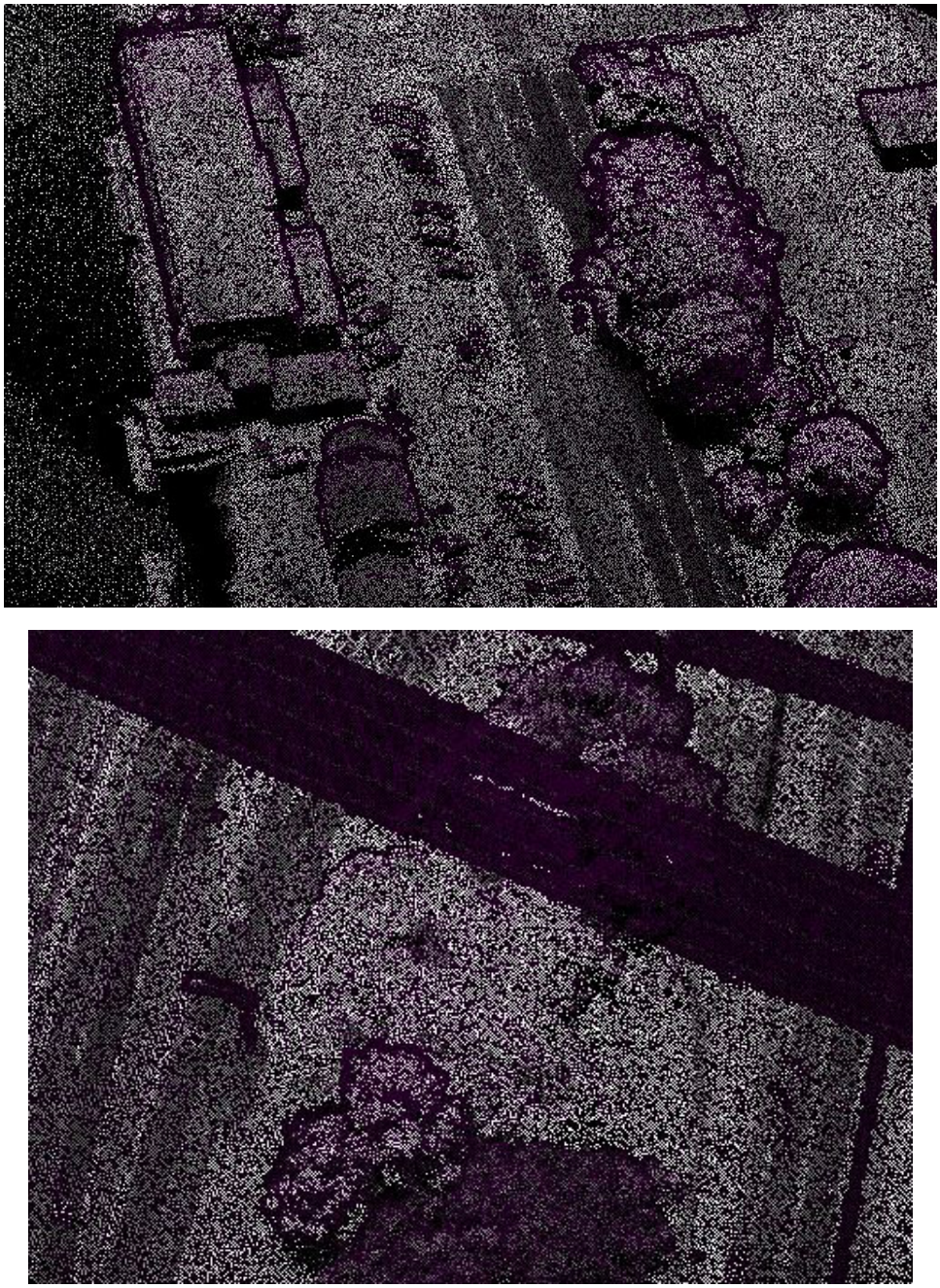

Figure 17 - Hidden point removal applied to LiDAR scans where foreground features show outlining characteristics.

While the LiDAR scans did surprisingly behave well with this algorithm, they did not work as perfectly as the pristine solid point cloud models did. Problems arose in two areas. First, based on a particular view, it is possible to orient the LiDAR scan in such a way that properly identified foreground objects overly omitted points in their outlined areas. This can be seen in the examples in Figure 17 (top), where buildings or trees cause points in their immediate screen space surroundings to be over aggressively marked as hidden. Even more dramatically this effect can be seen in the right image where there are power lines running across a road way, with their surrounding points all being marked as hidden incorrectly. This effect can be lessened by increasing the flipping sphere size of the algorithm (as described above in the algorithm details), but cannot be completely eliminated.

Figure 18 - Hidden point removal applied to LiDAR showing reasonable results but with excessive thinning of the point cloud.

The second problem resulted from LiDAR's inherent slightly noisy nature. As seen in the examples shown in Figure 18, the same point cloud is shown; the top showing all points, the bottom showing the hidden point operator applied. While the hidden point algorithm worked fairly well removing occluded points correctly, especially in the trees toward the back and with the foreground buildings, notice how the entire scene has been thinned of points. Areas that clearly should be visible do not show as much detail due to there being fewer points. This thinning most likely can be attributed to the slightly noisy nature of LiDAR data. These slight variations in point locations for flat or slightly curved surfaces are enough to confuse the hidden point removal algorithm. This happens because it causes some of the points of a surface to be included with the convex hull calculation of the algorithm, while others are not causing a seemingly random subset of points to be chosen as hidden.

Given these issues, application of this hidden point removal algorithm to LiDAR was generally considered a failure. While results were reasonable, they were not good enough for general usage. Not only does the technique suffer from various problems in choosing hidden points, but its runtime performance is also prohibitively slow. With hidden point removal being a view dependent operation, speed is an important aspect. As stated before, the technique was slow even on smaller test models and when applied to LiDAR scans of roughly 300,000 points, the operation took roughly a minute to complete (results varied based on view). Because of these problems a better solution will need to be found.

3.5.3. Simple Depth Function Removal

Realizing that the hidden point removal algorithm was too costly to be implemented in real time, simpler methods that focused on performance were tested. The first utilized a simple depth buffer technique combined with splatting (a common term for drawing point clouds with larger than one pixel points).

The point cloud was rendered in two passes for this method. The first pass drew the points to a back buffer with splatted points. The size of the splats was adjustable by the user during execution to test various sizes of splats. This pass was done to simply generate a depth field of the point cloud being rendered, but the splats gave some padding to the area around each point. The second pass takes this depth buffer and uses it when rendering the normal point cloud. However, as the points are drawn their depths are compared with the first pass' depth buffer allowing for the point to be discarded if farther back than a specified threshold.

What this allows for is to simulate a connected surface among the points, in order to block out points that are considered in the back, and should be occluded. This happens because of the splatting applied to the points in the first pass. If the splats are large enough in comparison to the density of the point cloud then they will form an approximate surface along where the actual surface of the object would have been. In essence, the surface represented by the depth buffer is then used to test the second pass' draw, where the points can be evaluated to be in front of or behind the approximate surface formed by the splats.

This method is perfect for running in real time, as the overhead cost is low. It does require the points to be rendered twice, but any fragment shader operations can be instructed to not output color for the first pass, limiting calculations to only point transformations. The overhead on the second pass is simply a check of the depth value versus the first pass' depth buffer and a determination if the point should be discarded or not. This operation is done entirely in hardware and highly optimized by graphic card vendors.

While this method is efficient in rendering, and did produce good results, it is very dependent on the point cloud density, view, and parameters set within the implementation. There are two adjustable parameters that are closely related to the structure of the point cloud and how well this method will work. The first is the splat size used in the first pass. Should this parameter be set too small, the resulting depth buffer that is forwarded onto the second render pass will essentially have holes in it and possibly allow for points that should have been occluded to briefly show through as the model is inspected. However, adjusting this too large will result in a shadow being cast on areas of the point cloud that are close in screen space but distant in actual depth. This can be seen in Figure 19. Notice areas around the top of the bunny's head, where the base of the ears have their points removed. This is due to the points at the top of the head being closer to the viewer and having a splat size that is large enough to create a zone where points significantly farther back are occluded.

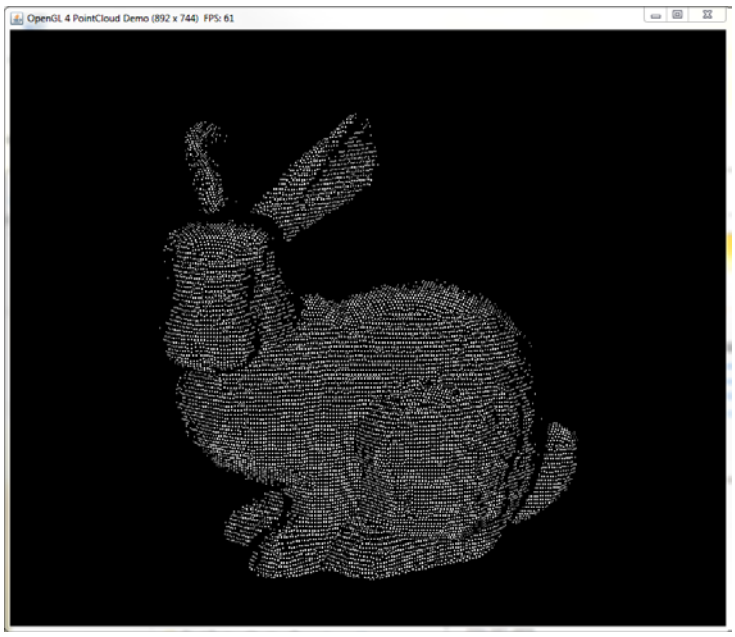

Figure 19 - Two pass rendering with depth buffer occlusion

The second parameter is the evaluation function offset value. After the first pass runs and the comparison depth buffer is generated, the second pass doesn't have to directly compare the depth values of the rendered points to the depth buffer. Instead, it should have some sort of offset value from which the comparisons are being made. The reason for this is illustrated in Figure 20. It shows a notional surface in orange where the points that were sampled from this surface are in black. With just the normal splats, shown in green (which are aligned to the viewer), other points from to the left would

normally be occluded that should be shown. However, if we offset the depth distance that we test against then we are moving the green lines back to the red lines, where the red sweeps in the picture are now the areas that would be found to be occluded and points omitted. An example point is shown in the diagram that would have been occluded without the offset and is correctly shown with including the offset. This point is behind the original green line (the splat of the neighboring point), but falls in front of the offset depth test represented by the red lines.

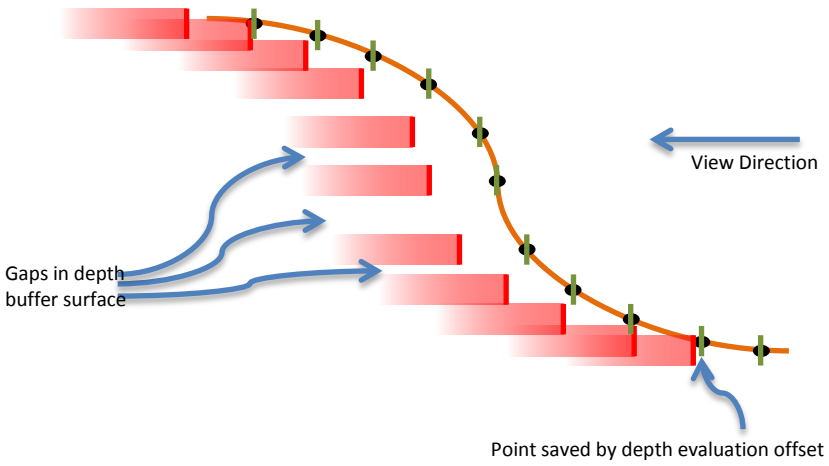

Figure 20 - Cross-section view of depth tested point removal

Overall, this method was not tested with LiDAR due mostly to its naivety. If the parameters were adjusted just right then the point cloud would be presented with reasonable results. However, this required perfect tuning, and with the goal of interactive point clouds, this method would quickly deteriorate once something in the visualization was changed. Recent work has started to address the same issues and should be researched in future implementations. This work would include the use of an erosion function. The erosion function would work directly on the depth values produced by the first pass before being utilized by the second pass. It will reduce the shadowing problem along with smooth out the depth values making the offset more effective.

3.5.4. Proposed Methods with Triangulation

The other methods previously described show various promising features in accuracy or speed, but none have both, which is required for interactive use of a hidden point removal algorithm. It has been proposed that the use of triangulation methods might actually create a good hidden point removal algorithm.

Triangulation of point clouds for objects reconstruction is a very difficult problem and goes beyond the scope of this is project. While the methods of triangulation and object reconstruction are still being widely researched and developed, the results most of the time can be considered to be fairly inaccurate when compared to the original object. However, the proposed use of the triangulated point cloud does not lie in direct rendering of the reconstructed mesh. Instead it is proposed to use this data during the render of a point cloud to remove hidden points.

This would be done by first preprocessing the data and creating a triangulation reconstruction of the point cloud. The accuracy of the reconstruction doesn't matter as much as the general shape of the reconstruction. This reconstruction then is used in a two-pass render, where the first render is of the reconstruction to a back buffer, just as the splatting method described above. From there, the depth buffer of that first pass is forwarded to the second pass where the point cloud is drawn and compared against the depth buffer of the first pass. What this will achieve is the crude triangulation will be providing a close enough approximation of the surfaces of the point cloud's real life counterparts, reintroducing the concept of self-occlusion back into the objects within the scene and therefore removing the correct occluded points.

This method has been proposed to colleagues that are working on reconstruction methods, and while the initial impression of actual use of a reconstruction was met with resistance, it was agreed that most reconstructions would be "good enough" for use as a hidden point detector in this approach. The one known downside to this method is the large amount of preprocessing required to create the triangulation of a dense point cloud. However, once made it can be reused many more times until the underlying point cloud changes.

3.6. Point Clouds on the Web

Implementation and testing of point cloud visualization on the web was an eye opening experience. In some respects the results were surprisingly good, in others they seemed very far behind what is expected in modern application development. To have a similar application for exploring point clouds on the web, access to hardware graphics acceleration is required within the client along with a way to access desired point clouds. The answer to graphics acceleration within a webpage was easily WebGL.

3.6.1. WebGL vs OpenGL

OpenGL is a computer graphics API used in most professional PC graphics applications and, to a smaller extent, the gaming industry. OpenGL is an industry standard API that is widely used, well understood, and is constantly trading the "state-of-the-art" title with its closest competitor, DirectX. OpenGL is an open standard and is widely compatible with all major operating systems, such as Microsoft Windows, Apple OSX, and almost all Linux variants. Most languages have native implementations or wrappers so it is also multi-language compatible. There is even a variant, OpenGL ES, that is is used by all modern phones and tablets to provide hardware acceleration in the mobile space.

WebGL is a JavaScript language binding to OpenGL ES, allowing for hardware accelerated graphics in a web browser. Although there are a couple of small differences, WebGL allows individuals to bring their OpenGL knowledge to the web. Even with all the opportunities that hardware acceleration brings to the browser, adoption of WebGL has been relatively slow. This is simply because its implementation in web browsers. Gaining access to graphics hardware is not something that was originally allowed within the security model of web browsers. Compatibility had to be directly implemented into each browser, and even now there are slight inconsistences in rendering between various browsers. A team at Google has been the driving force behind WebGL, making Google's own Chrome browser the initially capable browser and normally the most compatible and compliant with the specification.

While tests showed similar graphics performance between OpenGL and WebGL, there are differences between the two API's that can cause major differences in the approach to programming a visualization. Most of the more advanced features of OpenGL are not exposed through WebGL. These mainly reside in complex buffer operations that aren't exposed in the mobile version OpenGL ES. The biggest omission from WebGL though is the lack of Vertex Array Objects (VAO's). VAO's allow rendering state information to be stored and recalled in one simple instruction instead of the more error prone numerous instructions required by previous versions of OpenGL. While this doesn't introduce any specific limitations, it does require extra work from the programmer and it has a small performance hit to the graphics processor as the drawing state isn't as efficiently switched.

3.6.2. JavaScript Limitations

Because WebGL has direct access to the same hardware that traditional desktop OpenGL does, performance is surprisingly comparable. Looking solely at the graphics portion of applications, there is a small insignificant difference between OpenGL and WebGL in terms of performance. However, what has been a constant hindrance to performance in WebGL applications has been its use of the JavaScript language. While widely used throughout the web, JavaScript is meant to be extremely flexible to the programmer and not focused on performance. This limitation is really a by-product by the evolution of the web from its original focus of disseminating information into a platform supporting a huge number of tasks to include running full applications.

There are many aspects of JavaScript that make it slower than other programming languages, but there are mainly two that seem to be its greatest hindrance. First, there is no statically defined object typing within the language. Traditional programming languages have identifiers associated with the defined variables so that the computer knows what kind of data this variable will be holding. These identifiers can be such things as "float" or "String." JavaScript is not like this; instead every variable is labeled with "var." What this means is that variables can be cast to any sort of data within the program. However, while this flexibility can be very useful to the programmer, it can become impossible for the computer to understand the types of data residing in the variables until execution. Once the computer figures out what kind of data is in the variable, the computer can then proceed to handle these various types of data and continue execution of the program. There is actually a subset of JavaScript being developed to allow for static typing of variables, removing the need for runtime checks, but is supported only in a limited number of browsers. This toolset is called asm.js. Asm.js is developed by Mozilla, where their browser Firefox takes advantage of it the most. Firefox implements a JavaScript engine that compiles and then executes the compiled version of the asm.js code achieving execution speeds much closer to natively run code. Chrome also makes use of this more efficient JavaScript standard, just without ahead of time compilation. This language subset can be run on all browsers, but some of the performance advantages gained are lost as certain browsers aren't designed to take advantage of all the efficiencies added by asm.js. With the need for cross browser compatibility this was not a solution that could be leveraged, as programming with the speed increases of asm.js in mind would result in a very poorly performing application for those users using an unsupported browser.

The other performance hindrance is that JavaScript is an interpreted language. This essentially means that JavaScript is not running in the native language of the computer. Instead the computer is looking at

the JavaScript code and reading the instructions to then execute the program. This is not only more inefficient, but performance is also highly dependent on the browser. This is because many browsers implement their own JavaScript engine that is actually interpreting the JavaScript loaded from webpages. While traditional computer languages are written then compiled into native machine code to be run directly on a processor, JavaScript's interpreted execution creates a huge performance gap when compared to natively running code.

The difference in performance is hard to quantify because there is no real way to make a direct comparison. Different browsers along with different tasks will produce wildly varying results. However, performance will always be slower in JavaScript due to the interpreted nature of JavaScript and the overhead of running within a browser. While JavaScript creates a much more restrictive computing environment, it is not impossible to implement point cloud visualizations on the web. It just imposes a far greater limitation than that of native code applications.

3.7. WebGL Client

A web-based client prototype was constructed using WebGL. This client's purpose was strictly to show the potential for loading and viewing point cloud data within a web page. Even with the prototype's limited functionality, it proved that performant 3D rendering is possible within a webpage. The WebGL client implements the LiDAR renderer with its various underlying layers of point coloring. The point clouds were rendered the same way as the desktop client, and produced visually equivalent results.

While the resultant point rendering was consistent with the desktop application, other features within the WebGL client were implemented to test alternatives for data fetching and processing within a web-based client. This test proved that a WebGL client could make a request for data, manipulate the received data, then make a brand new request, unloading the current data and receiving new data to process. Additionally, a demo was setup to show that data could be handled in a streaming matter, as loading an entire dataset would quickly become cumbersome and sluggish. The demo took a scene, breaking it into smaller chunks and loaded those chunks independently, slowly building out the scene instead of hanging until the entire data request was loaded. During the loading the scene was fully interactive popping up loaded sections as they were received. This functionality could be extended in a more intelligent matter with the server backend, allowing for rough sections of the point cloud to be loaded first giving the user a general idea of the area being viewed while subsequent received packets would fill in the detail of the area being viewed.

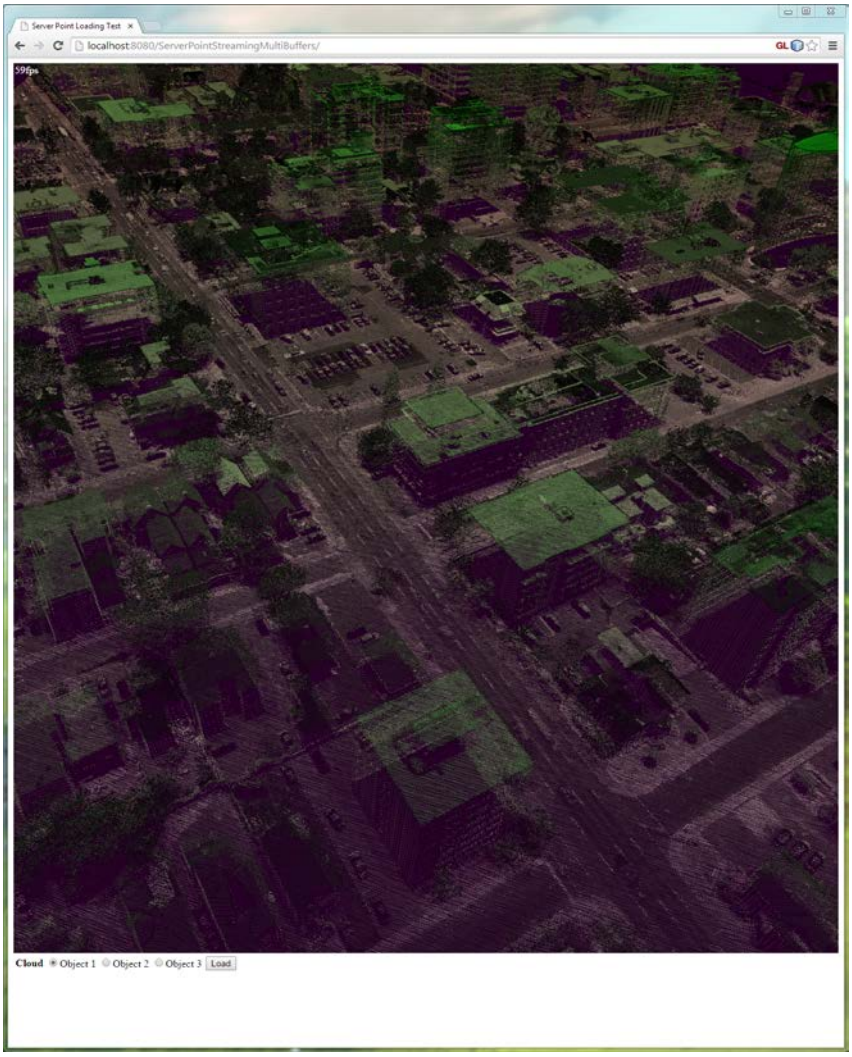

Figure 21 - WebGL Client Prototype

Tests were conducted that targeted the rendering, not the general processing, portion of the visualization. Large point cloud sets were rendered with simple camera controls and a framerate counter. Under these situations comparable performance was observed among desktop and web based tests. Obviously there were some differences in rendering given the different platforms, but overall rendering performance was within a similar, not requiring a vastly different approach to rendering as GPU performance was so similar. Quantitatively measuring performance was difficult and proved to not be entirely fruitful, however, some general "rules of thumb" were observed.

The first observation was that the buffer size for loading data onto the graphics card was much more constricted within WebGL. The desktop version essentially had no limit to the buffer size, even when working with a buffer larger than graphics memory which relied on the graphics card drivers to handle dispatching data to the card. While the desktop version would have its frame rate plummet due to memory swaps with main RAM, it would still function. However, in the WebGL implementation, the buffer size was much more constrained and proved to only allow for hundreds of megabytes of data

versus the desktop versus the available graphics memory for this test of 2GB. It appears that no data is actually transferred when allocating a buffer that was too large which resulted in the application referencing an empty buffer when rendering. There was no data available to render so no points were drawn. However, this limitation could be overcome as it is still possible to create multiple buffers and render a larger scene from these smaller buffer allocations. This of course requires a little more effort on the part of the programmer, and is slightly more expensive in processing for the client since multiple draw calls need to be issued, but overall the performance difference is negligible.

Second, there is a vast difference between JavaScript and other compiled languages in terms of performance. Even simple tasks such as preprocessing the data to collect point cloud statistics such as largest or smallest value within the dataset took significantly longer in comparison to the desktop counterpart. This meant that any approach taken on the web based client needed to be very light weight in terms of CPU demand. This resulted in more advanced hidden point removal algorithms to not be implemented for the web client. Implementations were also hindered by the fact that there aren't as many tools available for heavy calculations within a JavaScript environment.

Lastly we tested the difference between transferring data between the server and client as either JavaScript Object Notation (JSON) or as a binary stream. No statistically significant performance difference was observed between the JSON objects and binary data transfers, which resulted in the application to use the simpler and easier to work with, JSON string objects.

3.7.1. JSON

JavaScript Object Notation (JSON) is a useful way of transferring sets of data between networked systems using the JavaScript language. It is always possible to transfer RAW bits of data, but doing so requires the development of a protocol along with an encoder and decoder. JSON allows for sets of data to easily be transferred by packaging the data in labeled pieces and then accessing that data on receiving end without any sort of complex file parser.

JSON includes a binary transfer mode, BSON (Binary JSON), to allow for easier sending and receiving of messages along with accessing the data to begin a parser. However, there was no perceived benefit from using the binary mode of JSON transfers which may have been a by-product of our fast internal gigabit networks. For small datasets there would be no way to truly determine if the binary version of JSON transfers would have been faster as the amount of data would have been too small, and other resident network traffic would have played a huge role in determining the faster transfer. Even among large datasets that took a couple of seconds to load, there was no significant observable change using the binary form. Because of the simpler nature of string JSON objects, binary versions were not used, simpler string versions were used for these tests.

It is worth noting that when transferring over the internet, there may be a much larger penalty in using string JSON objects instead of binary versions simply due to the fact that bandwidth and latency over an internet connection will be much worse than a local 1Gb connection.

3.7.2. Websockets and Glassfish Server

A Glassfish Webserver handled communications between the web page client and the LiDAR server. A Glassfish server is a web-based Java Enterprise Edition (Java EE) application server allowing for processing and computation on the server to assist in execution of a web-based application. The LiDAR GRID Server was written so that a simple websocket connection could be used to connect and gather information about a desired region of point cloud data. The server allowed for inquiry on properties of the hosted dataset and then returned points within the requested region and in the specified format. The Glassfish server made it easy to then extend this functionality into a web page as there are great API's already available for Java that allows websocket communication. The communication to the web page was then extended in the same fashion with a websocket to forward the information to the client.

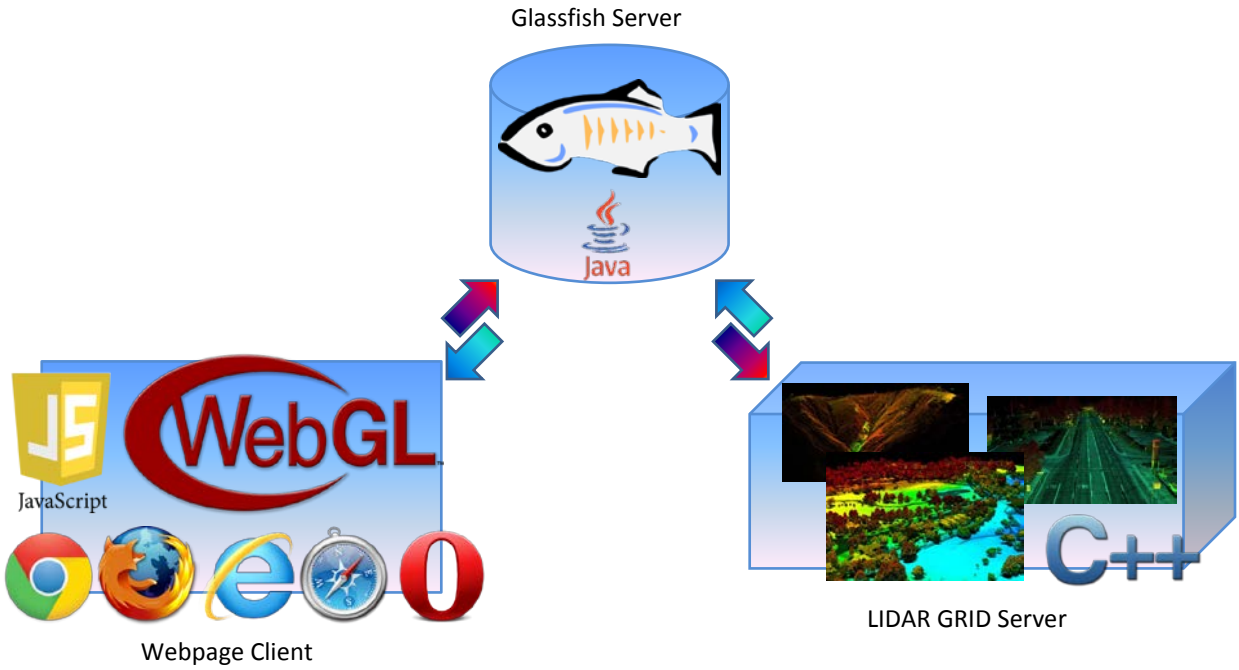

Figure 22 - Information transactions between WebGL client to backend point cloud server

A websocket is a protocol that allows for communication between two computers over a TCP/IP connection. It allows for full duplex communication, enabling the client to make requests and be given a response, but also the server to push data to a connected client without being triggered by a client. This allows for a more robust connection as there truly is a link established between the client and the server allowing for any sort of required communication to happen. This is stark contrast with standard web communication protocols that are strictly client initiated and transiently connected. This allows for the possibility of the server to send various data or possibly updated data to the client without the client having to constantly check to see if there is anything new available.

The Glassfish server allows for a full application to be running on the server, capable of processing anything a normal Java application would process. This allowed direct use of code from the Java desktop client. The Glassfish server then acted as a middle man for the WebGL client. It accepted connections from WebGL clients and serviced requests for areas that they wanted to display. The

Glassfish server would then interpret that request and talk directly with the point cloud server to gather the requested points. Those points would be processed and packaged by the point cloud server and be passed back to the Glassfish server where they would then be transmitted over JSON to the WebGL client, completing the request.

3.7.3. Browser Limitations

Testing among browsers was a must. Development of any web application, no matter how small, requires testing on multiple browsers due to their slight, but application breaking, differences. While normal computer programming has target platforms such as Windows, Linux, or Mac OSX, web programming targets browsers as their platforms. While the differences aren't as dramatic between platforms like Windows or Mac, browsers do have a large number of small differences that are more loosely defined. It is these small hard to detect differences that make web development a challenge when looking for maximum browser compatibility. Functions that are seemingly supported by two different browsers might result in completely different behavior.

The need for hardware graphics acceleration through WebGL initially limited the browser selection to Chrome, Firefox and Opera. However, during execution of the project OSX Mavericks was released for the Macintosh, which brought with it a new version of the Safari Internet browser that was compatible with WebGL. Additionally, during this time Internet Explorer 11 was released which also brought with it WebGL compatibility. That concluded compatibility with all major web browsers for WebGL.

However, it was quickly discovered that being able to run WebGL content was far from being bug-free and achieving parity with Chrome, the premiere client for WebGL. While Firefox and Safari had little differences in the ways of receiving user input, they functioned essentially the same as Chrome during any testing. However, Internet Explorer displayed major differences, even with simple things such as drawing a line, where the line would not be cleanly rendered, but would show odd blending attributes on its edges. Ultimately for this project differences in rendering were not too much of a concern since almost all of the rendering was just points, which in reality is something very easy for the browser's to get correct.

3.7.3.1. JQuery

Differences in reading user input were much harder to manage as each browser handled input slightly different than all the rest. This initially was difficult to handle trying to find code combinations that worked in some browsers and didn't break in others. Quickly a solution was found for this that is widely used among web-development, and that was JQuery. JQuery is a JavaScript addition to your web page that allows for a common interface to input functions such as detecting a mouse click or a wheel scroll.

JQuery works by having an interface to all major browsers' functions. These functions are then commonly mapped to a single interface. This interface then allows for the web page to be written to just use this one interface and JQuery internally worries about compatibility with each major browser. Browser compatibility perfectly matches the target platforms for this project with compatibility with Internet Explorer, Firefox, Chrome, Opera and Safari.

Usage of JQuery is exceedingly simple, as just by including it within your web page source and then including a reference to the file will allow access to each of its inputs methods. JQuery is used among many popular sites on the internet and so it is highly tested and compatible. It works without much overhead, so there isn't any perceivable performance decrease with its use. Additionally, it is exceedingly small, being only 83 KB in size. Since the amount of data being transferred for a point cloud viewer will be considerable, 83 KB is basically insignificant. Custom code would probably be much smaller, but the stability of JQuery along with its robust feature set makes it a much smarter option and less prone to any possible errors.

4.0 Conclusions

Point cloud visualization is highly dependent on the data available to the visualization client. The primary source of point cloud data for visualizations of interest to the Air Force is for LiDAR collections. These datasets have a wide range of possible categories of data per point, but for most collections only position, altitude and intensity data are available. To create a fairly generic LiDAR visualization client the focus was on these limited categories. The result was a visualization that highlighted the structures within LiDAR data.

There were other possible visualizations that could enhance the viewing of LiDAR data. Some of these are techniques that need to be improved, such as the hidden point removal techniques as they were too slow for real time use. Other techniques would need access to more types of data categories within LiDAR scans that aren't as widely available. Even more would benefit from the mingling of LiDAR data with other types of data to create a more robust visualization. Overall, point cloud data visualization can be improved beyond even what this effort achieved, but in many cases those techniques would not be useful as the data required for that technique would not be available.

Most of these techniques were easily transferred from a traditional visualization client written in Java to a web platform based on WebGL. The visualization quality for both platforms were comparable, as advanced techniques were not used on the desktop application so it was directly portable to WebGL. Speed was even very similar. Techniques that were supported on both desktop and WebGL platforms had similar performance as they were not CPU demanding, and both have fairly efficient access to the underlying graphics hardware. Those techniques that were not portable to WebGL had their own unique issues even on desktop platforms, so the general approach would need revising to have it functional on any platform.

Finally, the setup achieved for the WebGL client was a fairly efficient design that was flexible and easy to maintain. The generic LiDAR point cloud server was easily accessed by Java with its websocket API. The WebGL client was able to leverage this by using a Glassfish server backend which uses Java as its underlying language. This meant that very similar code was able to be used for the Java client and the Glassfish server.

What resulted from this task were highly useful visualization styles for displaying LiDAR point clouds, along with verification that point cloud visualization can be effectively achieved on a web based platform. Compromises are to be expected, but a computer with reasonable hardware specifications accessing a web based platform can expect a similar experience to desktop counterparts. While further experimentation in visualizing point clouds could result in better depiction, the results from this first effort greatly expanded the capabilities and understanding of how LiDAR visualization should be done in the future.

5.0 References

Basri, R., Ayellet, T., & Katz, S. (2007, August). Direct Visibility of Point Sets. *ACM Transactions on Graphics, SIGGRAPH 2007, 26*(3).

Datasystems, C. (2007, August 10). *GameStudio A6 / A7*. Retrieved 2014, from ModDB.com: http://www.moddb.com/engines/3d-game-studio/images/ssao-a7

Everitt, C. (2001, May 15). *Interactive Order-Independent Transparency.* Retrieved 2013, from nVidia Developer: https://developer.nvidia.com/content/interactive-order-independent-transparency

NOAA. (2013, 06 04). *What is Lidar?* Retrieved 02 19, 2015, from NOAA National Oceanic and Atmospheric Administration: http://www.webcitation.org/6H82i1Gfx

Stanford University. (2014, Aug 19). *The Stanford 3D Scanning Repository*. Retrieved Nov 19, 2014, from Standford University: http://graphics.stanford.edu/data/3Dscanrep/