Visualization of a Large Set of High-Resolution Geo-referenced Images

Adam Kropp
Advanced Undergraduate Project
Prof. Seth Teller, Advisor
Laboratory for Computer Science, Graphics Lab
May 7, 1999

Abstract

The goal of this project is to develop a set of visualization tools that use existing image and 3D data from the City Scanning Project to display information in novel and interesting ways. Two of the tools developed deal with viewing image information gathered from one common optical center, allowing the user the freedom to pan, tilt, and zoom from the optical center. These tools bring the data to life and help identify some problems with the data collection and processing techniques. The final tool developed attempts to reconstruct a realistic 3D view of the world from a point not represented in the captured data by combining images taken from many locations and a rough 3D reconstruction of structures in the captured region.

1. Background

The City Scanning Project has built a large suite of hardware and software tools to acquire and process high-resolution geo-referenced digital imagery of the world with the goal of constructing a three-dimensional model of the captured surroundings. The processing stages are not yet mature; they currently assume the world contains only flat vertical façades, and they require some human intervention in the matching of certain features in the environment. Nevertheless, the output, a texture-mapped OpenInventor model, is a fairly convincing reconstruction of the sampled area.

Both the acquisition and processing steps produce a large amount of information that is not limited in usefulness to geometry creation. The most basic unit of information is the image, which combines a digital image (1524x1012 pixels, 24 bit color) with the camera parameters, location, and direction in a common coordinate system for all images. The acquisition hardware (named Argus) captures a hemispherical tiling of images around a single optical center (a Node). The current data set uses 47 images per node when on the ground, and 71 image nodes when taken from a rooftop. Although each image contains its own location information and is registered with its neighboring images to correct for measurement errors, this logical grouping into nodes allows for inter-node registration to be accomplished by registering and adjusting an entire node rather than each individual image. It also allows for conveniently creating spherical texture maps of the world from the node's optical center for visualization purposes. Argus was moved around Technology Square to acquire many nodes (81 total) with a view of each building so that the geometry reconstruction algorithms would have enough information to proceed.

So far in the City Scanning Project, the image data has been the means to the end of generating a three-dimensional model, but the over 16 GB of data collected is useful in other applications as well. The raw data can be used in image-based rendering techniques, and can even be combined with the generated model for some novel viewing paradigms.

2. Overview

There were two major goals driving this project: developing methods to qualitatively assess the quality of the acquired and calculated data, and finding uses for the data other than that which it was originally intended. The latter is more interesting than the former, so

although they will both be covered in some detail in the following sections, more time will be spent discussing efforts towards that goal.

The first of these goals consists of relatively straightforward additions to the existing tools, such as an overlay of the reconstructed data on the original images. This simple tool allows the user to quickly verify registration between the reconstruction and features such as buildings, giving important insight into possible causes of a bad reconstruction. Another tool allows viewing of any single node from its center drawing from the original image data. Super resolution nodes (nodes having large numbers of total pixels) can be viewed with this tool with an arbitrary field of view. The user can see a large portion of a room or zoom in to read a sign on the wall or examine a portion of the tiling that did not get properly aligned.

The second goal is less defined than the first. Two projects were completed towards this goal: a super-resolution high-dynamic range single node viewer; and an off-node pseudo-image-based rendering view. The former is an extension of the simple high-resolution viewer that reads actual radiance data rather than simple image data to expand the dynamic range of the images that can be viewed. This allows detail to be discerned in the darkest shadow of a room, while the shapes of bright, sunlit clouds can still be seen through a window (plates 5, 6). Through a dynamic mapping done in real-time, the view transfer function (effectively the sensitivity of the camera to light) can be changed and the results seen instantly. The off-node viewer will be described in the greatest detail because it addresses a large number of different problems in trying to achieve its goal: building a viewer which reconstructs the most realistic view possible from any location, not necessarily at a node.

3. Single Node Viewing

A single node contains the information necessary to recreate a view of the world from one point in space and time. By combining all of the images in a hemispherical tiling around the viewpoint, it is possible to trace the color of any ray of light coming to the optical center, effectively allowing one to "look around" with an appropriately made viewer. Such an application replaces the simple photograph by something that captures a *place* at one instant, rather than a view at one instant. Since information viewed from a single node need not have any notion of the surrounding geometry, both indoor and outdoor scenes can be captured and viewed in this manner, as is demonstrated in plates 1 through 4.

WIRE

The first of the tools, named **wire**, was developed for this project to deal with single node viewing of a wireframe/image composite. The goal of this tool is to allow easy visualization of mis-registrations between the original image data and the final geometry, and between multiple nodes' views of the same location. Such information is useful in evaluating the registration and reconstruction algorithms, and in identifying possible areas that need to be improved.

To facilitate hardware acceleration of the view from one node, **wire** uses a low-resolution cylindrical texture map, which is a combination of all of the original images from a particular node. This texture is mapped onto the appropriate locations on a spherical mesh located around the optical center of the node, so that a view from its center is equivalent to a picture taken from the same location and direction. While allowing very high frame rates for interactive use, this approach eliminates a great deal of the detail available to the original images. Fine details tend to be blurred out due to the relatively low resolution of the texture map (1024x512) and the hardware blending performed on textures; however this is not important for the observation of rough details such as building edges.

Once the spherical node is drawn (with the depth buffer disabled), a wireframe version of the reconstructed building data is drawn from the same viewpoint and perspective. If the reconstruction is correct, the wireframe should coincide with building edges exactly (plate 1). **wire** can load multiple nodes at once and quickly switch between them so that comparisons of registration can be made between different nodes.

A practical application of this presented itself while testing out the program. Most of the buildings in the reconstruction matched up fairly well with views from the nodes; however one building seemed to be about ten feet taller in the reconstruction than it looked in the images. Not only did this allow the problem to be identified, but by examining all of the nodes that had a view of that building, a possible cause was identified. Several outlying nodes which had views of the building and therefore contributed to its reconstruction seem to match more closely with the reconstructed height, and are not aligned with the rest of the nodes – a result which suggests that the geometry is based on the worst sample points. This indicates that some sort of noise reduction technique is needed when combining the information from many nodes while reconstructing geometry to improve the results.

RAYCAST

The second tool developed for this project, named **raycast**, is a high quality single node viewer that samples pixels from the original images in real time using a modified form of raycasting. Although this cannot update the view at frame rates comparable to a hardware rendering approach without greatly sacrificing image quality, it has several advantages to that method.

First, every sample in the view originates from one or several of the source images. This guarantees that the node may be viewed in the same resolution as the original images were taken, allowing much finer detail to be observed. The cylindrical texture map used in **wire** contains 2048x1024 pixels, while a normal node contains 46 images, each with 1524x1012 pixels. Thus **raycast** is able to take advantage of 34 times the data that a hardware approach would use for such a node, and in a super-resolution node, that amount can be 200 times or more. This becomes apparent in (plates 3,4) where all of lobby 7 can be viewed, and then the user can zoom in to a sign and read it. Details as small as 0.25cm at 10m can be resolved in this node.

The other main advantage is that per-pixel operations may be applied on the fly. This allows for such effects as blending between multiple images to obtain a final value, seamlessly integrating low resolution and high resolution images (or detail images) in the node view to use extra memory only in areas that need it, and adjusting the viewer's visual response function for viewing high dynamic range images in near-real-time (without pre-processing the images).

The viewer in its simplest form takes a user-controlled view direction and field of view and rasterizes it on the screen. For each pixel on the screen, a ray is calculated from the optical center of the node to the corresponding point on the viewport, and then is extended out into space. The raycaster then must determine the color of that pixel based on its direction. This involves determining which image or images that ray intersects, calculating the point on each image in image coordinates where the ray intersects it, and finally performing a weighted average of the color from each image based on the distance the ray is from the center of the image (only if there is more than one image). This entire calculation is enclosed in nested for loops that scan through each line and each pixel within that line.

To speed the calculation of the ray, an incremental approach is used. First, the ray to the lower-left corner of the viewport is determined (corresponding to index (0,0) on the

screen). A ray (**start**) is constructed to the point $\left(-\tan(\frac{fov}{2}),1,-aspect\times\tan(\frac{fov}{2})\right)$, as if the viewer were looking down the y-axis; it is then rotated about the X and Z axes to align it with the true view direction. Next two delta vectors, **dx** and **dz**, are constructed to increment the starting vector in the x and z directions (which correspond to x and y on the screen). The **dx** vector is $\left(2\times\tan(\frac{fov}{2})\Big/width,0,0\right)$, and the **dz** vector is $\left(0,0,2\times aspect\times\tan(\frac{fov}{2})\Big/width\right)$. These, too, are rotated about the X and Z axes so they are aligned with the true viewport. At the start of each line, the view vector is set to **start** + **dz***j (where j is the vertical index). It is then incremented after each pixel calculation by adding **dx**.

Next, the image or images that the ray intersects must be determined. The naive approach to this problem is to simply project the ray on to the plane in which each image is located, then check to see if the point of intersection is within the borders of the image. This is reasonable with a small number of images, but as the number of images in a node is increased it becomes too time-consuming. Since this is an operation that must be performed once per pixel, it must be as fast as possible to allow real-time navigation by the user. To speed this search, a preprocessing stage is done which subdivides space in a tree structure, and keeps pointers to the images that intersect each region. The tree begins with an octahedron consisting of triangles with vertices of the form $(\pm1,\pm1,\pm1)$. A ray can be placed into one of these regions using three comparisons with zero.

During the preprocessing stage, an ImageBranch object is instantiated for each of these triangles. The ImageBranch object will keep track of any images that fall within its triangular region, and can optionally be subdivided into smaller regions. The preprocessor passes all of the images to each of the eight ImageBranches. If a branch contains the image, it is added to its internal list. Once the number of images reaches some internal threshold, the ImageBranch subdivides into four smaller equally-sized triangles (see figure 1), and passes all of its stored images to those branches. The resulting structure can be searched with O(log n) region comparisons instead of O(n), giving a large reduction in computation. Many of the values needed for the ray-triangle intersections used in this search can be pre-computed, giving an additional time savings. The structure is built such that the ray is

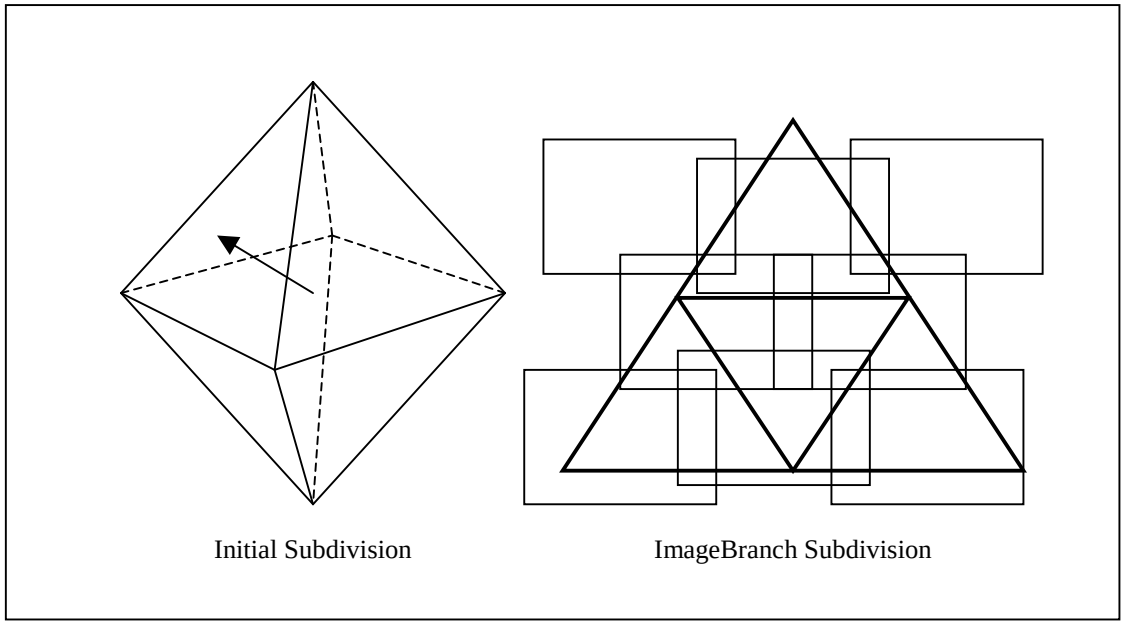

Figure 1: Subdivision of space for finding ray-image intersection

passed to the top level, which determines which of eight branches to search. The ray is passed to that branch, which determines if it contains the ray, and if it does, either searches its (short) list of images for intersections, or passes the ray to its four children. The children repeat this process until image intersections are found, or none are found. A list of images that are intersected by the ray is returned to the calling code.

Now that the images that are intersected by the ray are found, the pixels can be sampled in each image. Each image has a transformation matrix associated with it to specify its projection from the origin of the node. The ray is projected this matrix, resulting in a point (x, y, z) that can be used to index into the actual image data as (x/z, y/z). The color value from each image is weighted by its distance from the center of the image, and all of the values are added and divided by the total weight, i.e.:

$$R = \frac{\sum_i r[i] \times w[i]}{\sum_i w[i]}, \; G = \frac{\sum_i g[i] \times w[i]}{\sum_i w[i]}, \; B = \frac{\sum_i b[i] \times w[i]}{\sum_i w[i]}$$

where w[i] is the weight associated with the pixel from image i.

OPTIMIZATIONS

Three major additions were made to this scheme to optimize performance. First, the pixel weights will always be fixed for a given index into an image, so by creating an array of weights the same size as an image, all of the weights can be pre-calculated. The appropriate

weight can be accessed by the same index as is used to read the color value from the image. This approach increases speed at the cost of additional memory usage, an acceptable tradeoff when responsiveness is of primary concern. If all of the images are guaranteed to be of the same size, a single set of weights can be calculated at the beginning of time for use with all images.

The second improvement comes from noting that, in general, adjacent pixels will come from the same image. Additionally, the transformation to image space is expensive, so an incremental method of finding the next pixel is desirable. Both of these are addressed during the sampling stage. When the ray transformation is performed, and additional ray pointing to the next pixel is also transformed. Since this is effectively a perspective transformation, distances are not preserved, so a simple delta between pixels will not yield proper results. However, the z value is recorded as well, and by storing the delta for x, y, and z, the viewer can increment across the image. Using this increment value in each image, a count is determined for the number of samples left before an edge of the image is reached. The top-level loop can then simply increment each pixel location and sample each image until the count reaches zero. At that point, it must again find the images that need to be sampled. This reduces the number of ray-image intersection searches from once to pixel to once per image edge intersected per scan line. Since each image typically spans between an eighth and a fourth of the viewing area (or more if zoomed in), a full search will have to be performed only a few times per scan line.

The final optimization made to maximize speed is to split the raycasting task among

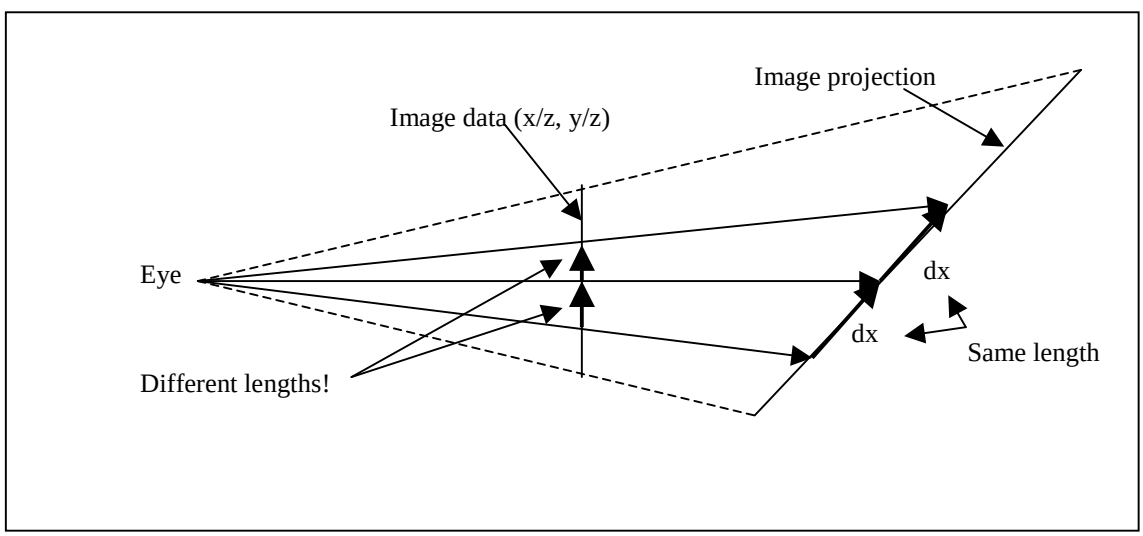

Figure 2: Example of the use of the delta vector for sampling.

more than one processor on a multiprocessor machine using pthreads. Two main structures are used to facilitate this. First, a RaycastInfo structure is created for each scan line on the screen. This structure encapsulates all of the information needed to calculate one line of the image, including the ray pointing to the beginning of the line, the **dx** vector, a pointer to the line of the output image, and various other useful pointers. Next, it is added to a thread-safe queue. All of the worker threads wait on a semaphore contained in the queue until a line is available for them to process. Once a line is available, one of the threads removes it from the queue and processes the line. When done, it waits on the queue until another line is available. This produces a nearly linear improvement in frame rate based on the number of additional processors.

HIGH DYNAMIC RANGE

The raycaster described above is capable of using an arbitrary number of images of arbitrary size, but is limited in its reproduction of a node because of a physical limitation of the camera – digital cameras have a very limited dynamic range. The location of this range can be adjusted by changing the exposure time on the camera, but a single image cannot capture detail in both shadows and very bright areas. A trial node was captured at multiple exposure settings, and the results were combined to produce a radiance map that reflects the true brightness of every pixel. This map was then compressed using a logarithmic scale into 24 bit pixels, but retaining the full dynamic range. Along with each radiance image is an information file that specifies the minimum and maximum radiance values, so it can be converted back to true radiance values at a later time.

In adding radiance map support to **raycast**, one of the goals was to be able to adjust the displayed range and visual response function on the fly. This precludes mapping each image with the appropriate visual response function each time the user changes it because of the amount of time it would take to accomplish. Also, because of the use of logarithms in both the decoding and visual response functions, it would be prohibitively slow to convert each pixel as it is sampled. The solution was to keep the images in their logarithmically compressed form, and generate a 256-entry lookup table for each image which converted each channel (r, g, b) to the appropriate value. These lookup tables must be re-generated each time the user changes the response function or dynamic range, but they contain only

256 entries instead of over a million pixels. In the decode stage, a simple table lookup is performed per channel per pixel, which has a minimal performance hit.

As can be seen in plates 5 and 6, details in the shadows of lobby 7 are visible, while at the next moment the clouds outside the windows are visible, though it is a bright sunny day. The image can be viewed with a compressed dynamic range so that all values are visible at once, or the range can be expanded to bring out detail in one particular segment of the range (at the expense of saturating regions that are brighter or darker than the target region).

4. Multiple Node Viewing

The single-node viewers described above strive to accurately display the collected data in a useful way, enhancing what can be seen with just the raw images. With the multiple node viewer, the goal was to use information from many nodes, as well as the reconstructed data, to approximate what the world would look like off of a captured node. An additional goal was to make this fast enough to be done in a real-time walk-through of the world. The basic concept used to accomplish this is to determine which nodes should contribute to each pixel in the viewer, and what that contribution should be. Since no actual information has been captured between nodes, it uses information from many nodes to approximate what would be seen from the eye point.

BASIC METHOD

In order to determine the color of a pixel from a particular ray when the eye is at the optical center of a node, only the direction of the ray (and thus the direction of the point being viewed) must be known. However, if the eye point is moved away from the optical center, the actual origin of the ray must be known (figure 3). It can then be traced back to the node to determine the proper color based on the direction from the *node* to the point. Once this is determined, the framework for raycasting a single sample of a single node described above can be used to determine the color.

It is normally a hard problem to determine the distance an object is from the viewer from a set of images; however, for this set of data, that problem has already been solved. The reconstructed data gives a good approximation of all of the large objects in the area being viewed. Although not all objects are covered, and the reconstruction data is not perfect, slight deviations only cause slight distortions in the images. The largest problem

comes from undetected occluders close to the node (such as trees), but this is only a minor annoyance.

In trying to achieve the goal of real-time viewing, a pure raycasting solution to this problem would be difficult, since a large amount of time is spent calculating the color contribution from the nodes after the point being viewed is determined. A hybrid hardware rendering/raycasting solution was developed to speed up this portion of the calculation. This approach uses the graphics hardware to compute visibility and ray-object intersections quickly, then uses this information to complete the raycasting. The algorithm is basically as follows:

1. Set up the viewing transformations in OpenGL
2. Render the reconstructed data using OpenGL
3. Read back the color and depth buffers
4. For every location where the depth isn't 1 (infinity), determine the location in world coordinates of that point
5. Trace the point back to each node which will contribute to the final color, and average the values together

The OpenGL depth buffer contains values between 0 and one, but these are actually mapped from –1 to 1. During the projection transformation, every point gets translated from its world coordinates (x, y, z) to screen coordinates (x', y', z'), where each coordinate is in the range [-1,1]. The x' and y' coordinates map to a location on the screen, while z' maps to the depth buffer. This information can be used to trace back to a pixel's origin. A ray is constructed from the eye to the location of the pixel being traced in viewport coordinates, with its z coordinate set to the depth component of that pixel. The ray is then transformed with the inverse of the GL_PROJECTION matrix and the inverse of the GL_MODELVIEW matrix, resulting in a point **p** in world coordinates.

From this stage, the color of the point **p** needs to be determined by using available information. For each node that will contribute to p (the contributing nodes are pre-determined for a frame, as will be described below), a ray is cast from the node's center in the direction of (**p - node**). This raycasting is accomplished exactly as in the single-node viewer. The colors from each of these nodes are then combined in a weighted average. These weights are determined at the same time as the contributing nodes.

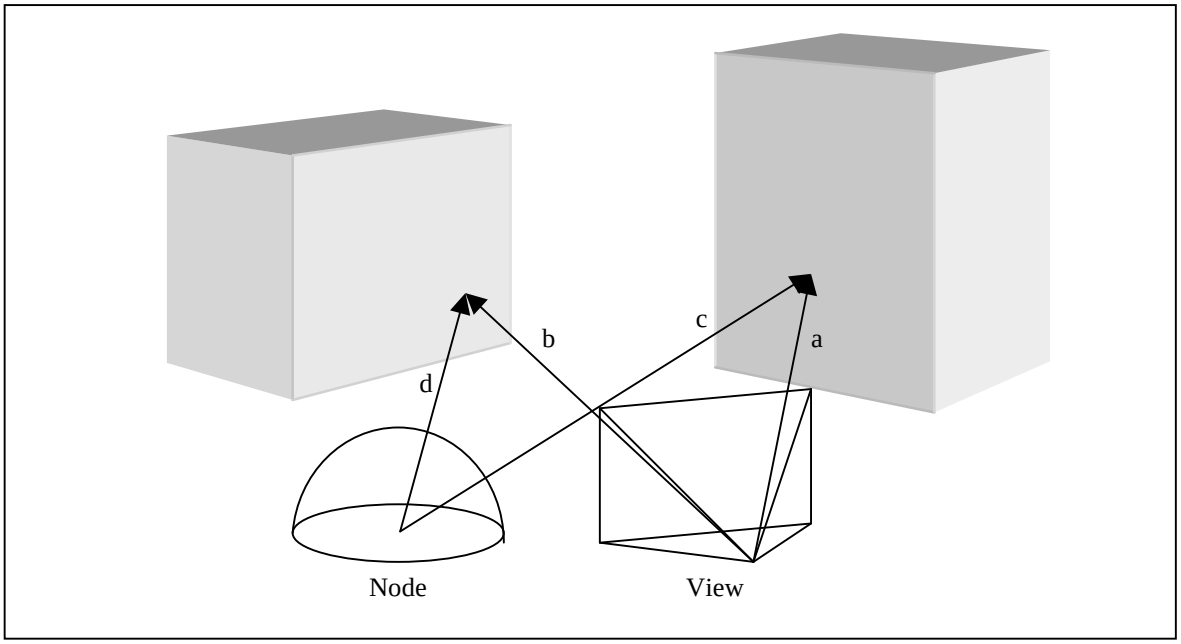

Figure 3: Example of off-node raycasting. Rays a and b are cast from the viewpoint into the scene to color two pixels on the screen. Corresponding rays c and d are cast from the node, and the appropriate color from the node's images is returned to the raycaster to color the view.

Two methods are allowed for determining the contributing nodes in **offnode**. The first method is to simply let the user specify which node to use. In this mode, the user can cycle through each node to see how it projects onto the scene. This turns out to be at least as useful as **wire** in checking both for registration between a node and the geometry, and between two nodes. It is very easy to see misalignments between the image and building edges, and by switching between nodes, the user can spot any movement between the projections of different nodes on the buildings (such as a window moving).

The second method uses a constrained Delaunay triangulation with vertices located at the nodes and constrained segments inserted at the edges of the buildings. The triangulation is calculated at the beginning of time, and stored as a list of Triangle objects which contain pointers to the node at each vertex as well as the position of the vertex. The Triangle object has a method to determine whether the viewpoint falls within its interior and a method to return a relative weighting for the three vertices based on the position of the view. This weighting is calculated using barycentric coordinates, which give a smooth transition between parameters at the vertices of a triangle. Given a triangle **ABC** and a point **p** (figure 4), the weights are calculated as follows:

$$w(A) = \frac{AREA(pBC)}{AREA(ABC)},\ w(B) = \frac{AREA(pCA)}{AREA(ABC)},\ w(C) = \frac{AREA(pAB)}{AREA(ABC)},\ w(A)+w(B)+w(C)=1$$

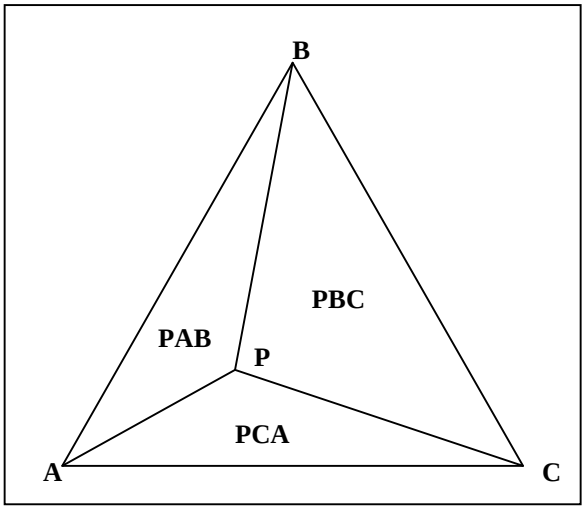

Figure 4: Areas in barycentric coordinates.

With this method, if the triangle the eyepoint is in has a node at each of its vertices, three nodes will contribute to every pixel in the view that falls on a building or ground polygon. If one of the vertices is a corner of a building, then the weight for that vertex will remain at zero, and only the other two nodes will contribute to the view. Likewise, if only one vertex corresponds with a node, than only one node will contribute to the view. This approach to selecting the contributing nodes means that the associated calculations only need to be done once per frame, allowing them to be fairly complex.

REFINEMENTS

The algorithm as described so far works well as long as the buildings being viewed are completely visible to all three contributing nodes. However, when a surface is in view which is occluded in some node's field-of-view, the surface will be colored with the occluder's color value, resulting in strange results. To counter this, the raycaster must determine whether the color from a node is describing the same polygon that the viewer is looking at. This is accomplished by encoding a unique ID for each polygon in its color when it is rendered. After the nodes are loaded but before the user can view the scene, a visibility mask is generated for each node consisting of six square images forming the sides of a unit cube. This mask is generated by rendering the scene from the node's center in each of six directions. During rendering, the color buffer is read to determine the polygon ID of the pixel being rendered. It is then compared with the surface visible in that direction by each node. If a node sees a different polygon in that direction, its weight is reduced to zero so that the other nodes fill in the correct values. This method again takes advantage of the graphics hardware for speed improvement, and pre-computes all of the expensive visibility tests. Although discontinuities in color may be seen on the occlusion boundaries the general coloration will not be disturbed by bad data like it otherwise would (plates 7,8).

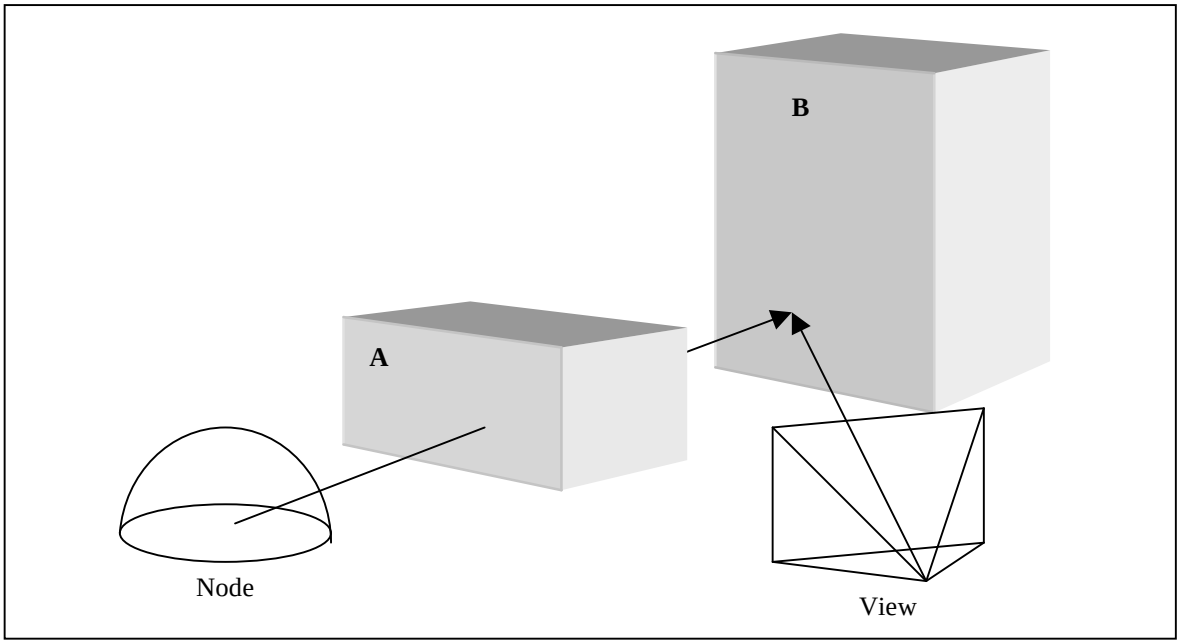

Figure 5: Example of off-node occlusion problem. When the node tries to give a color for a point on building B, it actually is showing a color from building A unless visibility detection is used.

With visibility accounted for, there are situations where none of the contributing nodes has a view of some portion of a surface, in which case the raycaster cannot color that pixel. To remedy this, another pre-calculation is performed which computes the node which has the "best" view of each surface, where best is defined as covering the greatest portion of the surface of all of the nodes, and in the case of a tie, being closest to the surface. The surface coverage calculation is performed again by using the graphics hardware. A view is constructed which originates at the node and makes the polygon being checked as large as possible in the viewport without going beyond the edges. Then the entire scene is rendered in a very small viewport (3x3), and the number of pixels belonging to the polygon and the average depth of those pixels is determined. If the number of pixels is greater than the current candidate node, or it the number of pixels is the same but the depth is less, then the new node is made the candidate node for the surface. Once all nodes have done this process for a polygon, the winner will be determined, and will be stored in an array indexed by the polygon's ID.

If a pixel is being sampled where the total weighting is found to be zero, the raycaster has no information from the designated nodes to color the point with. At this point, it uses the node determined to have the best view of the polygon to fill in the pixel value. Depending on the total number of nodes being used and the layout of the geometry,

this may still yield no sample for the pixel; however, it greatly reduces the chance of leaving a gap in the coloring of a polygon due to occlusion.

OPTIMIZATION

Three optimizations are made to the algorithm above to improve the speed of rendering a frame. The first is to split the screen in to scan lines and dispatch one line at a time to an available thread, just as in the single node viewer. This allows for a 2X or 4X speedup depending on the number of processors available. The second optimization is to perform pixel doubling along a scan line. Every line is still rendered; however within a line only every other pixel is rendered. Each skipped pixel is filled in with the average of the two rendered pixels surrounding it. This causes some image degradation, but if only performed when the user is changing the view, it is barely noticeable; it results in another 2X speedup.

The final optimization reduces the work needed to be done to determine which images a ray intersects from a node. Since this viewer is not scanning linearly across a node like the single node version, it cannot take advantage of any incremental update schemes, so it must determine the source images for each pixel. However, since it is likely that several pixels will still be taken from near each other, and since the same pixels will be used from many different views, it would be advantageous to somehow cache the results of a ray direction search in a node. The caching is performed by a hashtable that hashes on the ray direction, combining the most significant 6 bits from each component of the direction vector into an 18-bit hashcode. This scheme effectively splits the unit sphere into 2^{18} = 262144 contiguous regions (not necessarily all the same size or shape, but similar in size). The hashtable is then filled in lazily in the following manner:

When tracing a ray from a node:

- Check if the ray maps to a valid hash entry
- If so, return the images pointed to by the hash entry
- If not, determine the source images using the search tree
    - Create a new hash entry with the source images
    - Return the images

This method uses a large amount of memory per node; however after some initial settling time when the hashtables are being filled in, it produces results in O(1) time, resulting in a 2X to 3X speedup.

5. Conclusions

Although no specific ends were determined at the beginning of this project, three interesting and very different visualization tools emerged from it. **Wire** took existing rendering techniques for displaying the reconstructed data and node images and combined them in a new and useful way, while serving as a warm-up project for the more complex tools that followed. **Raycast** is probably the most technically challenging of the three as far as designing an efficient algorithm and data structure for the ray-image intersection test. Finally, **offnode** built on the foundation work in **raycast** to provide a truly novel approach to viewing the City Scanning Project data.

Wire and **raycast** are essentially complete as they stand, although raycast could have some additional work done on the visual response function code to make the interface more intuitive. **Offnode**, however, has not reached the goal of providing a truly realistic view of the world from any point. The two major problems which need to be solved to reach this goal are dealing with small occluders (such as trees, cars) and dealing with the sky. Some effort was put into projecting the sky from the nodes being used to color the buildings, but the disparity between sky color and cloud cover even among adjacent nodes makes this solution less than ideal, and buildings "punch out" holes in the sky which are very noticeable. A median algorithm was tested to remove small occluders which appeared on a building from one node but not the others, but results were poor due to the small number of samples used (three). For such a "tree removal" algorithm to work, many more nodes will need to participate in the calculation.

6. Appendix A – Color Plates

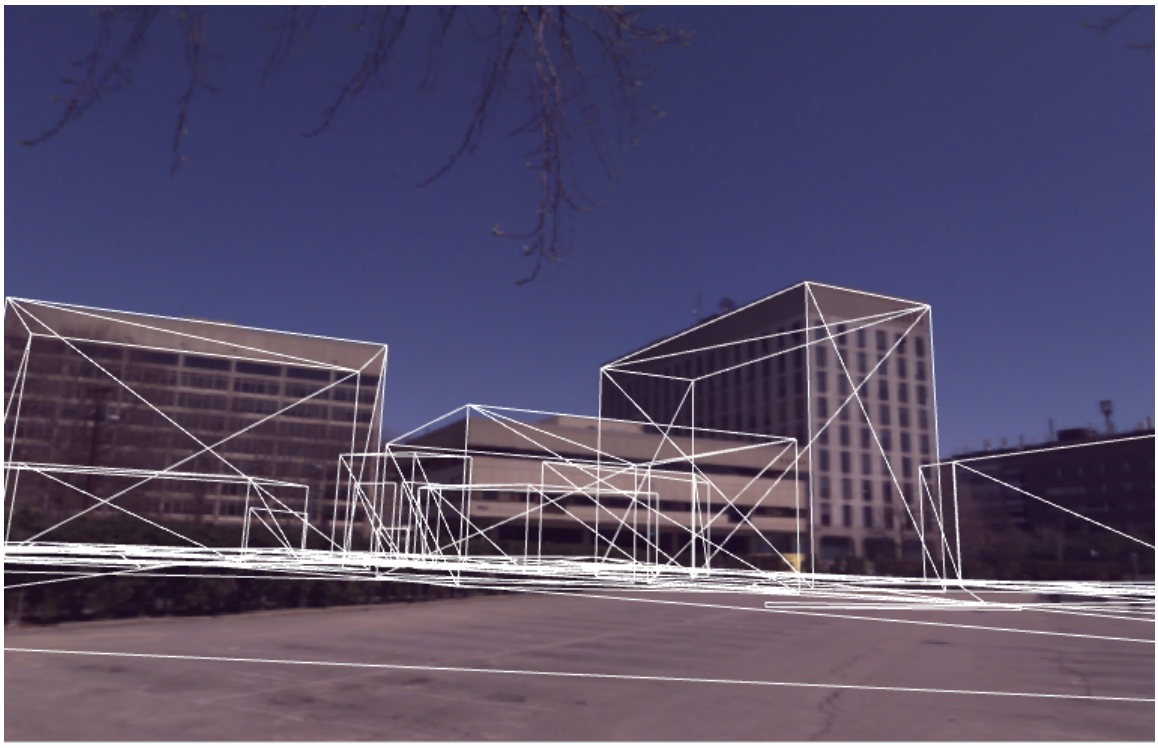

Plate 1: **wire** showing the wireframe aligned with buildings

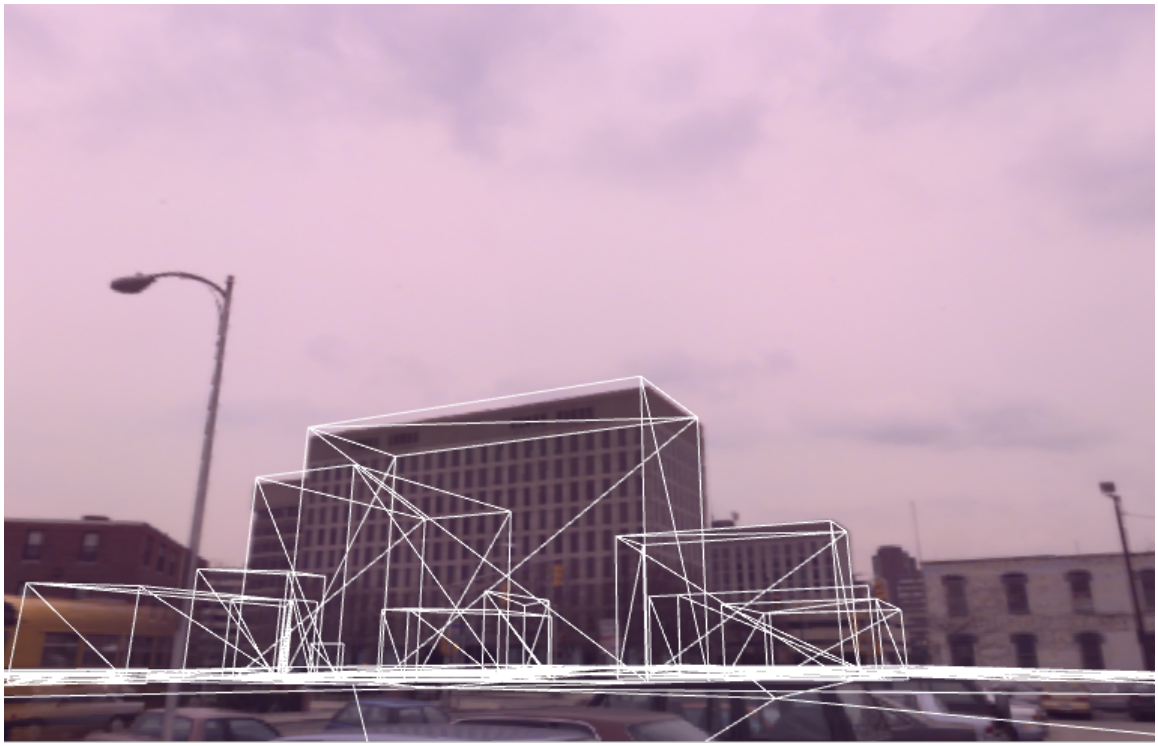

Plate 2: **wire** showing a mis-aligned node

Plate 3: Wide angle **Plate 4:** Close up

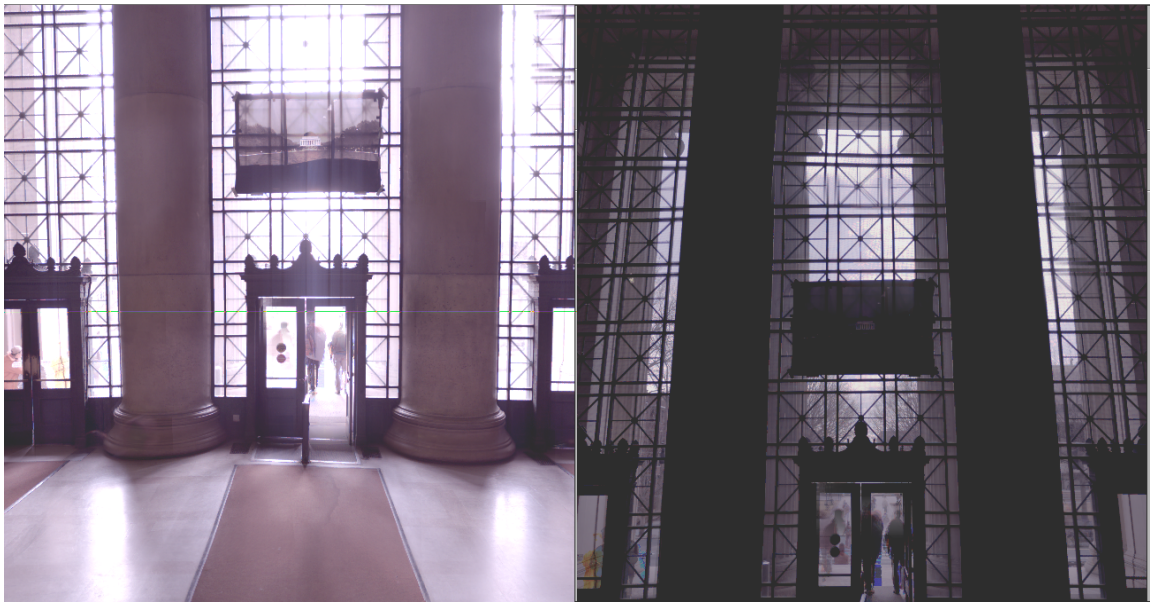

Plate 5: Long exposure -- inside **Plate 6:** Short exposure -- outside

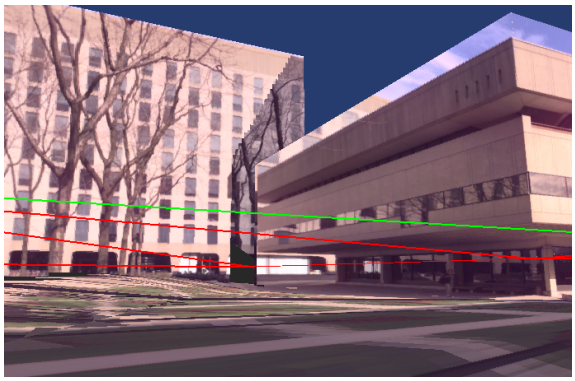

Plate 7: View generated from a single node with fill-in provided by a second node

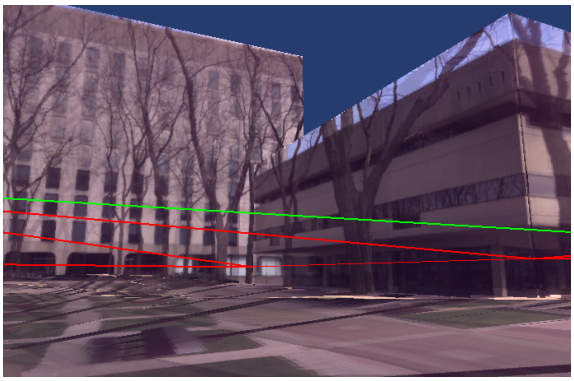

Plate 8: View generated from multiple nodes

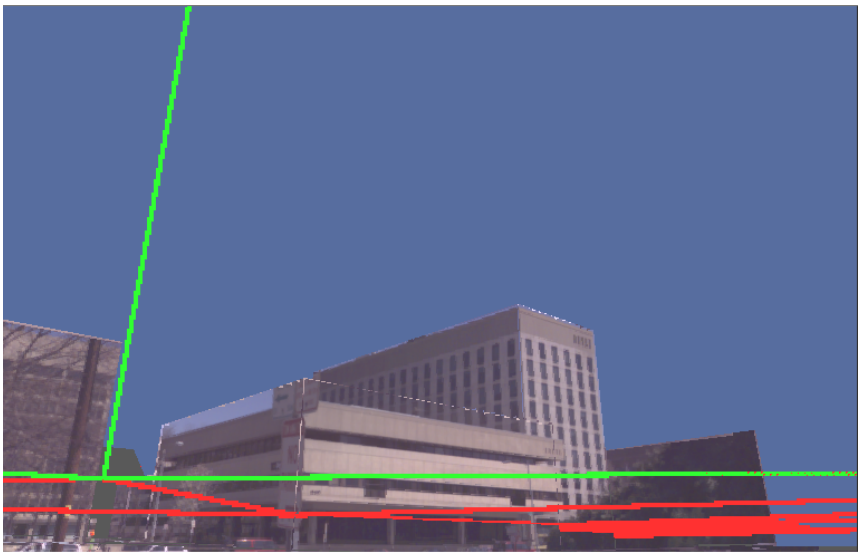

Plate 9: View from a node

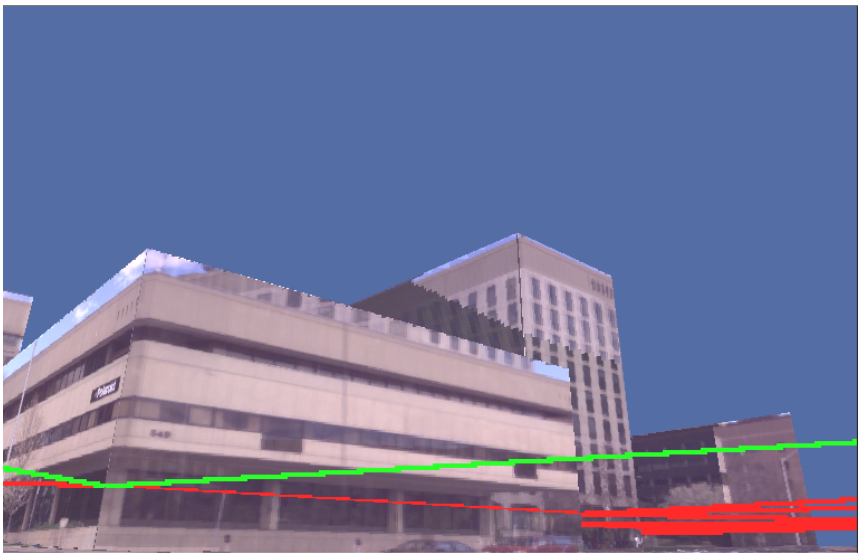

Plate 10: View from between nodes

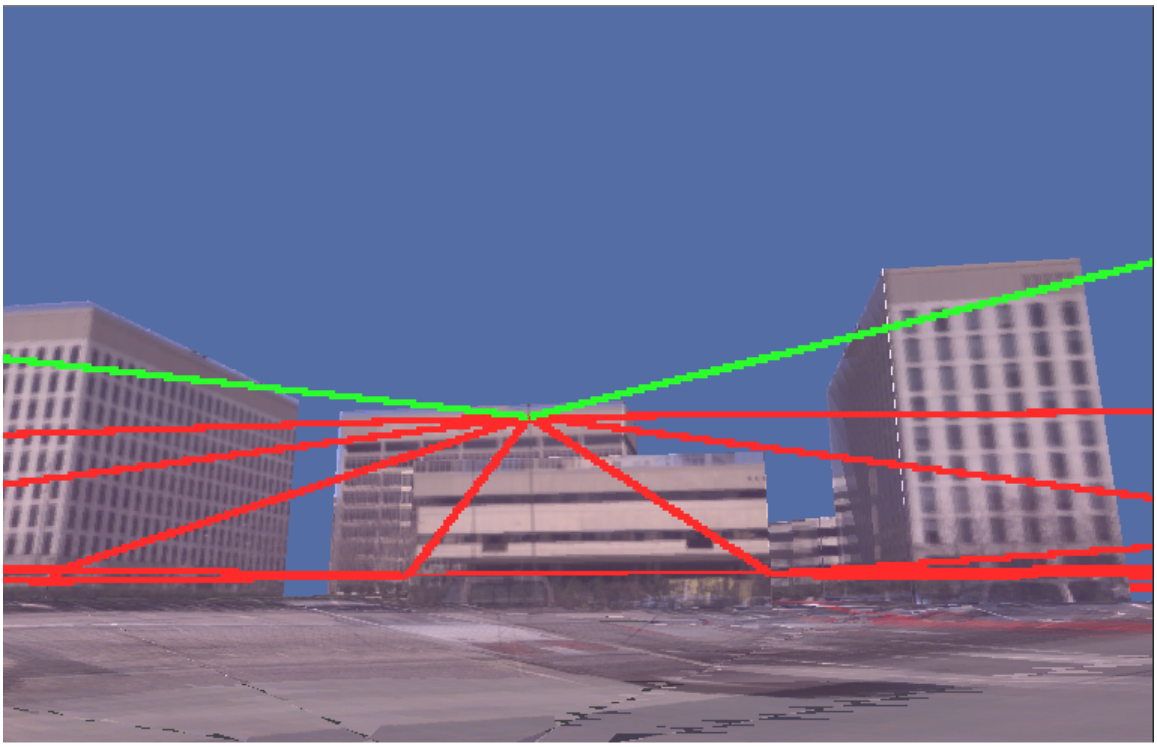

Plate 11: Another off-node view

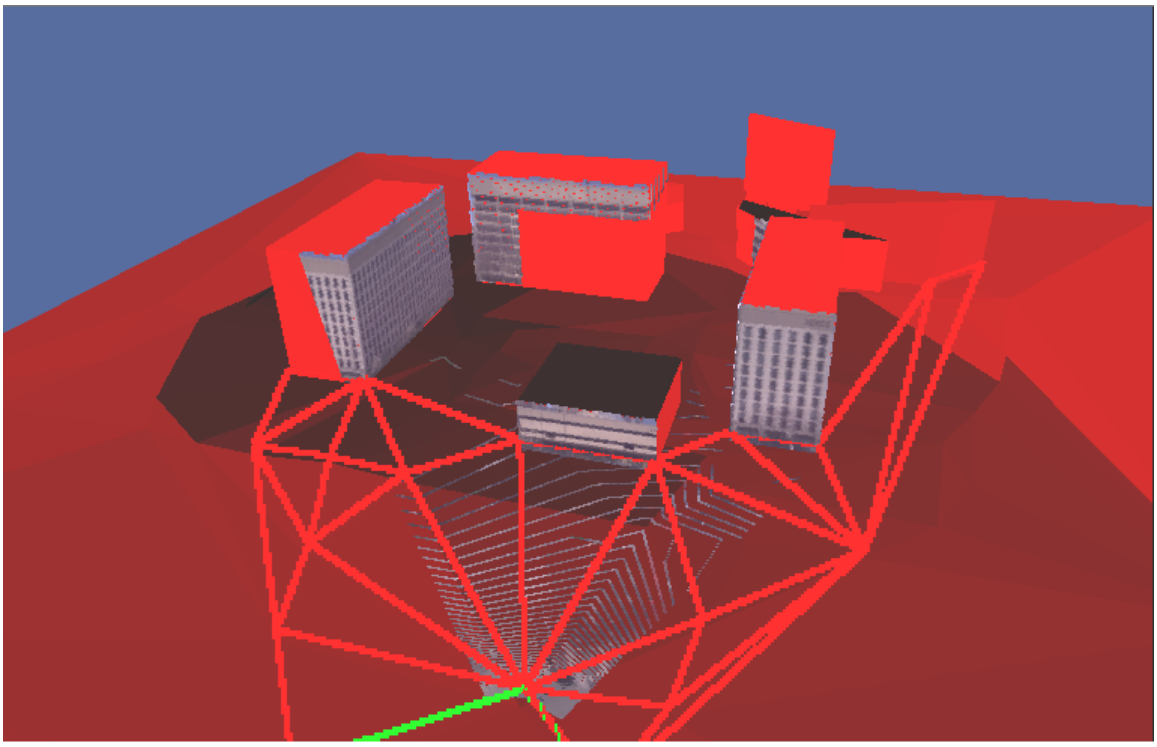

Plate 12: Visualization of projection from above view