Seamless Texture Mapping of 3D Point Clouds

Dan Goldberg
Mentor: Carl Salvaggio
Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology
Rochester, NY

November 25, 2014

Abstract

The two similar, quickly growing fields of computer vision and computer graphics give users the ability to immerse themselves in a realistic computer generated environment by combining the ability create a 3D scene from images and the texture mapping process of computer graphics. The output of a popular computer vision algorithm, structure from motion (obtain a 3D point cloud from images) is incomplete from a computer graphics standpoint. The final product should be a textured mesh. The goal of this project is to make the most aesthetically pleasing output scene. In order to achieve this, auxiliary information from the structure from motion process was used to texture map a meshed 3D structure.

1 Introduction

The overall goal of this project is to create a textured 3D computer model from images of an object or scene. This problem combines two different yet similar areas of study. Computer graphics and computer vision are two quickly growing fields that take advantage of the ever-expanding abilities of our computer hardware. Computer vision focuses on a computer capturing and understanding the world. Computer graphics concentrates on accurately representing and displaying scenes to a human user. In the computer vision field, constructing three-dimensional (3D) data sets from images is becoming more common. Microsoft's Photosynth (Snavely *et al.*, 2006) is one application which brought attention to the 3D scene reconstruction field. Many structure from motion algorithms are being applied to data sets of images in order to obtain a 3D point cloud (Koenderink and van Doorn, 1991; Mohr *et al.*, 1993; Snavely *et al.*, 2006; Crandall *et al.*, 2011; Weng *et al.*, 2012; Yu and Gallup, 2014; Agisoft, 2014).

Computer graphics deals with accurately and precisely recreating a virtual scene to match how the scene would appear in the real world. Pipelines exist to display scenes using dedicated graphics hardware. Various languages and packages make use of this pipeline and all have a similar approach which is described in § 2.3. Ideally the output scene will have 3D objects (usually sets of triangles) which all are colored accurately.

1.1 Project Overview

This project begins with a scene or target that is of interest. The user must capture a set of images from different perspectives around the target. The patch-based multi-view stereo (PMVS) software package computes a set of 3D points (point cloud) from the set of 2D images (Furukawa and Ponce, 2007). This software is available under the GNU GPL license for non-commercial applications.

Currently the software outputs a set of 3D points along with information about each point; normals and which image it is visible in. This information is stored in an ASCII file. The output is lacking edge information which is used to create a 3D model. The goal of this project is to produce a visually pleasing, 3D model. A 3D model or mesh differs from a point cloud in that it stores information about how vertices are connected to each other so that surfaces can be displayed. A set of 3D objects (triangles) is created

from that information. The mesh must then be colored with image information ("textured") to produce the desired output. The output of the PMVS routine serves as the input for this work.

This research can be broken down into three distinct modules. First, the point cloud must be made into a mesh through a process known as triangulation or mesh reconstruction. Second, an image is mapped onto each triangular facet. Third, the textured mesh must be displayed for interaction or be saved in a standard format file. These processes are outlined in the flowchart in figure 1.

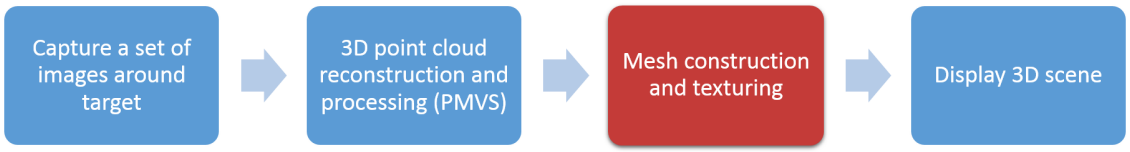

Figure 1: This is a flowchart describing the overarching project. The contribution described in this paper, the mesh construction and texture step is highlighted.

2 Background

2.1 PMVS

The PMVS package creates a 3D point cloud from input images and information about the cameras. The PMVS algorithm is described in detail in Furukawa and Ponce (2007). An overview of the algorithm follows.

Generating a 3D point cloud from a set of images is a common problem in computer vision. One such process is commonly referred to as structure from motion. This idea is similar to one way the human visual system sees in 3D. The disparity between two eyes gives a parallax effect. The visual cortex compares the image from each eye and detects the small changes. These small changes are then translated into 3D information.

Fundamentally, in structure from motion algorithms; changes from one image to another can be attributed to object geometry. In most cases the object movement is really the camera moving while the object remains stationary which achieves the same effect. By analyzing the change in features from image to image, an algorithm can track a feature along images. This allows the object geometry to be extracted. Stereopsis is the name given to this process using two images.

The first step of PMVS is to perform feature detection on each image. Harris corners and difference-of-Gaussian blobs are detected. The four strongest detections in 32×32 pixel local neighborhoods in a grid across the image are used in subsequent steps. The local neighborhoods are referred to as cells and help keep the output uniform over the whole scene.

Each feature is then compared to features in other images. The goal is to find the same feature from a different perspective. Epipolar constraints help the tracking algorithm find a good match. The term epipolar describes the geometry between two images of the same object from different vantage points. By assuming the 3D scene geometry is the same in both images, an epipolar constraint allows for a region of interest in the new image to look for the feature. The epipolar line describes a line in the new image where a feature from the old image may fall. This is shown in figure 2.

Each good tracked feature yields a patch. A patch is the data structure which holds information about the feature such as, reference image, a set of images where it is visible, its position, and its surface normal. After this initial run through, a sparse set of patches is found. A set of expansion and filtering steps are also used to create a dense set from the sparse one.

Expansion is done by iteratively adding new neighbors to existing patches until each cell is covered. For each cell in each image that the patch is in, if a neighboring cell is empty and the patch is not occluded by anything then a new patch is created in that new cell.

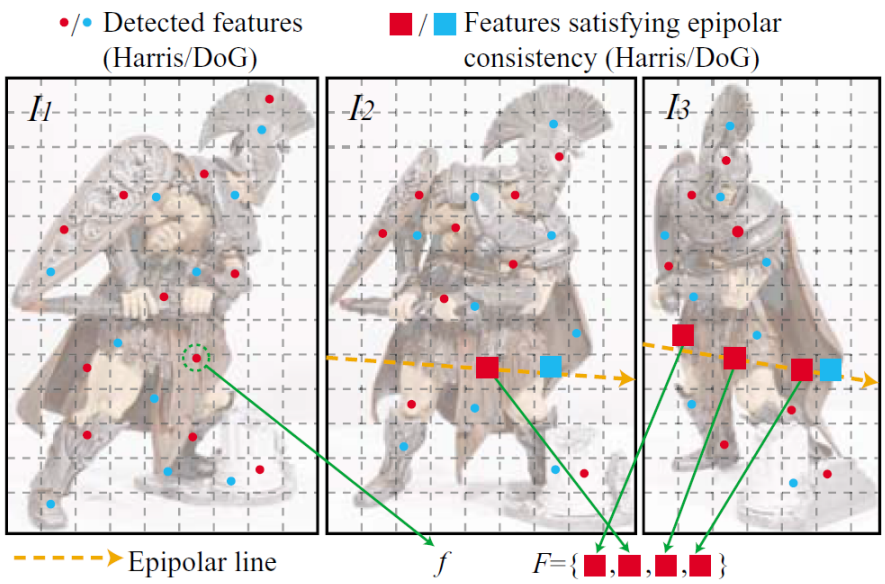

Figure 2: Feature matching step. Figure 3 from Furukawa and Ponce (2007). The features f' in F which satisfy the epipolar constraint for images I_2 and I_3 to be matched to feature f of I_1.

Filtering is then done to remove outliers. One filter removes patches that fall outside of the real surfaces and one filter removes patches that fall inside the real surfaces. A final filter performs weak enforcement of regularization which removes patches that do not have a sufficient number of n-adjacent neighbors compared to the total number of patches in a local neighborhood of cells.

The output of the software is actually an oriented point cloud. This means that in addition to (x, y, z) coordinates, each point also has a known (n_x, n_y, n_z) normal. The information is written in an ASCII format file which lists the information from each patch. In addition, a Stanford triangle format (.ply) file is written which supports storing the information for oriented point clouds.

2.2 Mesh Reconstruction

The process of converting a point cloud to a mesh is one that could be extremely problematic. The problem is that there are many ways that a set of points can be connected with edges. The problem does not have a single solution. Depending on the scene, the edge constraints may be different; some may require convex shapes, others may expect many planar surfaces. Many mesh reconstruction algorithms exist and the best one to use depends on the data set. Ones taken into consideration are simple and efficient. The scope of the project does not include a robust mesh generation. A few that are discussed briefly here are Poisson Surface Reconstruction (Kazhdan *et al.*, 2006), Delaunay Triangulation (Delaunay, 1934), and α-shapes (Edelsbrunner, 1995).

To begin, the goal for each method is to take a set of unrelated points and create a connection between three near points as a guess for a surface that could be present. Triangles are traditionally used because it is the only shape that provides information for a single plane while ensuring that all points lie on the plane.

The first mentioned method was Poisson reconstruction. This processes takes in the points and their normals from an oriented point cloud. Kazhdan *et al.* (2006) describes the process in much more detail. At a high level, the algorithm begins by spatially partitioning the cloud. The resolution of the output mesh is dependent on the depth of the spatial partitioning algorithm. A slice through a single plane of the points can show an outline of the shape. The reconstruction process on a slice is shown in figure 3.

First the indicator gradient shows an outline of the object along with the normal information (vector

field, $\vec{V}$). The key to the process is how the indicator function, $\tilde{X}$, is created from this. The goal is to solve $\nabla\tilde{X} = \vec{V}$ for $\tilde{X}$. Usually $\vec{V}$ is not integrable so the approximation is found by applying the divergence operator to form the Poisson equation shown in equation 1.

$$\Delta\tilde{X} = \nabla \cdot \vec{V} \tag{1}$$

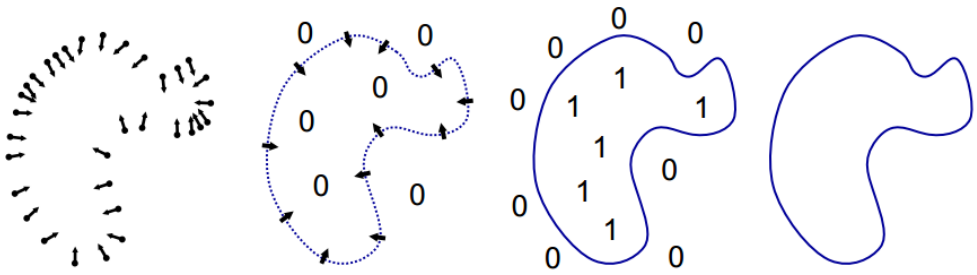

Figure 3: Poisson Reconstruction overview from Kazhdan *et al.* (2006). From left to right, the images show the oriented points, indicator gradient, indicator function, and the reconstructed surface for a slice.

Delaunay Triangulation is a much simpler method to create a mesh. In the 2D case, a valid Delaunay triangle is defined as a set of three points where there exists no point inside the circumcircle. A circumcircle is a circle that intersects each point of a triangle, this can be seen along with an example of a valid and invalid Delaunay triangle in figure 4. The process maximizes the minimum angle of the triangle. In the 3D case, there must not be any point within a circumsphere of the set of three points. Since this method simply finds any triangle in any direction with no regard for an overall structure, the method leads to a lot of extraneous triangles. A filtering step is needed to filter out the extraneous triangles.

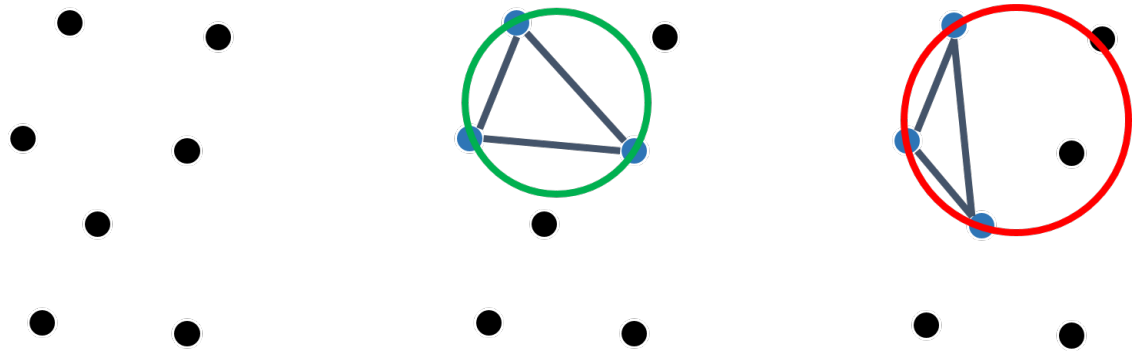

Figure 4: An example of Delaunay triangulation in a 2D case. From the whole set of points (left), a good triangle is shown (center), and an invalid triangle is shown (right). The invalid triangle's circumcircle contains another point in it.

The final process is α-shapes. This process is a uses similar concepts to Delaunay triangulation except focuses on both edges and triangles. CGAL (CGAL, *Computational Geometry Algorithms Library* 2014), a software library does a good job describing the process in its documentation by Da (2014). Tran Kai and Frank Da write, "one can intuitively think of an α-shape as the following. Imagine a huge mass of ice-cream making up the space $\mathbb{R}^3$ and containing the points as 'hard' chocolate pieces. Using one of these sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (e.g. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps,

arcs and points. If we now straighten all 'round' faces to triangles and line segments, we have an intuitive description of what is called the α-shape of S." An illustration of this process in 2D is shown in figure 5. The α describes the "size" of the spoon or the resolution of the surface construction.

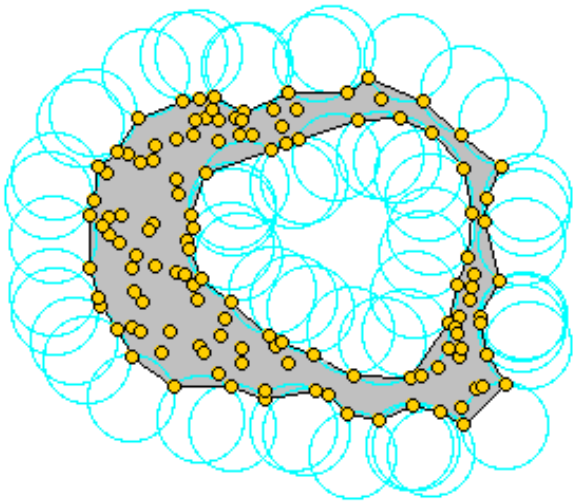

Figure 5: Illustration of what the alpha shapes method is doing from Da (2014). The yellow points could be thought of as chocolate pieces and the blue circles can be thought of as an ice cream scoop.

2.3 The 3D Graphics Pipeline: Projections and Texture Mapping

While the term texture mapping is unique to the field of computer graphics, the definition of a "texture" may change outside of the field. Outside of computer graphics, texture typically refers to the 3D features on an object that can be seen and felt. The term will be defined in the computer graphics context below. Since texture mapping is the final stage in the pipeline, an overview of the 3D graphics pipeline will be presented before a "texture" is defined.

The goal of a 3D graphics pipeline is to pleasingly display a scene in 3D on a screen so that a user may view it. There are many packages available to utilize graphics hardware to do the computations and display the scene, such as OpenGL (SGI, 2014) and DirectX (Microsoft, 2014), though all have the same general pipeline.

First the scene is created by generating objects which begin in "object space". This is the coordinate system that an object is created. To be placed in a world, the vertices must be transformed into world space. Next, after all objects are defined in a global coordinate system, a virtual camera is placed in the world.

The objects must then be transformed into camera space through either orthographic or perspective projection. Orthographic projection has no distortions for viewing location as rays come into the camera parallel. Objects farther away are the same size relative to objects close to the camera. A figure representing this type of projection is shown in figure 6. Perspective projection is for scenes in which closer objects appear larger relative to farther objects. This perspective projection mimics how all things with a pupil capture light, from cameras to eyes. As objects get farther and farther away from the pupil, the perspective effect is less and in the limit, it turns into orthographic projection.

To understand orthographic projection, imagine viewing a skyline off in the distance. All buildings, regardless of distance appear to be the same size and orientation. It is used to mimic a viewer from far away or in times when perspective may hinder viewing. The latter is the reason this projection is used for architecture applications, the viewer wants realistic proportions in the buildings. A representation of this projection is also shown in figure 6.

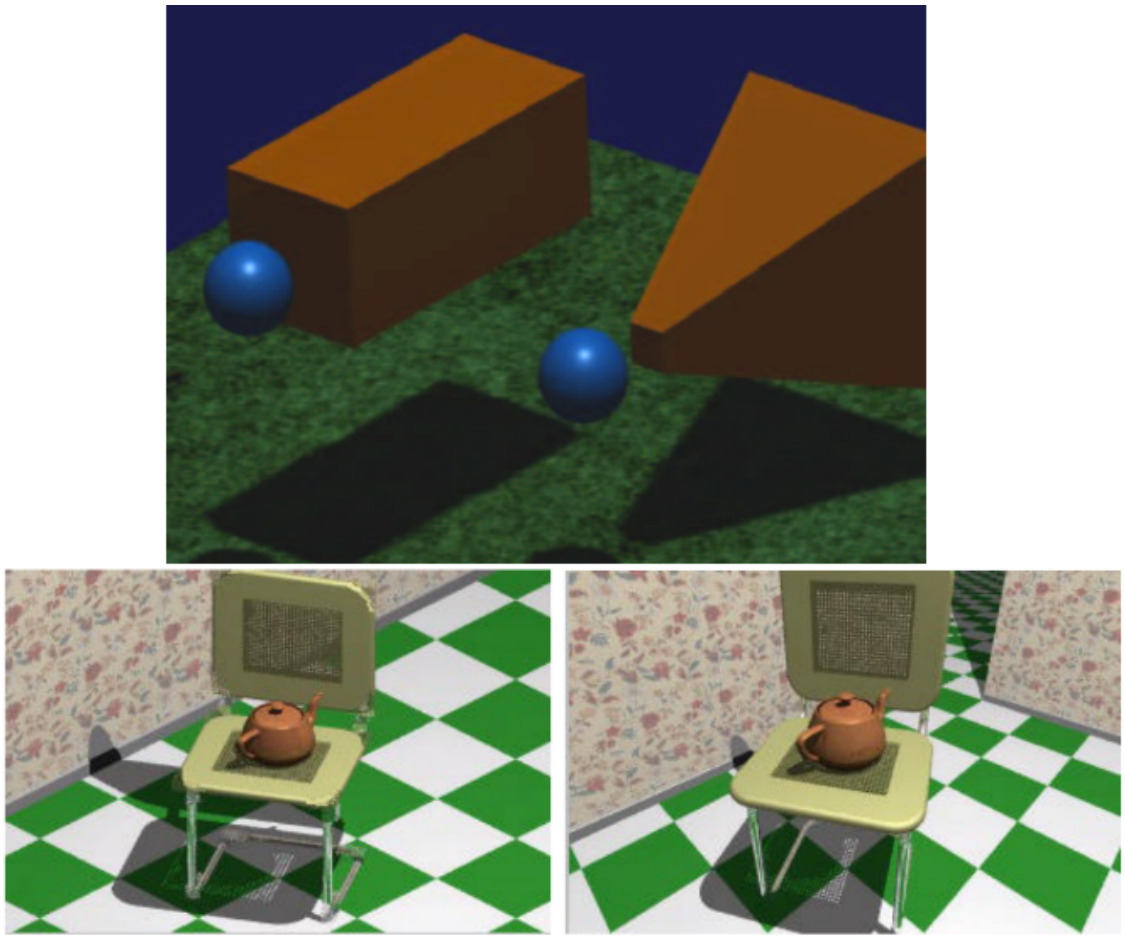

Figure 6: Example projection graphic taken from Dr. Bailey's presentation on computer graphics (Bailey, 2013). The top image provides a visualization of the 3D space that is being captured in each view (orthographic on left, perspective on right) in orange, with the camera at the blue sphere location. The bottom images show how each projection would make one specific scene appear. Again the orthographic projection is on the left and the perspective projection is on the right.

Next, the objects are transformed from camera space into screen space. This process is usually straight forward and handled by the graphics processing library used. The values of the 2D coordinates are scaled from 0 to 1 in both the x and y directions. This is done by taking the screen dimensions and dividing it by the field of view the user has defined. The objects outside of this view are clipped. This is the last step of the pipeline.

For the vertices to be properly displayed they must be colored. This inserts a step into the pipeline while the object is still in object space: texture mapping. A texture in computer graphics is anything that is used to color an object. Common textures used in early computer graphics were solid colors or patterns generated programmatically (Weinhaus and Devarajan, 1997). In the context of this project, the texture which is used to color an object is going to be taken from pixels of an image. The assignment of a pixel to a location on an object is the mapping component. Figure 7 shows how an image can be used to texture an object when some reference coordinates are known.

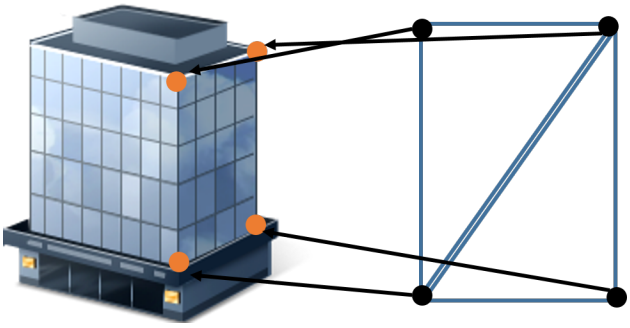

Figure 7: Example showing two facets (right) referencing a texture (left) for information about how to color the facet. The coordinates on the image (orange) would be 2D (u, v) points while the coordinates on the facets would be 3D (x, y, z) points.

The method of assigning an object coordinate to a texture coordinate is known as UV mapping since the texture coordinate is from a 2D image (u, v) and needs to be mapped to a location in 3D space (x, y, z). This is one of the final components of the 3D graphics pipeline.

In order to pick the proper (u, v) coordinate from a texture image, the location must be calculated. A camera projection matrix is a transformation matrix which describes how a 3D point in space is converted into a 2D point on the image plane. This is usually a 3×4 matrix which is applied to homogeneous coordinates (discussed in § 4.6). When the object in 3D space is in camera space, a simple camera projection transformation will give the (u, v) coordinate of the point on an image.

After a 2D (u, v) coordinate is matched to a 3D (x, y, z) vertex, the pair is sent to the graphics processor along with the texture file to be displayed.

3 Dataset Used

The target dataset that was used in this project was a scene of downtown Rochester, NY, USA. The initial scene was created with 48 images from an aerial camera around the city. Exelis flies an aircraft with 5 sensors mounted on it. The 4 oblique sensors are used in this project. They are Kodak KAI-16000 CCD detectors with 16MP resolution $(4,872 \times 3,248)$ and $7.4\mu m$ square pixels. Some example images are shown in figure 8.

4 Project Workflow Overview

The general steps in the workflow are shown in figure 9. It begins with three inputs that come from the following: 1) PMVS PLY file (after it is triangulated), 2) PMVS patch file, and 3) PMVS camera files. The

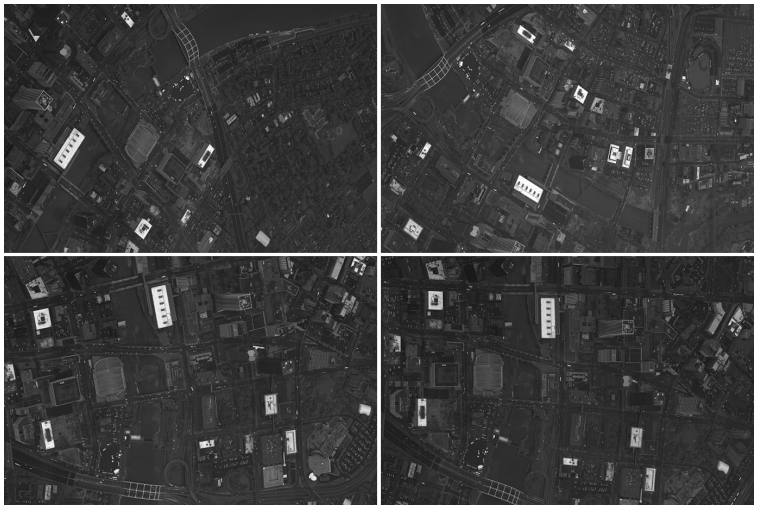

Figure 8: 4 of the 48 images used for this project are shown here. They are provided to give an idea of the images that Exelis provided.

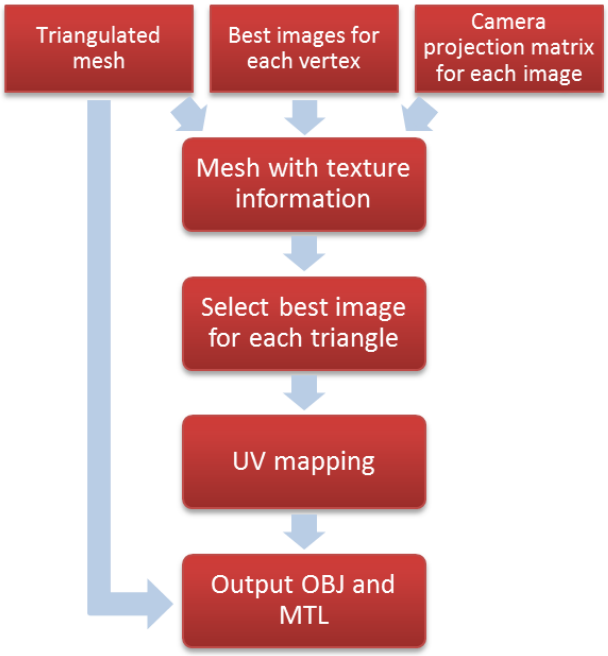

Figure 9: Workflow flowchart showing the major steps in this project. The 3 inputs are combined and processed to achieve the output textured mesh OBJ file.

final output file is able to be read in by any wavefront OBJ file viewer. The workflow is described in the following subsections.

I took an object-oriented approach using C++ and the Eigen (Guennebaud and Jacob, 2010), PCL (Rusu and Cousins, 2011), and OpenCV (Bradski, 2000) (for image IO) libraries. The UML diagram of my class structure can be seen in figure A.7.

4.1 Preprocessing

As stated previously, the first step is processing the images using PMVS. The point cloud produced was originally 2,000,409 points. In order to process the data further the point cloud was cropped to a subsample of 213,647 points (see figure 10). Cropping was done spatially by a bounding box which was taken around a region of large buildings. A mesh was created from the point cloud using an α-shapes algorithm implemented in a third-party application called MeshLab (Visual Computing Lab of ISTI - CNR, 2014). The software used to create the mesh has no effect on the following steps of the workflow. Any application could be used to facetize the data.

The three meshing techniques discussed in § 2.2 have implementations in MeshLab. The α-shapes algorithm was chosen over Delaunay triangulation and Poisson Reconstruction for a few reasons. Delaunay triangulation had many triangles which connected points which should have been separate. Also it required an extra filtering step. Poisson Reconstruction assumes the object it is constructing has no openings (is "water tight"). This is not a good approximation for this application. Overall, α-shapes gave the best looking output from MeshLab's built in algorithms. In the end, 1,871,228 faces are created from the 213,647 points. The settings used were the default settings in MeshLab.

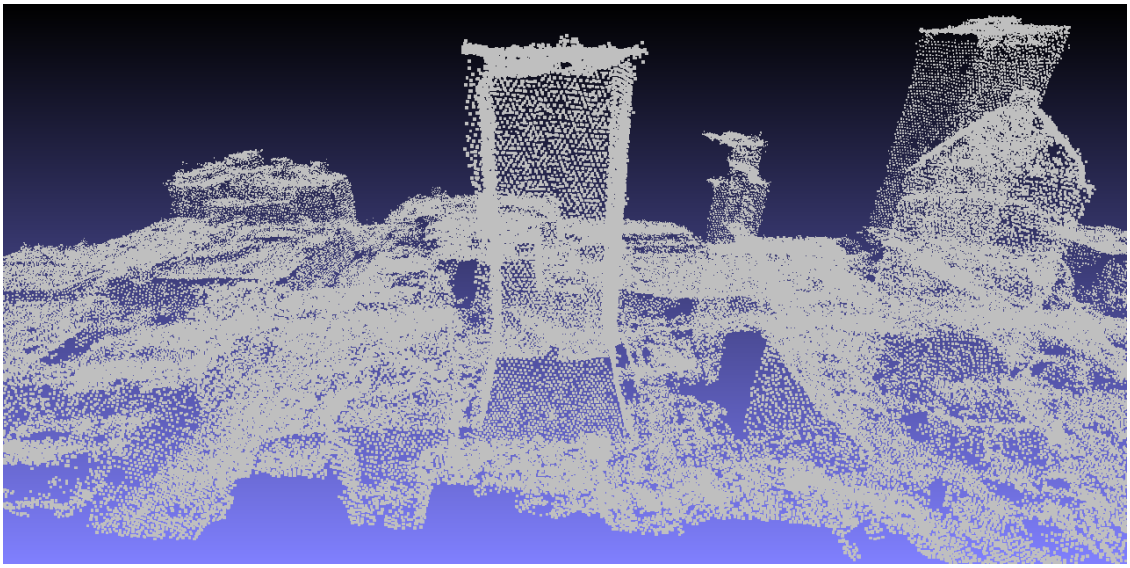

Figure 10: Sample point cloud output of downtown Rochester. This is the Rochester scene after it was processed with PMVS and then cropped.

The mesh is required so that the information about which vertices are connected can be combined with (u, v) texture coordinate information. There will then be triangles in the texture which correspond to each triangle facet. The edge information will then be written out to a wavefront OBJ file. The graphical processing library (OpenGL for MeshLab) will texture each pixel in the facet with an interpolated texture from the triangle in the texture.

4.2 File I/O

Information from various auxiliary files must be combined in the final product. The mesh PLY file, patch file, camera projection matrix files, image files must all be read in.

Two auxiliary classes shown in the UML diagram in figure A.7 are used to hold the information read in. A Patch object is formed for each element in the patch file from PMVS. It is read in and parsed into the Patch object using a C++ stringstream. The (x, y, z) point location, point normal, number of images that the point is visible in, and the indices of the images that the point was visible in, are all stored in the object. The point index is also defined in the Patch object but is not initialized until the next step in the workflow.

Camera objects stored information about each camera. The information read from the PMVS camera files was the file name of the image that corresponds to the camera and the camera projection matrix for that camera.

When the information is combined, a triangle with the patch file information is constructed. This list of triangles is what is subsequently processed in order to obtain a set of UV coordinates from the information.

4.3 Point cloud and patch file association

Each point in the input PLY needs to be associated to its patch file information. The naive solution to this would be to go through each point in the cloud and find the patch object with the corresponding (x, y, z) location. This process takes an extremely long time with a complexity of $O(N^2)$.

For efficiency, a spatial partition tree is used for the search. Each point read from the PLY is put into a K-D tree. A K-D tree is a binary spatial partition tree which recursively cuts a K-D space into binary sections along an axis. The points are always the leaf nodes of the tree. This tree is thoroughly described by Bentley (1975). A 2D example is shown in figure 11. This results in a clean, fast way to find a point in 3D space. The implementation used in this project was from the Point Cloud Library (Rusu and Cousins, 2011) C++ package. The average complexity of the association process using a K-D tree is $O(N \log N)$.

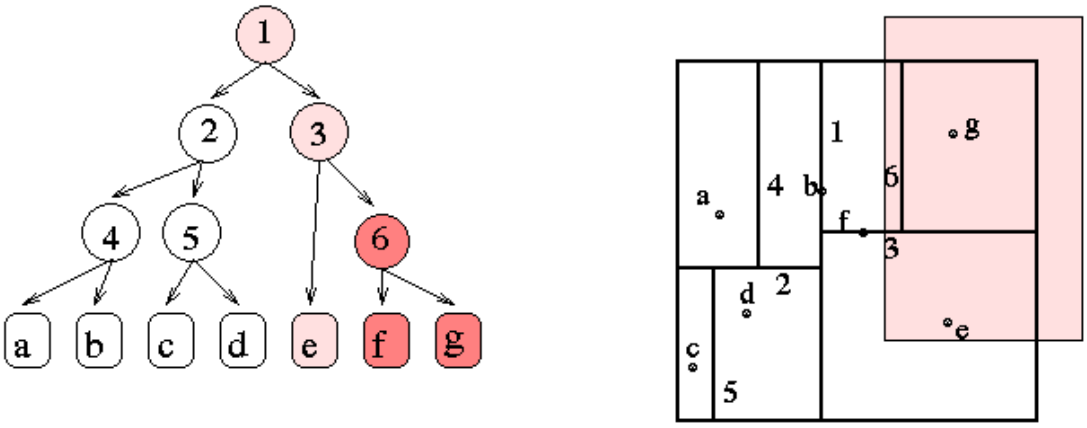

Figure 11: This is an example 2D K-D tree from `http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/kd.html`. The left shows a tree representation of the spatial partition shown on the right. The space is divided into 2 sections by a line. Each line is the parent node in the tree. Each space is divided again until only one point exists in the space. Usually, there can be more than one point at the lowest layer of the tree. A user may define a number n points to have at the tree's lowest layer.

When a patch is read in from the auxiliary patch file, the corresponding point is searched for in the K-D tree. If it is found, the patch is added to the list of patches used along with the index of the point in the PLY file. The list of indices is needed since that is how the triangle information is stored.

A problem arose in which duplicate (x, y, z) locations existed in both the patch and PLY files. About 10% of the points in the large dataset had at least one duplicate point. Most likely, this is due to the limits of the single precision data type which is used in the PLY and patch files. In order to deal with this, once the patch is added, that point is marked as taken and when a subsequent point with the same (x, y, z) coordinate is searched for, a different vertex with the same coordinate is chosen. In the large dataset used in this project, there were at most four point indices with the same coordinates.

4.4 Image Selection Process

For each facet, a single texture must be chosen to color it. The best texture to use for a single facet would be a section from an image or combination of images which include that triangle. The goal of the selection process is to choose an image region which gives the facet an accurate coloring. An accurate color will be determined by a set of parameters to measure accuracy of an image. The best way to determine the most accurate image would be to take various properties into account, assign weights to them which correspond to visual importance, and create a cost function from a linear combination of the properties. The complexity of the cost function depends on the parameters defined.

Before the cost function comes into play, the possible images must be identified. Each vertex of a triangle holds the patch information from PMVS. Included in this is the set of images that the point is visible in. The information is combined and a single set of images is gathered for the triangle as a whole.

Initially, it was expected that each facet would have at least one image that included each vertex. The large data set used with around 2,000,000 facets had around 86,000 facets where only two of the three vertices were visible according to PMVS — about 4%. The list of images that the facet is visible in is then just the images that two vertices are visible in.

After compiling the list of images that the facet is visible, the cost of each image is analyzed. A single image index is attached to that facet as a result of the cost calculation.

4.5 Cost Function

A cost function is used to incorporate various parameters in the image selection process. For each facet, the image with the smallest cost is chosen to texture it. The cost function is applied only to images in which the facet is visible. These specific metrics were chosen to represent different factors that effect image quality and image similarity to the scene.

There are three parameters that go into the cost function, viewing, brightness, and distance to the center of the image. Figure 12 shows an illustration of these three parameters. The first parameter, viewing, takes into account how well each camera can see the facet. This is quantified by the difference in angle between the camera pointing vector and facet normal. This quantity is related to the dot product of the two vectors (after they are normalized) as shown in equation 2.

$$\hat{a} \cdot \hat{b} = \cos\theta \tag{2}$$

Rather than performing an arccosine step, the cosine of the angle is used in the parameter. The cosine is subtracted from 1 since a low value cost is favorable, and cosine values near 1 correspond to small angles. The parameter that is incorporated into the cost function is shown in equation 3 where v is the parameter for a given image index, q, and a given facet index, p, $\hat{\boldsymbol{c_n}}$ is the normalized camera normal, and $\hat{\boldsymbol{f_n}}$ is the normalized facet normal. This parameter is unique for each facet and camera combination.

$$v_{qp} = 1 - |\cos(\hat{\boldsymbol{c_{nq}}}, \hat{\boldsymbol{f_{np}}})| \tag{3}$$

The next parameter is the brightness parameter. It is calculated for each camera. The average brightness $\bar{x}$ of each image index i needs to be calculated. If the images are color, they are converted to grayscale prior to any operations. This conversion is done using a linear combination of the three channels as per the default values for the OpenCV function. The default values for red, green, and blue weights respectively are as follows: 0.229, 0.587, 0.114. The average of each image is a simple operation for each pixel index j for M

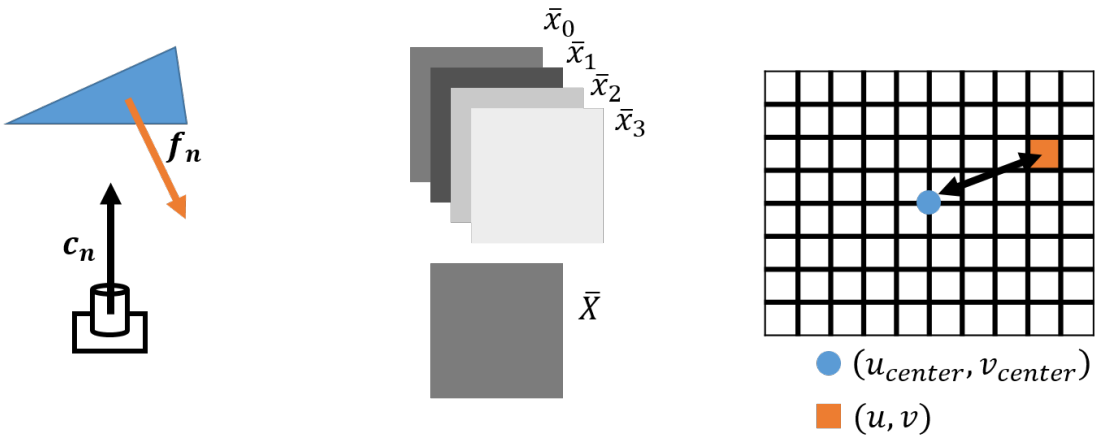

Figure 12: Illustrations of the 3 parameters for the cost function. Viewing (left) takes into account the angle between the camera and facet normal. Brightness (center) takes into account each image's average value versus the average of the whole image set. Distance (right) takes into account the (u, v) location on the image and how close it is to the center of the image.

total images where a single pixel in image i at location j is x_{ij}. The equation for this operation is shown in equation 4.

$$\bar{x_i} = \frac{1}{M} \sum_{j=0}^{M-1} x_{ij} \tag{4}$$

Next, average brightness across all N images ($\bar{X}$) in the set is needed. Since all images have the same number of pixels and averaging is a linear operation, the average value across all images is the same as the average of the average of each image. The calculation for the average across all images is shown in equation 5.

$$\bar{X} = \frac{1}{N} \sum_{i=0}^{N-1} \bar{x_i} \tag{5}$$

The absolute difference of each image brightness to the overall brightness is the brightness parameter (b) for the current image index, q, as shown in equation 6.

$$b_q = |\bar{x_q} - \bar{X}| \tag{6}$$

The final parameter used is a "distance to center" metric. This is calculated for each facet index, p. Images should have less distortion toward their centers. In order to create a parameter to describe this, for each facet, the (u, v) location on the image is calculated. The distance to the center of the image (normalized by the size) is then the parameter. It is shown in equation 7. In this equation, (u_q, v_q) is the average value of the (u, v) for each vertex of facet q and (u_{center}, v_{center}) is the center of the image. A graphical representation was shown in figure 12.

$$d_q = \sqrt{(u_q - u_{center})^2 + (v_q - v_{center})^2} \tag{7}$$

The three terms of the cost function are then combined linearly. The weights $[w_0, w_1, w_2]$ for each term can be inputted by the user. The only constraint on the weights is that they sum to one ($w_0 + w_1 + w_2 = 1$). If the user's inputs do not meet this requirement, they are scaled automatically so that the sum is one. This allows the user to determine which parameter should be focused on more when choosing an image.

The overall cost, C is defined in equation 8 for any image index, q, and any facet index, p.

$$C_{qp} = w_0 v_{qp} + w_1 b_q + w_2 d_q \tag{8}$$

4.6 UV Mapping

The position of each vertex in the image can be calculated. This mapping process is made easy by using the camera matrix information from the PMVS routine. Each (x, y, z) location can be projected on the 2D camera plane by using the camera projection matrix, P. This process requires the (x, y, z) coordinates to be projected into homogeneous coordinate space. In homogeneous coordinates, there is an extra dimension which can be thought of as a type of "weight". A 3D homogeneous coordinate $(1, 1, w)$ describes an infinite set of points in 2D along a 3D line, specifically $(1/w, 1/w)$. Figure 13 describes this with an image.

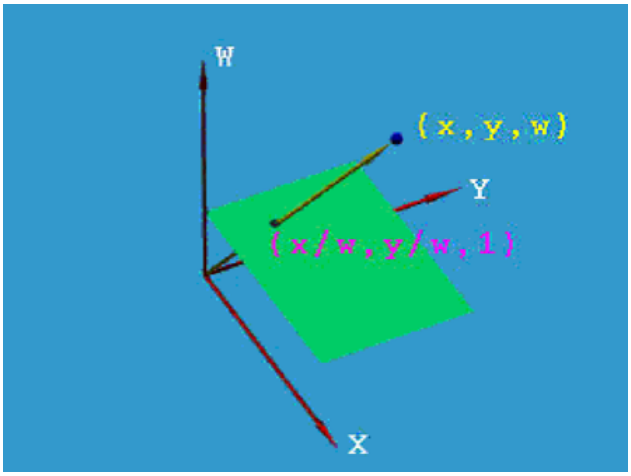

Figure 13: One way to visualize homogeneous coordinates is shown here. The image was borrowed from Dr. C.K. Shene of the Michigan Technological University (`http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/geometry/homo-coor.html`). The green plane is the plane $w = 1$. The line extending from the origin describes the homogeneous point (x, y, w) or the 2D point $(x/w, y/w)$.

Homogeneous coordinates are commonly used when doing geometric transformations because it allows translations to occur more simply; with matrix multiplication rather than a component-wise addition. An example of this is shown in equation 9. Also, this allows translation and rotation to be done in a single step by multiplying their respective matrices together.

$$\begin{bmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+h \\ y+k \\ 1 \end{bmatrix} \tag{9}$$

The component to transform a 3D point in space to a 2D point on a camera plane would then be a 3×4 matrix, $\boldsymbol{P}$. This operation is shown in equation 10. A visual representation of what the camera projection matrix is shown in figure 14. To get the final 2D, (u, v) coordinates, the homogeneous coordinate must be converted back into a Cartesian coordinate by the relationships shown in equation 11 and 12.

$$\boldsymbol{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} u_h \\ v_h \\ w \end{bmatrix} \tag{10}$$

$$u = u_h/w \tag{11}$$
$$v = v_h/w \tag{12}$$

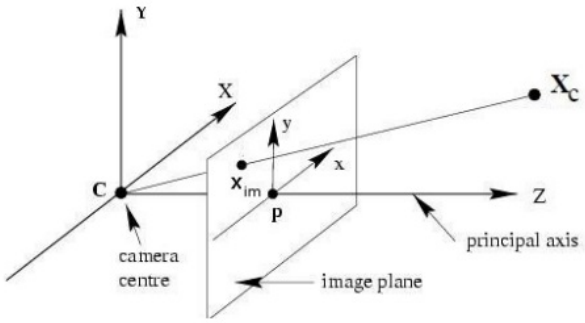

Figure 14: Visualization of what a camera projection matrix does. The (x, y, z) vertex X_c in camera space (C) is projected onto a 2D plane (P) where the image is formed. The 2D (u, v) location on the image is represented by x_{im}.

The mapped (u, v) values are then saved along with the image of that triangle. These are saved as the local (u, v) values in a triangle list. As discussed earlier, the method of display is going to be a third party software package to view OBJ files. The process of texture mapping is easier to do when every (u, v) coordinate in the environment is in the same reference coordinate space. This means the images need to be combined into a single image and the local (u, v) coordinates must be adjusted to a global (u, v) in the new combined image.

4.7 Texture Atlas

In computer graphics, the combination of numerous texture files into a single file is known as a texture atlas. There are various ways to create a single texture atlas from a set of images. The most complex would take the known facet textures and intelligently blend and combine images of nearby triangles and place them on a texture atlas. This process was not implemented here, but could be a great way to combine visually pleasing blending to a local neighborhood in the output file. An example intelligent texture atlas is shown in figure 15.

The process implemented in this work was a simplistic, naive approach that simply concatenated the various images into a single image file. This makes the computation simple for remapping the UV components. The workflow used here currently concatenates the images along the horizontal dimension. The equations for computing the texture atlas location (u_{atlas}, v_{atlas}) is shown in equations 13 and 14. In these equations, the individual image location is represented by u_i and v_i, where i is the image index going from 0 to $N - 1$, and the width of each image is W and equal for each image.

$$u_{atlas} = \left\lfloor \frac{\lfloor u_i \cdot W \rfloor + i \cdot W}{W \cdot N - 1} \right\rfloor \tag{13}$$
$$v_{atlas} = v_i \tag{14}$$

4.8 Display/Output

Displaying the textured scene is a vital part of the process. Rather than displaying the scene in my project, it would be more and favorable to have a generic output that any basic graphics renderer would be able to

Figure 15: Example intelligent texture atlas from an example on the website of Agisoft Photoscan (Agisoft, 2014). Notice that an area that is close together on the atlas will be close together in 3D. Currently the method to create something this accurate is that is done by hand.

display. The OBJ file format is something that satisfies this requirement.

How the OBJ file satisfies the requirement and what it brings to the table that makes it a complete product versus the PLY format is that it can store texture information. The OBJ file includes all of the information that a graphics renderer would need: vertex information, facet/connectivity information, texture coordinate location, and texture file name.

5 Results

In the end, the wavefront OBJ output from the steps described here would not properly display. The textures from the texture atlas would not load properly in viewers (see figure 16). Multiple viewers had similar display problems due to the size of the texture atlas. In order to determine the cause of the display problems, multiple test atlases were created. To also test the validity of the (u, v) mapping, a specific pattern was used as the texture atlas. The test texture atlas is shown in figure 17.

The viewers tested were MeshLab, CloudCompare (EDF R&D, Telecom ParisTech, 2014) and Blender (Crowder, 1999). Blender's controls made the output difficult to manipulate and analyse with increased delays in response time with larger files. An analysis of atlas size was done for both CloudCompare and MeshLab. The results from the two viewers were the same.

The comparison is shown in table 1. The original texture atlas is atlas number 1. It was resized to dimensions that were closer to square in atlas 2 to test if aspect ratio was a problem. The smallest (number of pixels) file tested that did not render was atlas 3 at around 576 MP. Atlas 4 was the largest atlas that did properly render at 529 MP. Another aspect ratio at a smaller overall size was done in atlas 5. This aspect ratio (roughly 1:72) is the same as the original texture atlas. To test if the file size on disc had any effect, atlas 6 was uncompressed but the same size as atlas 5. After these tests, the results show that the only factor that had an effect on the viewer's ability to render to mesh was the number of pixels in the image. Both MeshLab and CloudCompare do not render OBJ files with texture files that have somewhere between 529 and 576 million pixels.

The colored display showing (u, v) accuracy is shown in figure 18. This was generated using the pattern in 17 as the texture atlas. In the process described in this paper, the texture atlas was created in a way that each source image was concatenated together in the horizontal dimension. This means the changes in color hue in the pattern correspond loosely to changes in source image number. Many different surfaces in

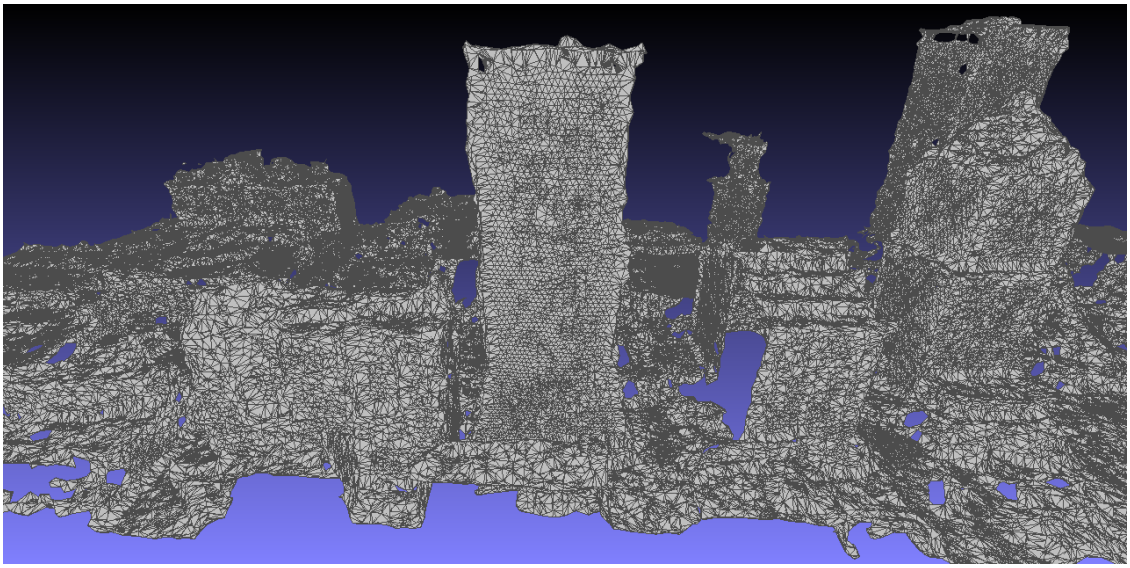

Figure 16: This is the output from my workflow. The scene is the Rochester scene as output by PMVS and then cropped. The textures were not loaded by MeshLab since the size of the texture atlas file was too large for it to handle as a single texture file.

Figure 17: Pattern used as a test texture atlas to show the (u, v) mapping. This pattern changes color hue in the horizontal direction. The original texture atlas as outputted from the process described in this paper is arranged with each source image concatenated horizontally. This means that in the test texture atlas shown here, different colors correspond to source images.

Table 1: The results of the size analysis using the real texture atlas and test texture atlases. All images used for the analysis of size were single-plane, grayscale images. After an atlas was created, the OBJ was rendered using both MeshLab and CloudCompare. If the texture appeared on the rendered mesh, it was counted as rendered.

Atlas Number	Size	Rows (thousands)	Columns (thousands)	Number of Pixels (millions)	Successfully Rendered
1	325 MB	3.3	234	759.6	No
2	331 MB	34	22	759.6	No
3	2.35 MB	24	24	576	No
4	2.25 MB	23	23	529	Yes
5	1.86 MB	2.4	170	400	Yes
6	381 MB	20	20	400	Yes

figure 18 are different colors from each other, but within the surfaces there is relatively little change in color. This is expected. The goal of the mapping was to use the same image for similar portions of the mesh.

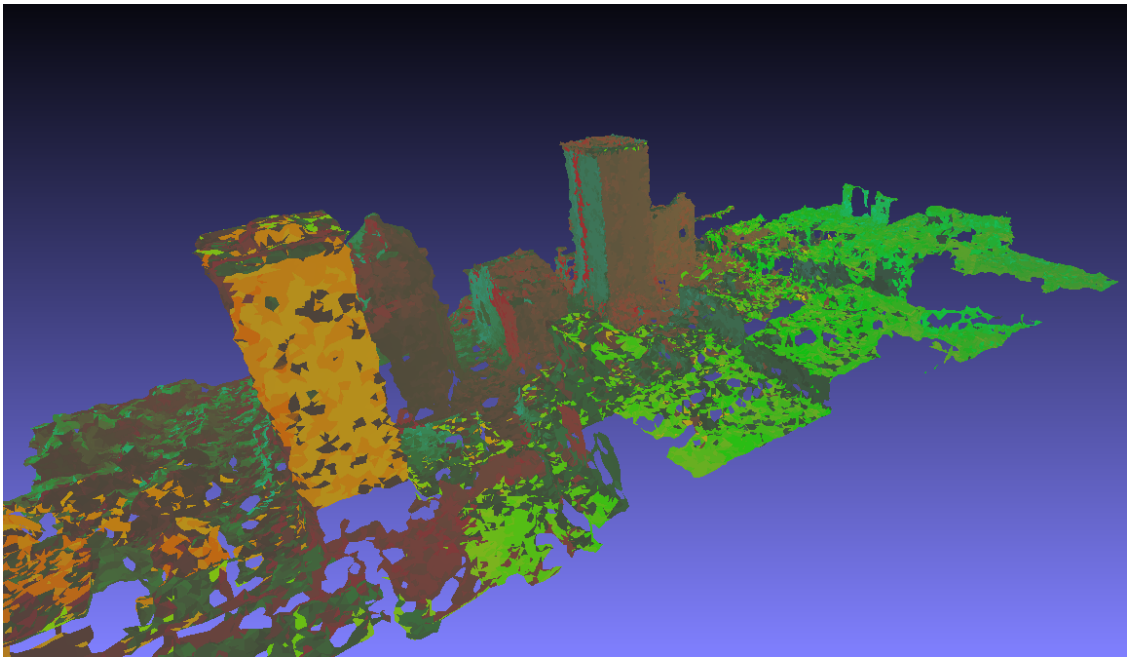

Figure 18: This is an altered output from my workflow. The scene is the Rochester scene as output by PMVS and then cropped. The texture atlas used was a simple rainbow pattern shown in figure 17. This was done to show how the texture atlas was mapped to the scene. The uniform colors on nearby plains suggest the mapping was done correctly.

Dr. Nghia Ho from Monash University released code (BSD license) which performs every step outlined in this paper except the texture atlas step (Ho, 2013). His output is the output shown in figure 19. This output is using a different, smaller dataset. Dr. Ho's process did not properly handle cropping of the input PLY. Instead of a texture atlas, the MTL file carried information about each texture image separately. In addition, the 3 parameters of the cost function discussed here were implemented in that program. This output shown is the result with the three parameters of $(1, 0, 0)$ for the viewing angle, image brightness, and

center distance metrics respectively. This will be the format for discussing the outputs in this paper.

The program was run with parameters of $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$, and $(1, 1, 1)$. The program normalizes the weights after the user inputs them. The results of these runs are shown in the appendix in figures A.1, A.2, A.3, A.4, A.5. The program run with $(0, 0, 0)$ only uses images that can see the face and chooses one arbitrarily. In addition to those runs, one run was done using a random image from the set of all images to texture each facet. This result is shown in figure A.6. All results which use images that can see the face are similar and the cost parameters make a negligible difference.

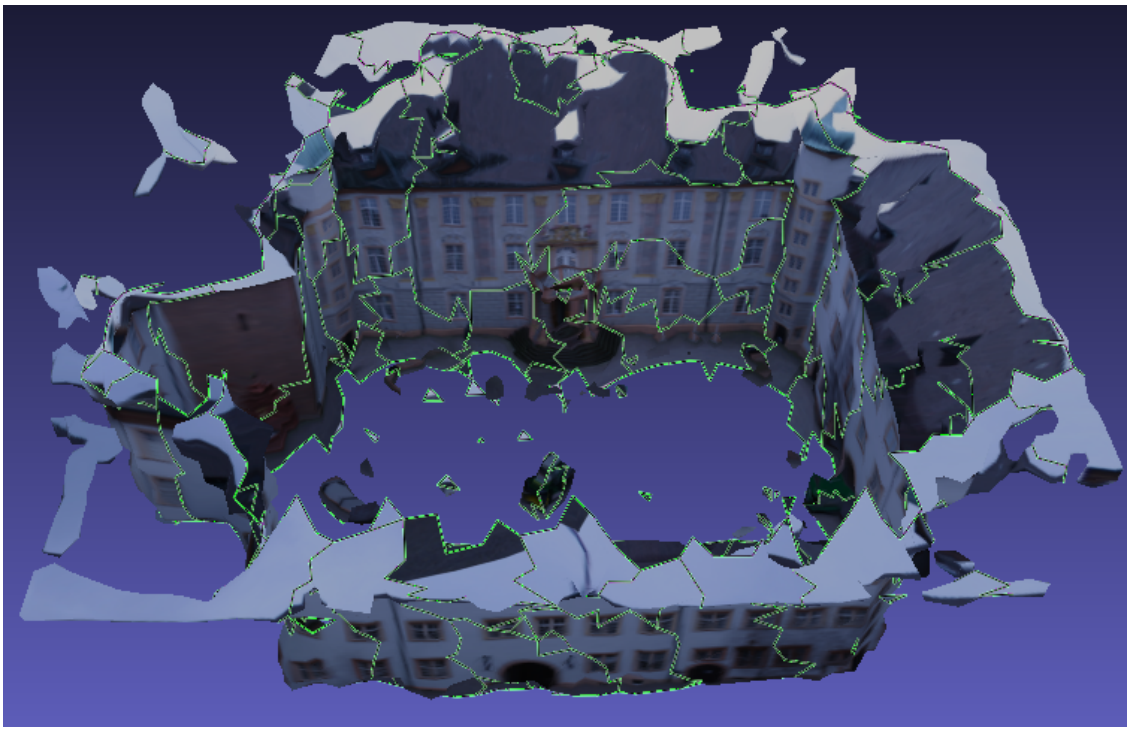

Figure 19: Output of Dr. Ho's "castle" scene. Green lines superimposed indicate where the texture is from a different image.

One example problem area is shown in figure 20. From this it can be seen that the optimal image defined by the cost function still does not provide a smooth, visually appealing result. The best way to properly texture the mesh must include a blending of a local neighborhood. To do this with the computer graphics approach, a new texture must be created with the blended images. An intelligent texture atlas could be one approach to accomplish this. The cost function alone cannot achieve a visually appealing image.

6 Conclusions

Overall, with processing power and storage potential increasing, using computer vision to reconstruct large 3D scenes with photos is becoming very popular. Additional software is needed in order to properly view these data sets. The knowledge of camera information greatly increases the ability to texture surfaces with aesthetically pleasing results.

In the end, software was created to bring the PMVS output to a more complete computer graphics output. The original output was only a point cloud without object information. The output from Ho (2013) code creates a pleasing textured 3D mesh for viewing and analysis.

My code produced a wavefront OBJ as well as the needed MTL file for texture information. The difference was that I included a texture atlas step so my code was made to work with a single texture image. The

Figure 20: This figure provides a closer look at the castle scene. The region shown here shows the interface that this workflow attempted to correct. What is happening here is that an image is being used to texture the left half and a different image is being used to texture the right half. Ideally the seam that is visible at the interface would be unnoticeable. This could be corrected by blending the textures in an intelligent texture atlas.

reason why no textures appear on my output is that MeshLab has problems with texture images that are too large. Another viewer was used as well. The scene was textured though it was obviously incorrect. The viewer also would freeze a few seconds after loading the file due to the large file size. An intelligent texture atlas would also improve the size and this problem would be non-existent.

7 Future Work

In the future, the scene could be improved further by implementing a smart texture atlas technique, taking into account a nearby neighborhood of images. This could allow for an image processing approach to the computer graphic problem where textures are thought of as independent from one another per object. Blending between adjacent image sections would be easy to implement with this approach.

In addition, other cost functions could be implemented with various parameters that fit the need of other applications. Quantitative and qualitative experiments could be designed to assess the effectiveness of various cost functions.

Another improvement would be implementing the mesh restructuring step within the code. This should be possible but was outside of the scope of this project.

Acknowledgements

There are many great professors at RIT who have taught me worlds about computer graphics and computer vision. For these subjects, I'd like to recognize my advisor, Dr. Carl Salvaggio, committee member and professor of my computer vision class, Dr. Harvey Rhody, committee member Mr. Paul Romanczyk, and my two computer graphics professors, Dr. Reynold Bailey and Dr. Joe Geigel. Learning about the nuances of compiling and managing packages in Linux I'd like to thank Mr. Jason Faulring. Finally, I'd like to recognize Dr. Nghia Ho who helped me with getting his code up and running and with other questions I had.

References

Agisoft (2014). *Photoscan*. URL: http://www.agisoft.com/ (visited on 10/17/2014).

Bailey, R. (2013). *3D Viewing and Projection*. Rochester Institute of Technology. Introduction to Computer Graphics (CSCI 510) Lecture Notes. URL: http://www.cs.rit.edu/~rjb/.

Bentley, J. L. (1975). "Multidimensional Binary Search Trees Used for Associative Searching". *Commun. ACM*. 18 (9), pp. 509–517. DOI: 10.1145/361002.361007.

Bradski, G. (2000). *Dr. Dobb's Journal of Software Tools*. URL: http://www.drdobbs.com/open-source/the-opencv-library/184404319.

CGAL, *Computational Geometry Algorithms Library* (2014). URL: http://www.cgal.org (visited on 10/17/2014).

Crandall, D., Owens, A., Snavely, N., and Huttenlocher, D. (2011). "Discrete-continuous optimization for large-scale structure from motion". In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pp. 3001–3008. DOI: 10.1109/CVPR.2011.5995626.

Crowder, B. (1999). "Blender". *Linux J.* (60es). URL: http://dl.acm.org/citation.cfm?id=327728.327735.

Da, T. K. F. (2014). "2D Alpha Shapes". In: *CGAL User and Reference Manual*. 4.5. CGAL Editorial Board. URL: http://doc.cgal.org/4.5/Manual/packages.html#PkgAlphaShape2Summary.

Delaunay, B. N. (1934). "Sur la sphère vide". *Bulletin of Academy of Sciences of the USSR*. (6), pp. 793–800.

Edelsbrunner, H. (1995). "Smooth Surfaces for Multi-Scale Shape Representation". In: *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK, UK: Springer-Verlag, pp. 391–412. DOI: 10.1007/3-540-60692-0_63.

EDF R&D, Telecom ParisTech (2014). *CloudCompare (version 2.2)*. URL: http://www.cloudcompare.org/ (visited on 11/24/2014).

Furukawa, Y. and Ponce, J. (2007). "Accurate, Dense, and Robust Multi-View Stereopsis". In: *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE*, pp. 1–8. DOI: 10.1109/CVPR.2007.383246.

Guennebaud, G. and Jacob, B. (2010). *Eigen v3*. URL: http://eigen.tuxfamily.org (visited on 10/17/2014).

Ho, N. (2013). *TextureMesh*. URL: http://nghiaho.com/?page_id=1629 (visited on 10/17/2014).

Kazhdan, M., Bolitho, M., and Hoppe, H. (2006). "Poisson Surface Reconstruction". In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. SGP '06. Cagliari, Sardinia, Italy: Eurographics Association, pp. 61–70. URL: http://dl.acm.org/citation.cfm?id=1281957.1281965.

Koenderink, J. J. and van Doorn, A. J. (1991). "Affine structure from motion". *J. Opt. Soc. Am. A*. 8 (2), pp. 377–385. DOI: 10.1364/JOSAA.8.000377.

Microsoft (2014). *Microsoft DirectX*. URL: http://windows.microsoft.com/en-us/windows7/products/features/directx-11 (visited on 10/17/2014).

Mohr, R., Veillon, F., and Quan, L. (1993). "Relative 3-D reconstruction using multiple uncalibrated images". In: *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR '93., 1993 IEEE Computer Society Conference on*, pp. 543–548. DOI: 10.1109/CVPR.1993.341077.

Rusu, R. B. and Cousins, S. (2011). "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567.

SGI (2014). *OpenGL:The Industry's Foundation for High Performance Graphics*. URL: https://www.opengl.org/ (visited on 10/17/2014).

Snavely, N., Seitz, S. M., and Szeliski, R. (2006). "Photo tourism: Exploring photo collections in 3D". In: *SIGGRAPH Conference Proceedings*. New York, NY, USA: ACM Press, pp. 835–846. DOI: 10.1145/1141911.1141964.

Visual Computing Lab of ISTI - CNR (2014). *MeshLab*. URL: http://meshlab.sourceforge.net/ (visited on 10/17/2014).

Weinhaus, F. M. and Devarajan, V. (1997). "Texture Mapping 3D Models of Real-world Scenes". *ACM Comput. Surv.* 29 (4), pp. 325–365. DOI: 10.1145/267580.267583.

Weng, J., Huang, T. S., and Ahuja, N. (2012). *Motion and Structure from Image Sequences*. 1st Edition. ISBN: 9783642776458. Springer Publishing Company, Incorporated. URL: http://dl.acm.org/citation.cfm?id=2408279.

Yu, F. and Gallup, D. (2014). "3D Reconstruction from Accidental Motion". In: *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pp. 3986–3993. DOI: 10.1109/CVPR.2014.509.

Appendix A

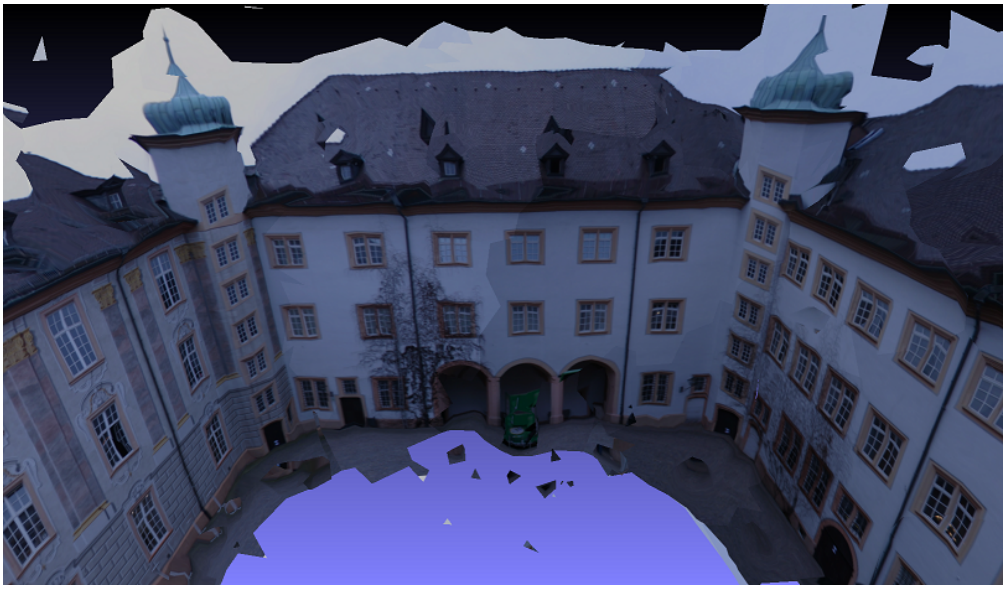

Figure A.1: Output with the castle dataset using $(0, 0, 0)$. This means each images that the facet was visible in was given a cost value of 0.

Figure A.2: Output with the castle dataset using $(1, 0, 0)$. This means the cost value was only dependent on the viewing parameter described in equation 3.

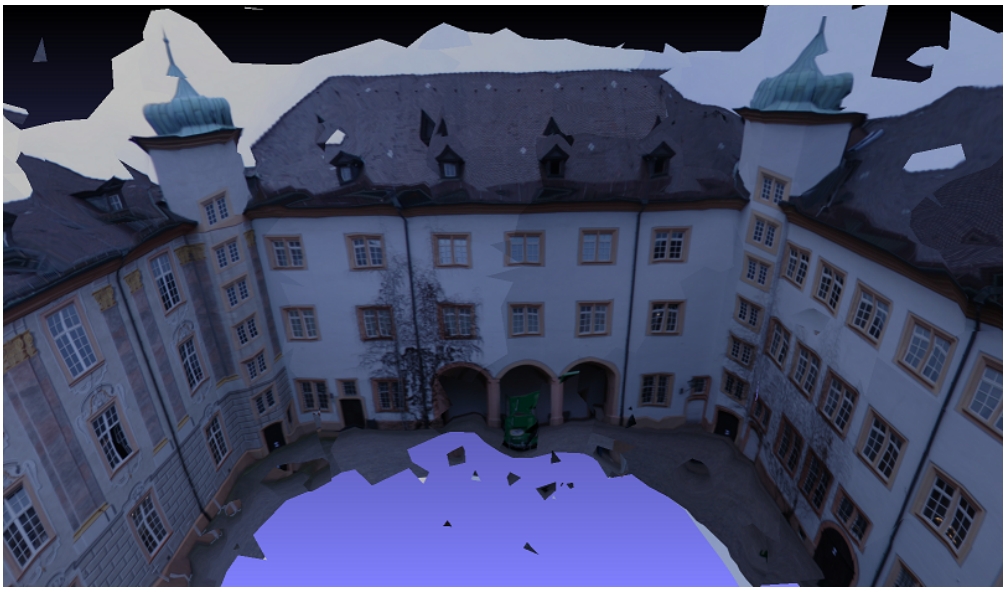

Figure A.3: Output with the castle dataset using $(0, 1, 0)$. This means the cost value was only dependent on the brightness parameter described in equation 6.

Figure A.4: Output with the castle dataset using $(0, 0, 1)$. This means the cost value was only dependent on the distance-to-center parameter described in equation 7.

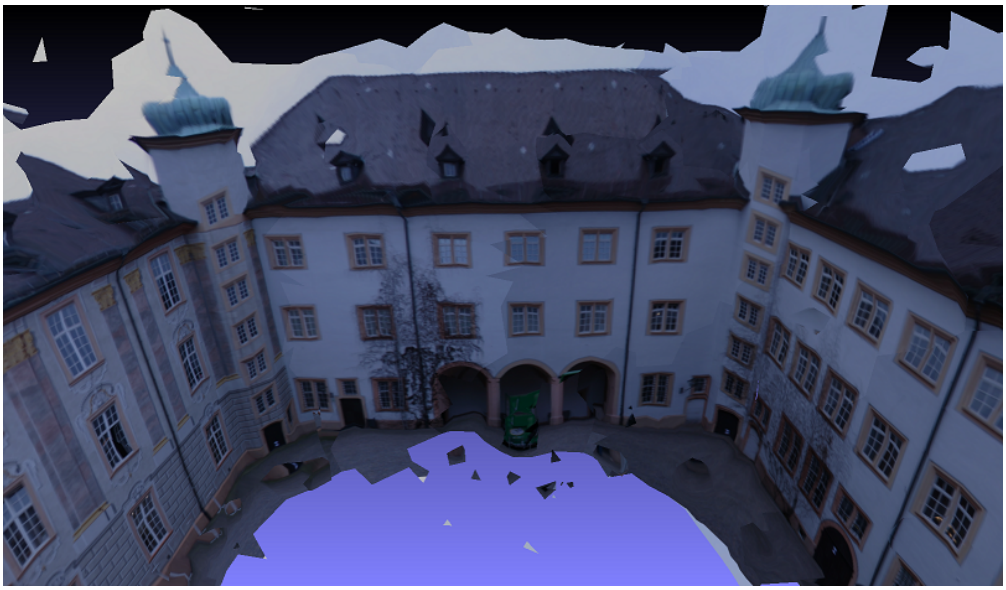

Figure A.5: Output with the castle dataset using $(1, 1, 1)$. This means the cost value was dependent on the all 3 parameters described here with equal (0.333) weights.

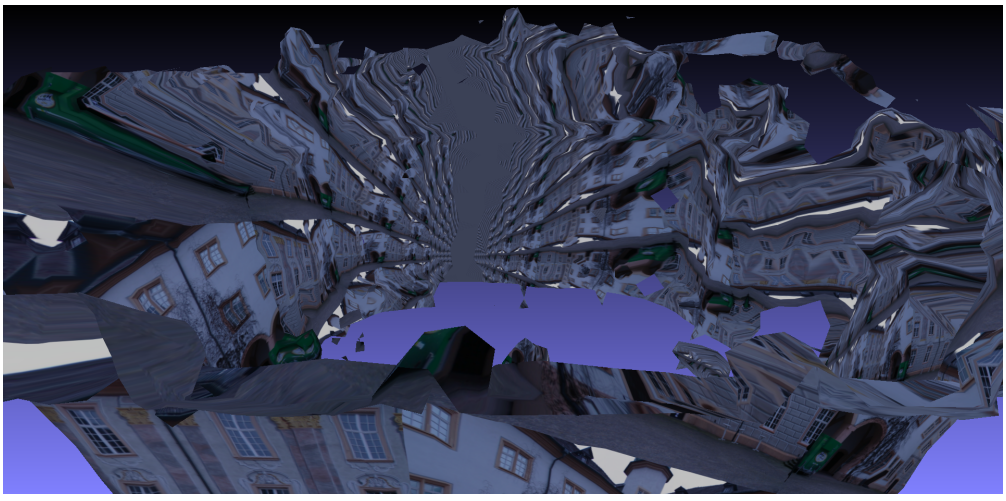

Figure A.6: Output using modified code so that the cost function for all images was the same constant value. This differs from the $(0, 0, 0)$ case because this modified version ignores whether the facet is visible in an image or not.

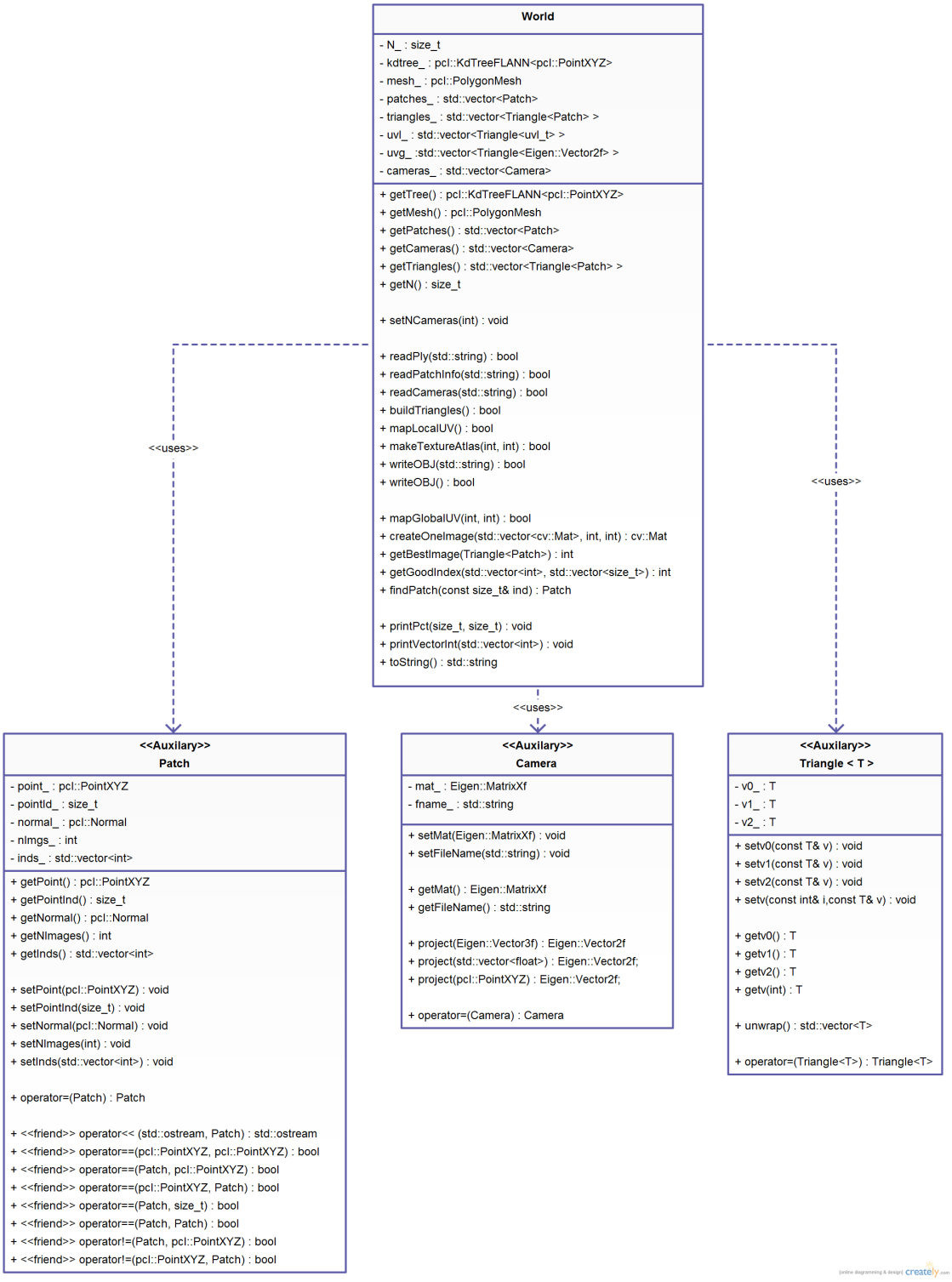

Figure A.7: UML Class Diagram of the classes used. For each class, the top rectangle is the class name, the next section is fields (variables) associated with each class, and the final section is the methods for that function. A plus (+) denotes public access specifier while a minus (-) denotes private access specifier.