3-D Stereoscopic Reconstruction using Structured Light

Sergey Weinstein
Advisor: David Martin

0. Abstract

Much like humans, computers are able to infer 3-D positions of objects using sight alone. Replacing our eyes with electronic cameras and our brain with soul-less, heartless algorithms, it is possible for a computer to reconstruct a 3-D scene from simple photos. In order to understand the relative positions of objects in the world, we can take two photos of the same scene (from different viewpoints) and look at how positions of objects change from photo to photo. To illustrate this, try extending your arm and holding it in front of your eye. Look at your thumb with only your right eye, and then switch eyes. Your thumb seems to jump a great deal between those two viewpoints, whereas the background stays in mostly the same spot. This change of relative positions allows us to understand the 3-D structure of the things we see.

The major difficulty with simulating this on a computer is the fact that its very hard to know where an object is from photo to photo (e.g. a computer doesn't know what a thumb is). The common way to identify the 'correspondences' between photos is to look at the image one small area at a time, and then trying to find a similar patch in the other image. This process is very time consuming and can result in inaccurate correspondences.

The strategy taken by this thesis involves using structured light (projecting a checker board on to the scene while photographing it.) This strategy allows the computer see the image as a binary collection of pixels rather than a multi-valued grey (or color) image. The computer can then give a unique index to each part of the scene and quickly figure out where things are in both pictures. It can then recover the 3-D structure of the scene and paste the surfaces with the original objects' textures

1. Introduction

1.1 Challenges and Overview of Procedure

Computers face three main challenges when reconstructing 3-D structures from pairs of stereoscopic images: 1) finding where things are located from image to image (correspondences), 2) estimating the original geometry of the cameras, and 3) rendering the scene. Once a computer has all this information, it becomes possible to recreate the original 3-D scene.

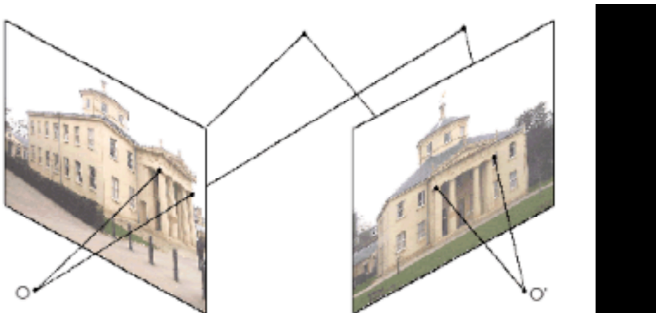


Figure 1 - Hypothetical Point Projection from a Stereo Pair

Figure 1 illustrates a hypothetical reconstruction scenario: we have two images and correspondences between them. The two planes represent the image planes of the two

cameras, and the O's correspond to the projection centers. Although the actual scene is not pictured, it is assumed that the cameras were pointing at the building and placed in the orientation as shown. This orientation (which normally must be estimated, but is given in this example) is referred to as the geometry of the cameras.

In this example we are given the relevant information (correspondences and geometry) to be able to project points out into 3-D. This projection is done by intersecting a pair of lines (for each pair of correspondences), which are defined by the center of projection and a correspondence point in each image. Here we have two pairs of correspondences that have been projected into 3-D, but the goal is to be able to do this for every possible location in the image.

Once we have a large set of correspondences and have projected them into space we end up with a large cloud of points that resemble the actual scene. The last step is to create a surface and texture for this set of points.

1.2 Correspondence

Automatic search for correspondences is a very daunting task. There are many difficulties in creating an accurate yet efficient algorithm; variables such as object texture, lighting, object foreshortening, and image noise all stand in the way of an elegant algorithm.

Two traditional methods of solving this problem are "correlation" based and "feature" based correspondence searches. Both these methods assume very little about the camera, scene, and lighting and so are quite popular.

"Correlation" based searching is based on the idea that an area of an object does not change brightness from photo to photo. Thus the image of the object should be relatively stable from image to image. The algorithm for correspondence matching involves looking at a small window centered on a pixel of interest in one image, and searching for a similar looking window in the other image. In the following figure, sample windows are shown on the left image and their possible matches on the right.

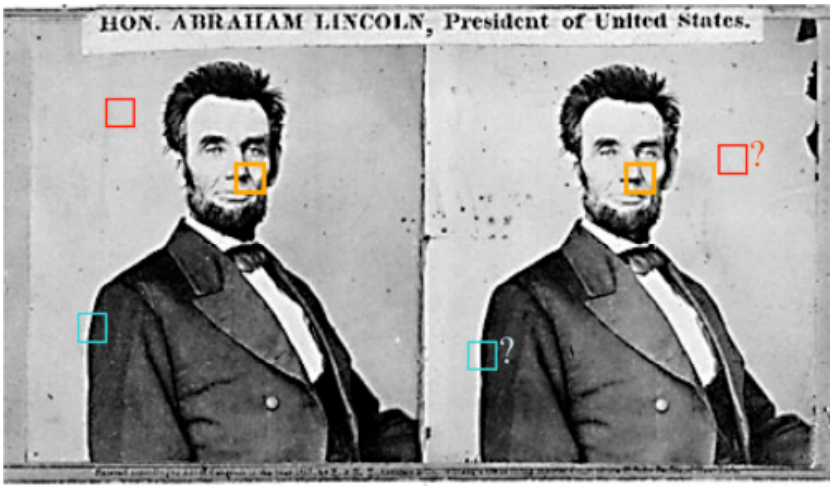


Figure 2 – Examples of Correlation Based Search Windows

The orange window centered on Abraham Lincoln's nose is easily found on the right image with this technique because it contains a lot of texture. In this case, taking the

absolute difference between the pixels in the windows does a good enough job in finding a closest match because of the unique texture pattern in the window. Problems arise, however, when we try to find matches for textureless areas such as the red box centered on an area of the wall. It is clear that there is not enough information to find a good match. Similarly, the blue window is hard to match as the texture it contains only restricts window position to a vertical line, leaving it free to slide up and down Abe's arm (this can be fixed with a larger window that captures detail about the shoulder, but variable window sizes create other problems).

The other popular technique is the feature based search. This algorithm relies on higher-order information in the image: edges, corners, shapes, etc. These are called image features and they are a bit less sensitive to noise and lighting disparities between images. Unfortunately there are usually few features in an average image, and even those features are sensitive to geometric distortions that arise from having photos of an object from different points of view.

Although both these techniques are fairly popular, they both have their faults. Correlation based search requires texture, and the certainty of results is questionable, and feature based search, although far more reliable, gives very sparse correspondences, and is useless for the actual scene reconstruction.

One thing to note is that these algorithms were devised for any image of a scene – so they make no assumptions about the lighting. If we loosen this requirement we can try another method of correspondence search that involves using structured light. That is, projecting patterns of light onto a scene while photographing it. This thesis takes this route, but also takes it one step further and shows how it is possible to be able to use special patterns to divide a scene into smaller areas and tag each location with a unique index. Thus eliminating the need to do soft searches for similar areas or features, and instead allowing the search to be concrete and based on the index number of each image area.

1.3 Estimation of Camera Geometry

The information provided the correspondences between two images allows us to estimate the original placement of the cameras. To understand how this is possible, we must take another look at the camera set up.

Figure 3 shows a similar set up as Figure 1, except the actual scene is now a single point of interest: M. To get an intuition of how we can relate correspondences to the geometry of the cameras, we consider where the right camera would be if it were in the field of view of the left camera (its image location is denoted by e in the figure). Similarly we can think of the left camera's image in the right camera's point of view (e'). This creates an immediate constraint on the geometry: the vectors Ce and C'e' must be collinear. The consequence is that Cx and C'x' are constrained to be coplanar as otherwise they would never intersect. Thus from this geometry we can assume that Ce, Cx, and C'x' are all coplanar. From this assumption we can later derive a linear transformation from x to x', for each pair of correspondences in any image thus recovering the geometry of the two cameras.

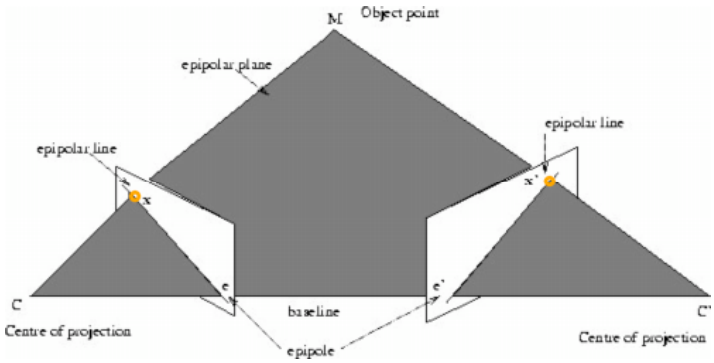


Figure 3 - Imaginary Two-Camera Set-up

The scene consists of one point M. M's location in the cameras is denoted by x and x'. C and C' are the projection centers, e and e' are the images of the cameras (the epipoles).

1.4 Rendering of Scene

After recovering the geometry and projecting the points into 3-D we get a disconnected cloud of points (Figure 4 left). The goal of the 3-D reconstruction is to make it look like the original scene, so creation of surface, smoothing of the surface, and texture application are required to adequately render the scene (Figure 4 right).

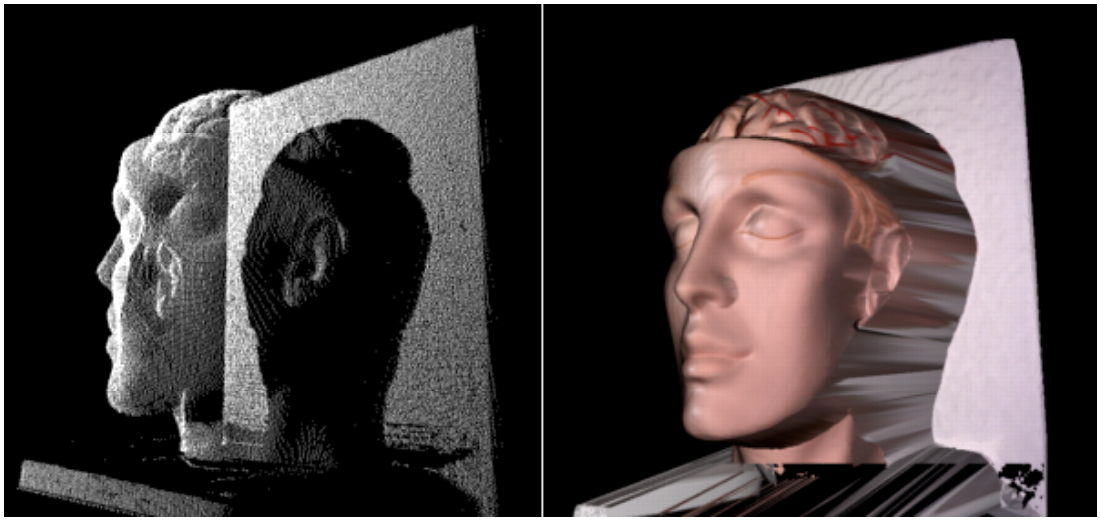


Figure 4 - Reconstructions of manikin head

2. Implementation

2.1 Correspondences

The technique chosen by this thesis is the use of structured light to get as much information about the scene as possible. This was accomplished by projecting and photographing multiple binary patterns (black and white, but no grays: Figure 5) onto the scene (Figure 6) from each viewpoint. The patterns chosen reflect the need to divide the scene into smaller areas that can then be uniquely indexed.

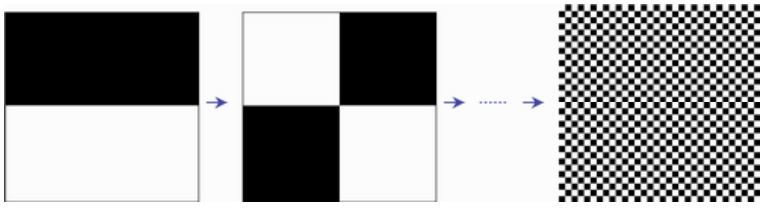


Figure 5 - Projection patterns

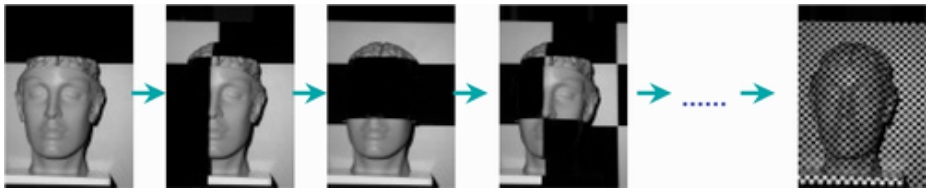


Figure 6 - Projection onto scene

The index (also referred to as the byte code) of each area can be recovered by creating a stack of photos and looking down through the stack one pixel at a time (Figure 7). We create a string of 1's and 0's corresponding to the brightness of the pixel in each photo. For this example the index of the pixel represented by the arrow would begin with 11010. This sequence of bits is treated as a binary number and is the index number for this pixel. Note that since the camera may have higher resolution than the projector, many pixels may have the same index value (but they should all be in a small cluster).

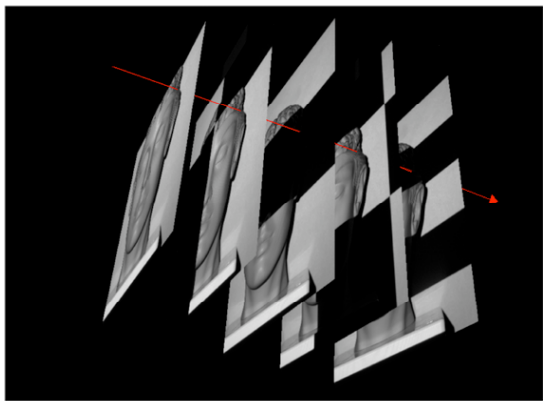


Figure 7 - Recovering the Index of a Pixel

To make this indexing method more robust it is necessary to remove object reflectance as a variable. Since the brightness of each pixel is the product of object reflectance and incident light; for each pattern, projecting its opposite and subtracting the images isolates the resulting incident light value (Figure 8).

The negative values in the difference image represent those pixels that were unlit in the first image, positive those that were lit, and values near zero represent areas of the scene where the projected light has very little influence (shadows). This difference image

lets us see which pixels can be ignored in the correspondence search (those whose values are near 0) thus saving computation and increasing fidelity of the index values. The check to ignore any pixel is seeing if any of its values is near 0 in any of the difference images. An illustration of what the algorithm ignores is the left image of Figure 9 – the black areas indicate pixels that are considered shadows.

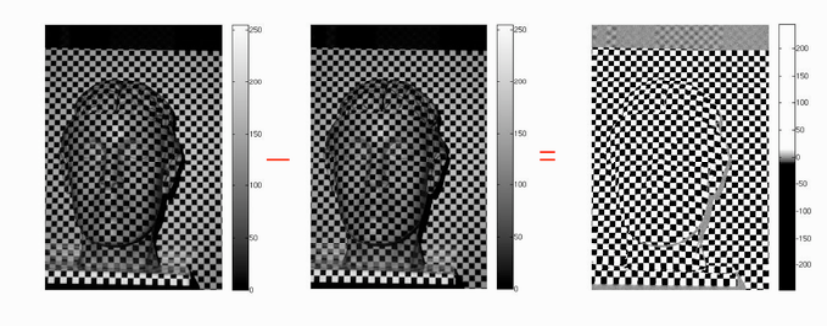


Figure 8 - Difference of Inverted Patterns

The range of brightness is 0-255 in the left two images (the actual photos), but the difference ranges from –255 to 255 in the rightmost "difference" image

The process of pixel indexing is repeated for every pixel in the image, and we end up with a 'byte code' image pair (one for each camera), where each pixel's value is the index number generated by the stack of difference images.

With the indexing complete, the next step is to actually find the correspondences. The literal search algorithm (finding the same byte code) is discussed in Section 3 in detail, so, for now, assume the computer can find sets of pixels of the same byte code in each viewpoint efficiently. Ideally, the neighborhood of a pixel does not matter since we just care about its index number, so the search should just involve finding clusters of pixels of the same index value in each image and having each cluster's center of mass be the correspondence location. Unfortunately image noise and object reflectance can never truly be factored out, so information of the neighborhood of each pixel is still required. The information we desire is by how many bits is a pixel different from its neighborhood. That is, if a pixel's byte code is 0000001 and all of its immediate neighbors (there are 8 of them) are byte code 1001011 then we say that this pixel differs by 3*8=24 bits from its neighbors. We call this value the error.

This process is done in general by looking at a square window of some size around a pixel (checking only immediate neighbors, for example, implies a 3x3 window with the pixel of interest in the middle). All the points in the window are XORed with the center pixel's index value, and the sum of the bits set to 1 after XORing is calculated. This is what will be referred to as the "XOR filter." An example of running the XOR filter on a byte code image is shown in Figure 9. Red denotes large differences between a pixel and its neighbors, and blue implies minimal difference.

.

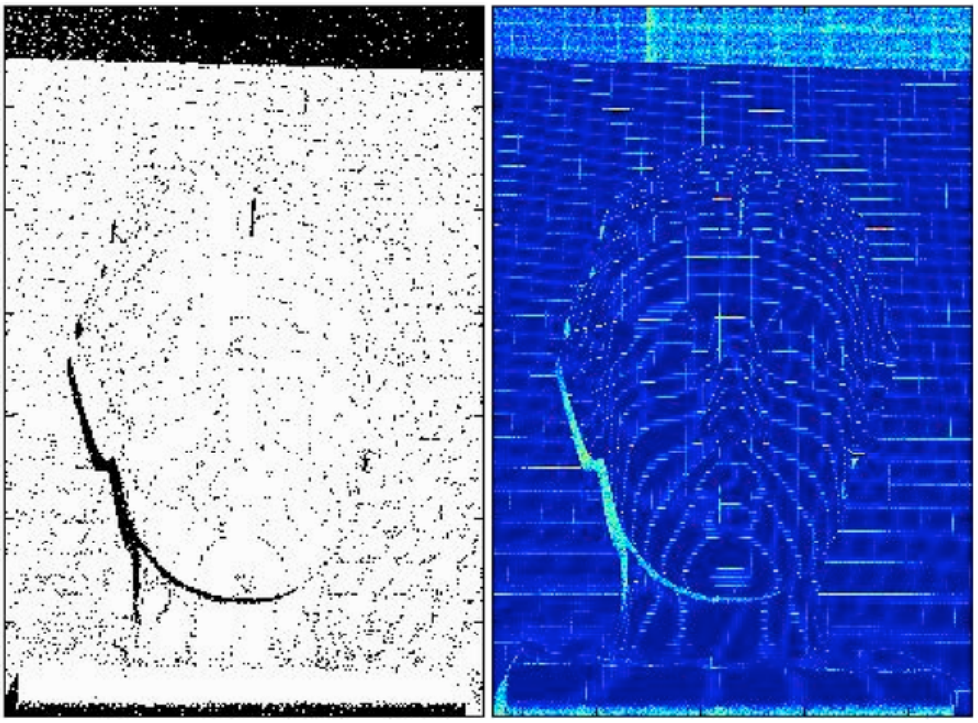


Figure 9 – Threshold "shadow mask" and Image of Error Filter with a 3x3 window

This error filter lets the computer decide much more accurately if it gets a false positive for any given byte code. Since the size of each cluster should be the size of the camera's image of the smallest area the projector projects onto the scene, all pixels of some byte code should theoretically be in the same small area. If you get false positives then the idea is that they are surrounded by pixels of differing byte codes, so the error filter would give them a high error value. If the computer sees a lot of pixels with the same byte code, it will ignore those with high error. Through experimentation, it was discovered that using a window size of 3x3 and a maximum error of 3 worked best in eliminating false positives.

For one more level of precision, the algorithm implemented by this thesis strictly enforces a maximum radius for a cluster of pixels. This is done by considering the total number of pixels within the error filter threshold and estimating the radius of the circle that would circumscribe them. The algorithm then finds the center of mass of all these pixel locations, and throws away all pixels that are further that this radius from the center. The center of mass of these remaining pixels becomes the correspondence location. This is illustrated in Figure 10.

Here all the locations marked with a color have the same byte value. The blue locations represent those pixels whose error was too high, the green pixels represent the pixels with the correct byte code and small error, but which were outside of the maximal radius. The red pixels are those that have fulfilled the error and radius requirements and their center of mass is marked yellow. This yellow dot is the correspondence location for this byte code.

This process is repeated for all byte codes present in the image, and the set of correspondences composed of the center of masses of all of these byte code clusters. The complete flowchart of the correspondence algorithm is illustrated below.

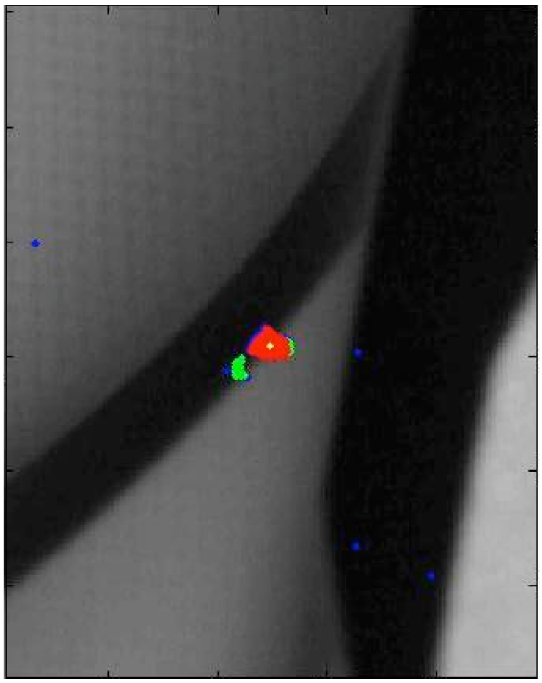
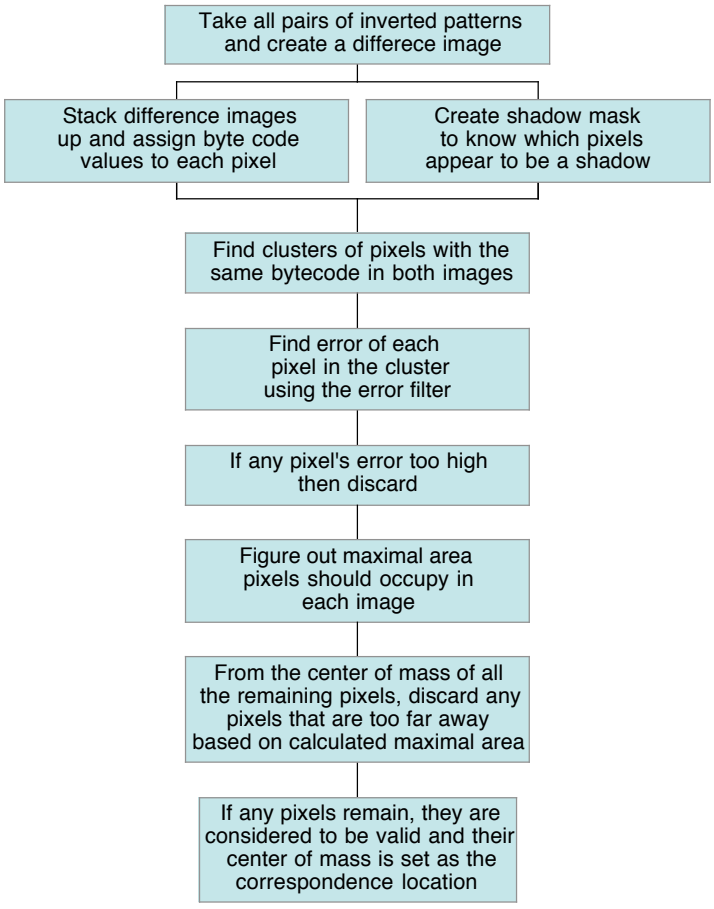


Figure 10 - Blue, red, green and yellow pixels have the same byte code

Correspondence Search Algorithm



2.2 Estimating Camera Geometry

This section explains how the estimation of camera geometry was performed using the correspondences that were attained by the methods outlined in the previous section. Most of the algorithms and equations used here were taken from outside sources. [1],[2] The particular method of geometry estimation is called the "eight point" algorithm. A run through of this algorithm follows.

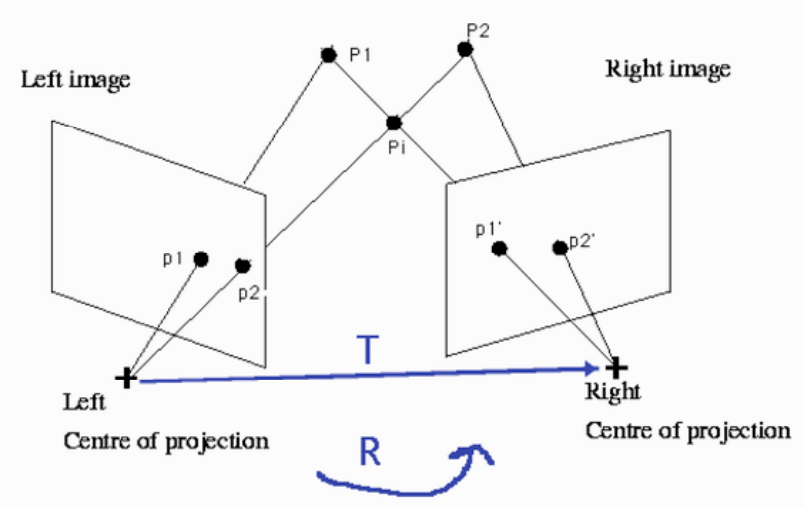


Figure 11

Figure 11 shows a similar camera set up as Figure 3. It can shown that the immediate consequence of knowing the geometry allows us to set up a relation between the left and right views of a point Pi in the scene:

$$P' = R(P - T) \tag{2.2.1}$$

We can transition from world to camera coordinates p, with this relation:

$$p = \frac{f}{Z}P$$

Where f is the focal length of the camera and Z is the depth coordinate of P. As we saw in the introduction, T, P, and P' are coplanar so we can express them in this way:

$$(P - T)^{\top} T \times P = 0$$

using equation 2.2.1:

$$(R^{\top} P')^{\top} T \times P = 0 \tag{2.2.2}$$

Expressing T as vector product:

$$T \times P = SP$$

where:

$$S = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

so now 2.2.2 becomes:

$$P'^{\top} EP = 0$$

$$E = RS$$

we can convert to camera coordinates by dividing by $Z'Z$ to get:

$$p'^{\top} E p = 0$$

Now we have a relationship between the camera geometry and pairs of correspondences in camera coordinates. We estimate *E* using all our correspondences at once if we put all our correspondences in the rows of a matrix and estimating E using least squares:

$$\begin{bmatrix} x'_i & y'_i & 1 \end{bmatrix} \begin{bmatrix} e_1 & e_2 & e_3 \\ e_4 & e_5 & e_6 \\ e_7 & e_8 & e_9 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} x'_1 x_1 & x'_1 y_1 & x'_1 & y'_1 x_1 & y'_1 y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_n x_n & x'_n y_n & x'_n & y'_n x_n & y'_n y_n & y'_n & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \end{bmatrix} = 0$$

By construction E should be rank 2, so we enforce this by setting its smallest singular value to 0. [2]

Using this E and normalizing it by the length of T (denote this E with Ehat) we get:

$$\hat{E}^{\top} \hat{E} = \begin{bmatrix} 1 - T_x^2 & -T_x T_y & -T_x T_z \\ -T_y T_x & 1 - T_y^2 & T_y T_z \\ -T_z T_x & -T_z T_y & 1 - T_z^2 \end{bmatrix}$$

It can now be seen that factoring T out of this can be done in three ways (one for each column). In this thesis we try all three ways, and take the T whose length is closest to 1. We know this T up its sign.

Once we have T it is possible to find R with three vectors:

$$w_i = \widehat{E}_i \times T$$

where i=1,2,3 corresponding to the rows of Ehat.

So to get rows of R:

$$R_i = w_i + w_j + w_k$$

for all i,j,k being all permutations of (1,2,3). One remark is at this point we only know E up to a sign difference, so we can get two values of R for each T, since we also know T up to sign, we get a total of two possibilities for T and a total of four possibilities of R.

To find out which of these four combinations is correct, we try projecting out a set of correspondences (as in Figure 1). Since the scene takes place on the positive Z axis of each camera (and assume all of the objects are at least 1 focal length away from the camera), we assume that the set of R, T that projects the most points to positive Z, is the best approximation to the geometry of the cameras. Once we find this pair R, T, we will use it to project all of our correspondences into 3-D.

2.3 Rendering

Now that we have the projected points (Figure 4 left) we must generate a surface and texture to this scene. The methods in this thesis assume that all objects in a scene are connected by some surface. Although this assumption is not valid all the time, application of texture usually removes most of the artifacts from having an imaginary surface. Since the gaps between disconnected objects are the color of the background, the texture in those areas makes the surface less noticeable.

The creation of the polygon vertices for the surface is done with a built in Matlab function called Delaunay triangulation. This method creates triplets of points (triplets are designated by three indices into our array of points) based on their proximity to each other in a plane. This method cannot create a true 3-D surface (as it creates triplets in a plane), but all the 3-D information that can be acquired using stereopsis can be projected onto a plane without overlapping so we don't lose information. Conveniently, this plane happens to be any plane perpendicular to the Z axis.

To create a surface, we ignore the z coordinates of all of our points, apply delaunay triangulation to this set of points, and use the triplets of points as vertices for polygons which will become the surface for our scene (Figure 12).

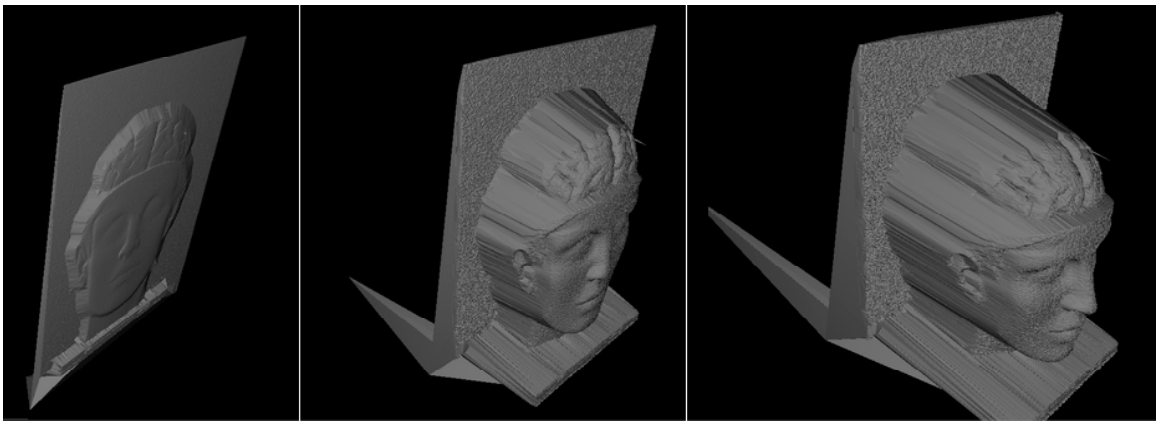


Figure 12 - Surface application using Delaunay triplets (from plane to 3-D)

As evident from the same figure, the surface highlights the sampling resolution and projection error. The bumps are not from the scene, but artifacts from projection. In fact, Figure 12 is a particularly error-free projection, the error is usually more like that of Figure 13 left. This example has many "spikes" which are results of a few points that were badly projected. The goal of rendering is to reduce these errors with filters.

To get rid of the most egregious artifacts, a median filter is used on all the points in the data set. This algorithm for this filter is as follows:

-For each point p and the polygons it's a vertex of:
    -Find the set of points connected to it by an edge (call this p's set of neighbors)
    -Find the median x, y, z coordinates of all the points in this set
    -Set p's coordinates to these median [x y z]

This algorithm is made more robust by increasing depth of the neighbors of p (i.e. depth 2 would include all the neighbors of p's neighbors).

This median filtering works well when smoothing out large errors as those in Figure 13 left, but as the middle rendering shows, this filter is not effective with small errors.

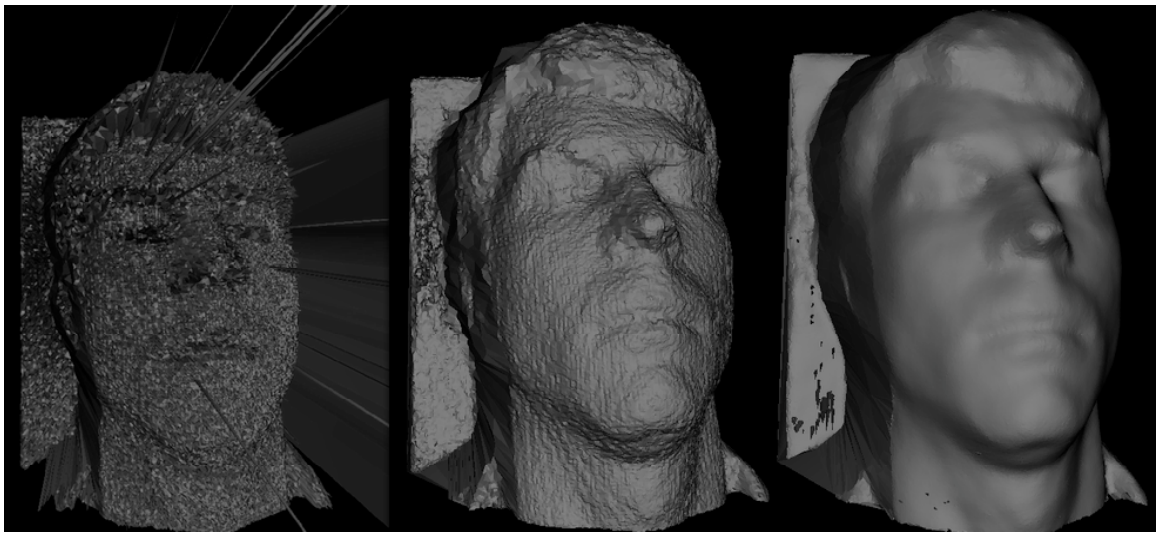


Figure 13 - Unfiltered, Median Filter (Level 3), Median then Mean (Mean Level 4)

The next step of filtering is to take a smart averaging of the points. We call this mean filtering. The assumption of this filtering is that there are no large discontinuities on the surface. From this assumption we take a weighted average of locations of a point's neighbors to find its new location. The algorithm is as follows:

-For each point p:

-Find the set of points connected to it by an edge (p's set of neighbors): $c_1, c_2, \ldots c_k$

-Find the median coordinates of these points: $[x_m\ y_m\ z_m] = c_m$

-p's new coordinates = $\dfrac{\sum_{i=1}^{k} w_i c_i}{\sum_{i=1}^{k} w_i}$

where:

σ = mean distance from c_m

$$c_i = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}$$

$$d_i = \sqrt{c_i^2 - c_m^2}$$

$$w_i = e^{-d_i^2/\sigma^2}$$

The results of this filtering (with neighbors up to depth 4) are on the right of Figure 13.

Now that the surface is smooth, we can complete the last step, applying the texture. Fortunately since each vertex of the polygons that make up the surface has been projected from a pair of 2-D image coordinates, we can use this information to easily apply texture directly from the image. Figure 14 shows the result.

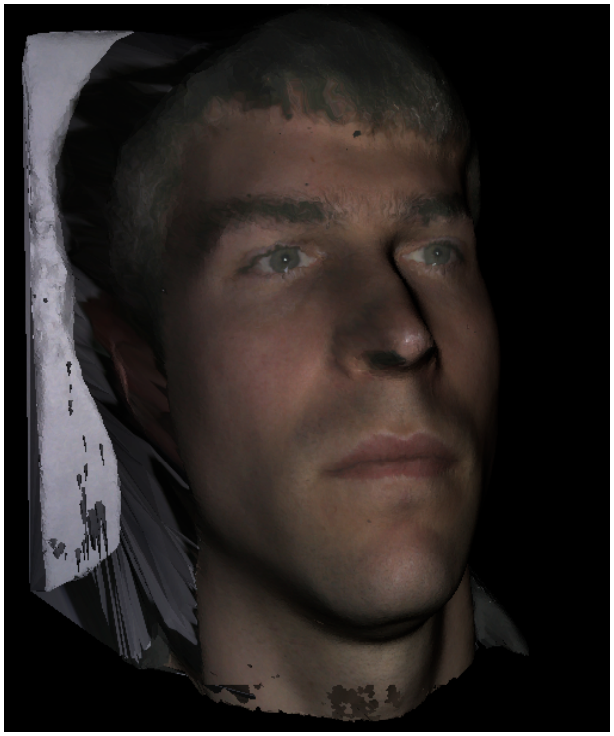


Figure 14 - Texture applied to smoothed surface

3. Discussion and Results

3.1 Correspondence Search Algorithm

Although the technical section outlined the final flow of the correspondence search, it is a result of incremental progress and much trial and error. The final algorithm for correspondence search was worked on heavily before it became efficient. The high resolution of the images posed a problem early on in the project due to the amount of data.

One of the more significant features of the correspondence search is that it operates around O(nlogn) time – a great thing since the images have over 7 million pixels each. This is accomplished by the fact that the search for clusters of the same byte code is done after flattening each image into a vector (keeping track of the original locations of each pixel) as shown in Figure 15. The two vectors of byte codes (one for each viewpoint) were then sorted in descending order and the search for byte codes was simplified to simply scanning from left to right in each vector and looking for runs of the same byte codes.

The efficiency of this algorithm was also increased by the fact that occlusions – areas of the scene that are visible from only one viewpoint – are handled gracefully. Since byte codes corresponding to the occluded area only appear in one vector of byte codes (since they are sorted, easy to see that they are missing), these values could easily be skipped rather than searched for.

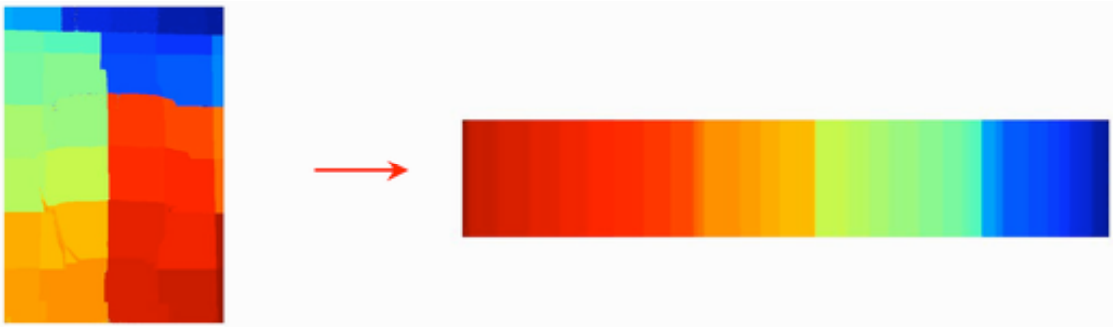


Figure 15 - Flattening byte code image into vector

The colors indicate the byte value

3.2 Rectification and Epipolar Lines

Further constraints on the location of correspondences exist. For example, in Figure 16, for every correspondence in the left image, the coplanar constraint requires the right correspondence appears somewhere along the line (if we don't yet know where the object is in 3-D that is) that is the intersection of the image plane and the epipolar plane – this line is called the epipolar line. So if we manage to have enough trustworthy correspondences (at least 8 by our estimation of R, T) it is possible to get even more correspondences (hopefully more accurate ones) by repeating our correspondence search, but this time adding the constraint that all correspondences must lie in a line in both images. This is a bit difficult to implement, however, since this line does not usually lie along any row or column, but has nonzero slope, so searching along an arbitrary line in an image is a bit time consuming when there are over 7 million pixels.

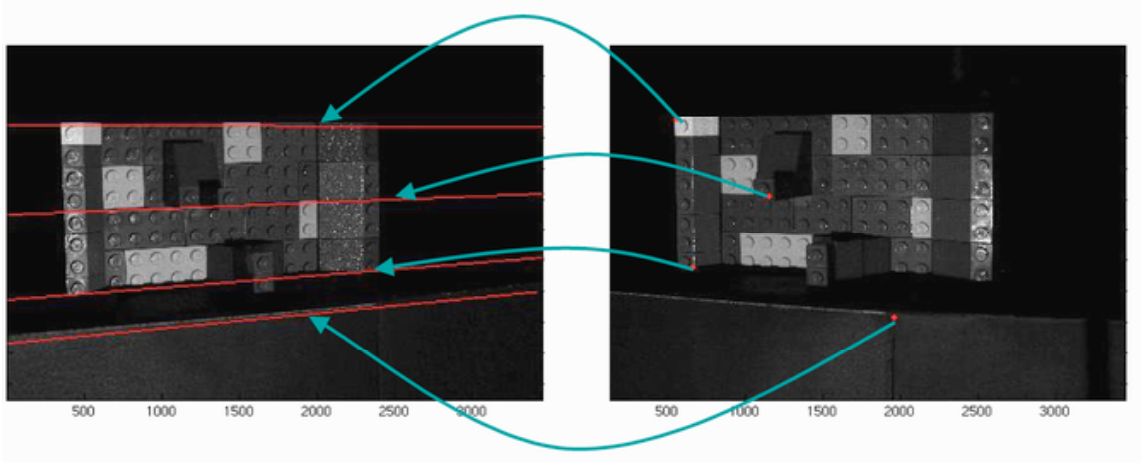


Figure 16 - Examples of epipolar lines

Even if can efficiently search along lines of nonzero slopes, this method is still a bit involved, because although we know that for every point in one image, its correspondence in the other image lies along some line, we never really know from which image we get this initial correspondence; both images have clusters of pixels with the same byte codes which are possibly very spread out (pixels of same byte codes can be far away from each other) so the only real restriction we can make is that our correspondences should lie along some band of considerable width in both images.

The popular solution to the slope problem is to rectify the images. What this means is that we want to simulate what the images would look like if our cameras were both pointing in a direction perpendicular to T (and parallel to each other). What this does is it makes the epipole of both cameras infinitely far away, and thus the epipolar lines become not only parallel in both cameras, but exactly horizontal in both images (even along the same y coordinate if both cameras have the same focal length). If done successfully, this would simplify implementation of this epipolar line constraint significantly.

We tried implementing this but after much difficulty (methods described by [1] were surprisingly brittle when there was error in R, T) we were only mildly successful. As you can see in Figure 17 the images appear to be rectified. This should imply that every pair of correspondences lies on the same row of both images. In this example, this holds true for the bottom area of the images, but as you go higher up, the disparity between the locations of the correspondences increases. This is most likely due to an error in the estimation of R, T so rectification process ended up compounding on this and made the error more egregious. So our constraint of the y value for correspondences is quite a bit weaker than theoretically supposed and we abandoned the use of rectification (we were at the ‘band’ restrain again with imperfect rectification). Of course good examples of rectification are possible, and example is Figure 18.

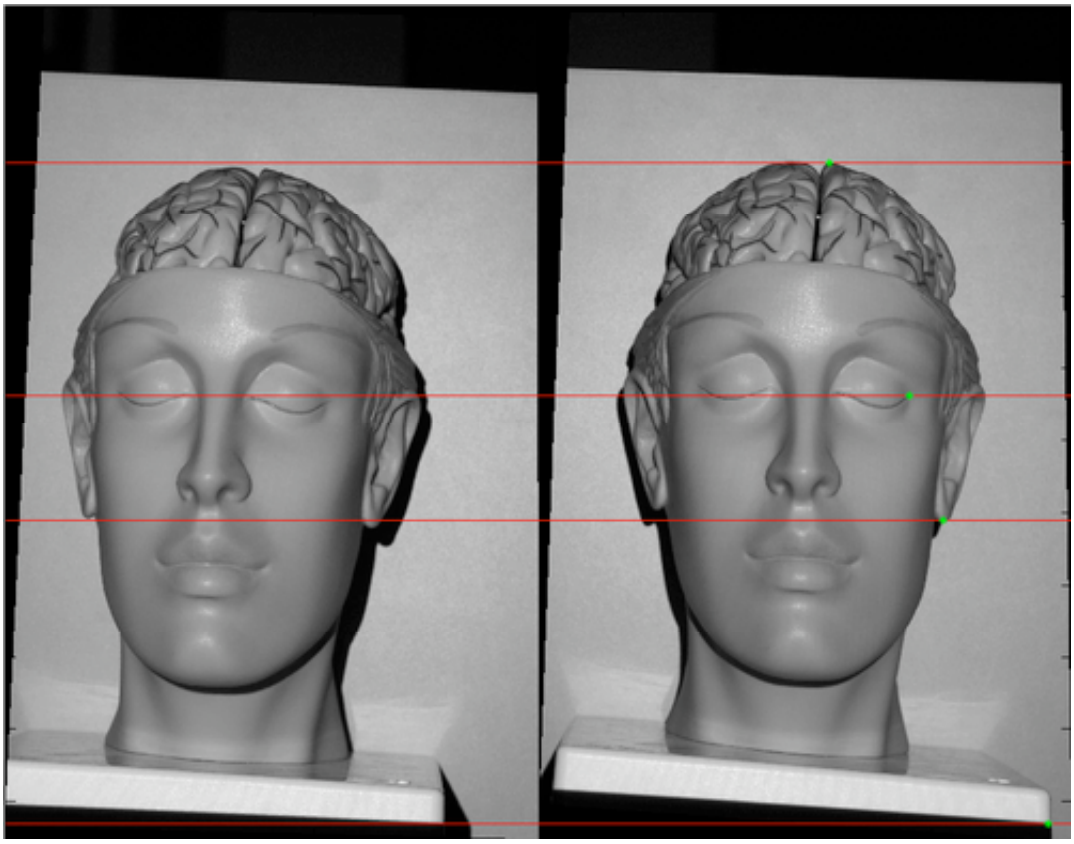


Figure 17 – Unsuccessfully Rectified photos of manikin and epipolar lines

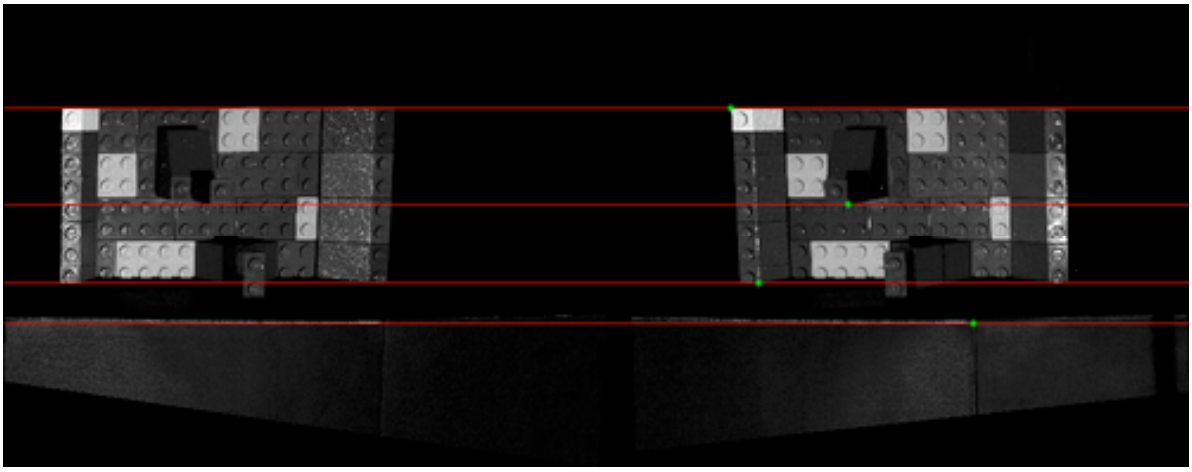
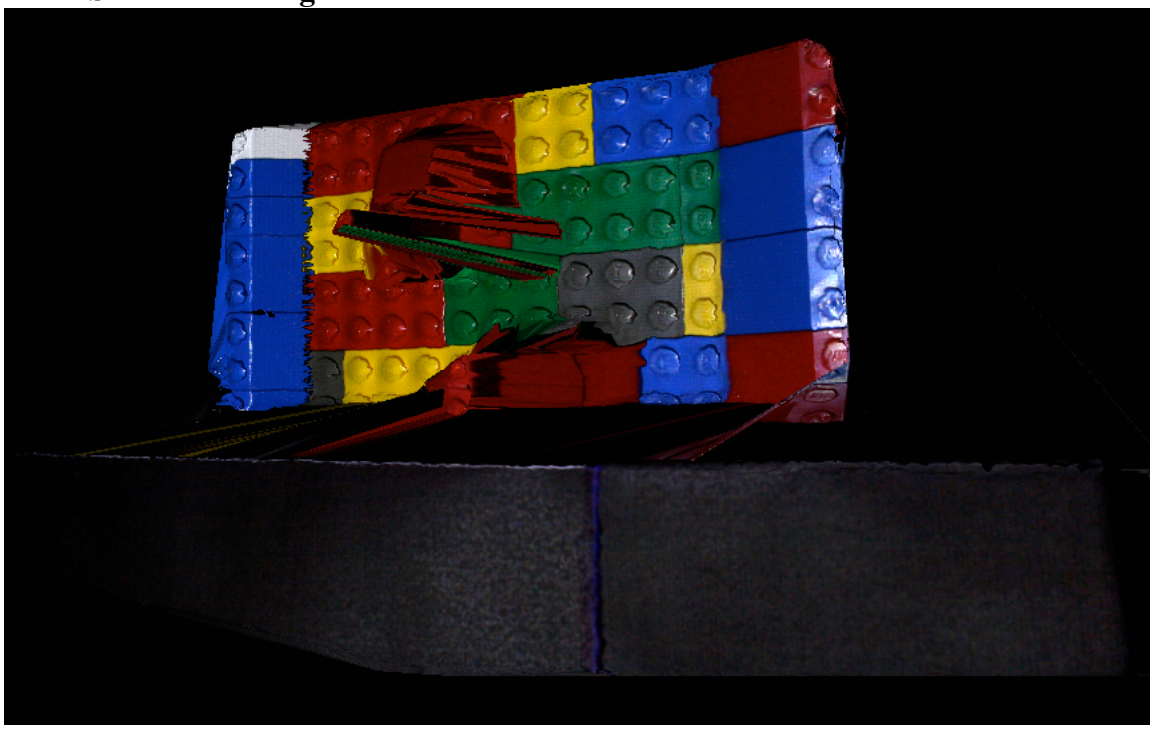


Figure 18 - Successful Rectification

3.3 Some Rendering Results



Bibliography

[1] Emanuele Trucco, Alessandro Verri. Introductory Techniques for 3-D Computer Vision. New Jersey: Prentice Hall, 1998

[2] Richard Hartley, Andrew Zisserman. Multiple View Geometry in Computer Vision. Cambridge, UK: Cambidge University Press, 2000

H.C. Longuet-Higgins, A computer algorithm for reconstructing a scene from two projections. Nature, 293:133–135, Sept 1981.

R.I. Hartley, "In defence of the 8-point algorithm," in Proc. International. Conference on Computer Vision, 1995, pp. 1064 – 1070.

O. Faugeras. Three-dimensional computer vision. MIT Press, 1993.