Generation of Orthogonal Projections from Sparse Camera Arrays

Ryan E. Silva

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

Roger Ehrich, Chairman

James Arthur

Denis Gracanin

Francis Quek

May 17, 2007

Blacksburg, Virginia

Generation of Orthogonal Projections from Sparse Camera Arrays

by

Ryan E. Silva

Committee Chairman: Roger Ehrich

Computer Science

(ABSTRACT)

In the emerging arena of face-to-face collaboration using large, wall-size screens, a good videoconferencing system would be useful for two locations which both have a large screen. But as screens get bigger, a single camera becomes less than adequate to drive a videoconferencing system for the entire screen. Even if a wide-angle camera is placed in the center of the screen, it's possible for people standing at the sides to be hidden. We can fix this problem by placing several cameras evenly distributed in a grid pattern (what we call a sparse camera array) and merging the photos into one image. With a single camera, people standing near the sides of the screen are viewing an image with a viewpoint at the middle of the screen. Any perspective projection used in this system will look distorted when standing at a different viewpoint. If an orthogonal projection is used, there will be no perspective distortion, and the image will look correct no matter where the viewer stands. As a first step in creating this videoconferencing system, we use stereo matching to find the real world coordinates of objects in the scene, from which an orthogonal projection can be generated.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Ehrich, for the initial idea for this project and for giving me the tools I needed to dive into a completely new area of computer science. Thank you for listening to my ideas, and for providing help when I hit roadblocks. I also appreciate the use of your equipment, which helped me collect data for the project. Finally, the thesis edits you've given me have been crucial to creating a good document.

Thank you, Dr. Quek, for your valuable advice in the field of computer vision. I appreciate your interest in my project, and your willingness to listen to my questions and problems. Thank you also for pulling me into the VIS Lab, where I met people who became not just valuable resources but also friends.

Thank you to the VIS Lab, for making my last months at Virginia Tech much more fun. It's not enjoyable to research solo, and it makes a big difference when there are people with whom to share research and results. Thanks specifically to Bing and Fran, for being receptive to my questions and making an effort to help solve my problems. Thanks to everyone for providing a great community within the department. I only wish I joined the lab sooner!

Of course, I don't know where I'd be without my fiancée Anna-Laura. You've given me so much moral support, and done everything you could to make my work easier. Thank you for all the ways you've helped me.

Finally, I'd like to thank my parents, Marcos and Nancie, for instilling in me a strong value on education. I wouldn't have gotten this far without their help.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

Videoconferencing provides a way for two physically distant parties to communicate in a way that imitates face-to-face communication. But the effectiveness of videoconferencing as a communication tool varies based on the type of video. If the video image is too small to record anything but the head and shoulders, communication is limited because the gestures and body movement captured by a large screen convey crucial information. Even when using a graphical model of a person, embodied representations which communicate using gestures are much more effective than their non-embodied counterparts [CBVY00]. Thus, a videoconference with a large screen will be more effective than a small single webcam setup. If we made it even larger and used a wall-sized screen, there would be plenty of room for high resolution video of the full bodies of all conference participants. Furthermore, there would be plenty of space for collaboration using other software applications at the same time, which is difficult to do on a small screen [SS97].

The end goal of having a real-time videoconferencing system on a large screen display has implications on the type of video we can use. The video isn't limited to the small area in front of a computer—it now captures the whole room. People are free to walk around the room and as close to the screen as they like. With that in mind, one camera at the center of the display is insufficient. The focal length of the lens would never be wide enough to capture people standing near the side of the screen, and

people near the sides who are captured would be viewed from the side instead of from the front. Also, people watching the video from the sides would be viewing from the wrong viewpoint.

Since one camera is not enough to support a videoconference on a wall-sized screen, my solution is to mount an array of cameras on the screen and create an orthogonal projection from the resulting images. Since the large display here at Virginia Tech is made from individual monitors paneled together, my cameras may be mounted on the bezels between the monitors. We start by considering the problem of creating the orthogonal projection and hope that it can be optimized and multi-threaded in the future with the end goal of creating a wall-sized orthogonal projection in real time.

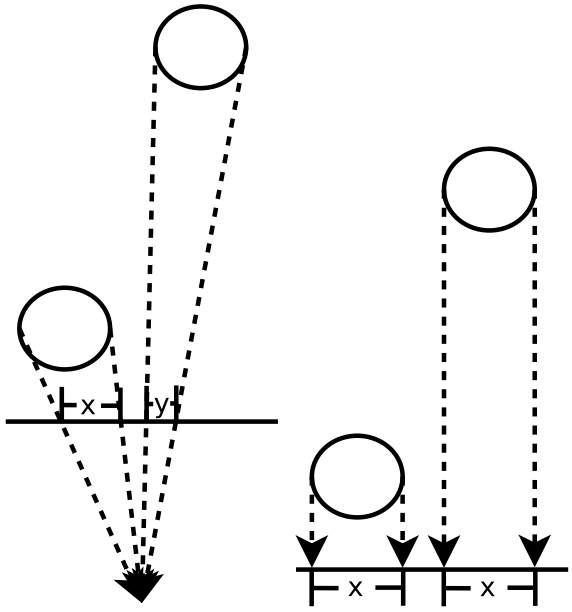
To better visualize an orthogonal projection, imagine a white wall in front of a scene. To color a single point on that wall, start at that point and move along a line orthogonal to the wall. Eventually that line will intersect an object in the scene. Color the original point on the wall the same color as the point on the object that was hit.

In an orthogonal projection, there is no perspective distortion. That means that objects do not shrink with distance from the camera (see Figure 1.1). If we were photographing a scene with a large depth range, an orthogonal projection would look strange, because objects very far away would be expanded, and nearby objects would seem compressed [RGL04]. But since our setting is an indoor room, the maximum depth of the scene is limited, and we believe that an orthogonal projection will still be useful.

The goal of the work reported here is to locate videoconference participants at different spatial positions and to determine orthogonal projections of them using a camera array. Accuracy of depth measurements is only a secondary consideration, since different subjects are rarely at close spatial positions.

1.2 Gathering Data

To simulate a scenario of people in front of a large display containing an array of cameras, we needed to do some improvising. We decided against using a real camera array; we'd have to find many cameras, and we'd have to worry about camera calibration and slight variations in the images from different cameras. In our first attempt, we used a copy stand to hold a single camera. The



(a) Perspective projection (b) Orthogonal projection

Figure 1.1: In the perspective projection, the projection of the closer circle is larger than the projection of the far circle, because the projectors (arrows) all intersect at the center of the camera. In the orthogonal projection, all of the projectors are orthogonal to the image plane, resulting in no perspective distortion. Both circles project as the same size, independent of their depth.

copy stand allowed vertical camera translation to specific measured points. To move horizontally, we translated the entire copy stand along a measured board. With this setup we are able to simulate the kind of camera array that would be used in our large display.

Later, I discovered that the vertical translation of the copy stand wasn't very precise. Vertically translating the camera resulted in a slight horizontal pixel shift, which was enough to cause errors in the results. To help solve this problem, I built a wide aluminum stand on which to mount the camera (Figure 1.2). The stand had three rigid parallel horizontal bars, 10 inches apart above each other. Using a tripod base connected to a clamp, a camera can be positioned at a precise location on one of the bars. Then the clamp can slide along the bar horizontally and move to a precise location on a lower or higher bar for vertical translation. As long as the camera is initially aligned to point perpendicular to the bars, the camera will maintain the same alignment for all camera positions.

Figure 1.2: The aluminum stand I built to take the pictures for the camera array

The most difficult problem with using the aluminum stand was aligning the camera. In this attempt, I took much more care to make sure the camera was pointing perpendicular to the translation axis, because if it wasn't, it would change the assumed location of objects relative to the camera. To align the camera, I first aligned the stand itself to be parallel to the rear wall. I did this by moving each side of the stand the same distance from the wall. Then, I projected a vertical laser beam toward the wall and used a carpenter's square to make sure it was perpendicular to the wall. I moved the camera so that the beam ran through the center of the camera and then rotated the camera about the vertical (y) axis until the beam on the wall was in the center of the image. The camera didn't need to be rotated about the x-axis (running left to right) because the camera's mount was already flat.

Since only one camera was being used for all of the images, we couldn't photograph anything that might move, precluding the use of people as subjects for our photos. Instead, we bought pieces of PVC piping and placed them upright in the scene. The PVC piping provided good pedestals on which we could place objects to photograph and allowed us to position the objects at different depths

and heights. In our first several experiments, we photographed various balls on top of the pipes. The balls in a sense imitate a person's head. In a later experiment, we taped photos of peoples' faces to the pipes. This provided less reflective surfaces and better stereo matching points. It also more closely modeled an actual person in the scene. Figure 1.2 shows the copy stand and the heads on PVC pipes.

Figure 1.3: Our setup for gathering data includes the copy stand in the foreground and PVC pipes with photos of faces taped on top.

1.3 Overview of Algorithms

To create an orthogonal projection, I started by creating a stereo matching algorithm. By finding corresponding points in adjacent photographs, I can determine the real location of that point in the scene (the x, y, and z coordinates). In theory I can find the real location of any point which appears in more than one image in the camera array. Then for each point with the same (x, y) model coordinates, I would use the point with the smallest z-value in the orthogonal projection.

Another way to solve the problem is to think about it in reverse: for each point in the orthogonal projection, find the color. I created an algorithm which does this. It relies on finding good matches of corresponding points in stereo images and on the scene being void of duplicated objects.

In reality, not every point in our scene can be unambiguously located in multiple images. Some surface points are difficult to match because the surface is specular or lacks distinctive features. Other points lie on borders between objects of different depths, so those points appear different from different cameras in the array. I implemented a simple area-based matching algorithm to determine which model points have good matches and which don't.

I also implemented a simple edge detection algorithm which helps locate good matching points. A plain, featureless surface would be a poor place to do stereo matching, but places with a very large luminance differential are great for matching because they have a unique structure that can be found in another image. Edge detection finds those areas with a large luminance differential.

Finally, I implemented an algorithm which computes a depth map for each camera image and attempts to stitch together a single orthogonal projection from those depth maps. It takes advantage of the camera array by using information from several images to create each depth map.

Chapter 2

Literature Review

After thorough review of published literature, we could not find any previous research that created an orthogonal projection from a sparse camera array. However, stereo imaging is extremely well researched, and we focus on those techniques. The area of light field imaging was very intriguing, but unfortunately that method required dense camera arrays.

2.1 Light Fields

We originally thought that light field imaging might help solve our problems. Informally, a light field is "radiance along rays in empty space" [Lev06]. To create a light field, we would use multiple images of a scene. Each pixel in each image is essentially a ray of light passing from the camera's eye to an object point in the world. By finding these rays for all images from the camera array, we would have a large set of light rays. Then we could create an arbitrary perspective projection using those light rays. This method is attractive because no information about the 3D scene model is needed and no matching needs to be done to create a 3D model. The final image is simply a result of linearly resampling the rays in the light field [LH96].

Once we further researched light field imaging, we realized that it would not solve our problem. We're trying to create an orthogonal projection, and only one light ray from each camera is an orthogonal ray—the rest are at an angle less than 90°to the image plane. In order to reconstruct

an orthogonal projection, we would need an extremely dense array of cameras. We would use the center pixel, and interpolate the rest of the pixels. The denser the camera array, the more trivial the task; but without a very dense array, quality will suffer because most of the pixels in the orthogonal projection would be interpolated.

2.2 Stereo Matching

When we decided that light field imaging wouldn't be useful, we decided to research stereo matching. Stereo matching is the process of finding points in two images which are both projections of the same point in the scene. The depth of that point determines the disparity between the points of projection on the two image planes [Sun02]. Most methods were fairly complicated, including neural stereo matching [Rui04], Walsh attributes [ACR96], and dynamic programming [GY05]. Roy and Cox improve on the traditional line-by-line stereo matching by finding an accurate depth map in the case of N cameras, but our focus is on finding the color of the closest (x, y) point and not necessarily on finding the best depth [RC98].

2.2.1 Constraints

To make the complex problem of stereo matching easier to digest, we will consider several constraints as suggested by Sun [Sun02]. Those constraints are similarity, uniqueness, continuity, ordering, and epipolar.

Similarity

Perhaps the most obvious constraint is the similarity constraint. It simply says that the matching points must be similar. Barnard breaks the similarity problem into two simple types of matching: area matching and feature matching [BF82]. Area matching is good for images with good texture, but can have problems in situations with large depth disparities [CM92, Fua93]. Feature-based matching provides more precise and reliable results, but requires the extra step of feature detection [Sun02, Fua93].

In area matching, two points are matched by comparing a window around those points. It is

successful in domains where the model's surface varies smoothly and continuously [CM92]. Area matching is based on the continuity constraint: that close pixels usually represent spatially close, contiguous points on an object [BF82, HIG02]. Some kind of correlation measure is used to decide if two groups of pixels correspond to the same point in space. When a match is found, the disparity between the two matching pixels in the two stereo cameras can be used to find the real world location of the point in space.

The results of the correlation depend heavily on how the similarity between two points in two images is calculated. Obviously, two pixels by themselves can not be compared because there would be too many matches in both images. So a window or region of pixels is centered at the target pixel in both images. Then one of several methods is used to compare the pixels in the two regions. Cross correlation or sum of absolute or squared differences are the most common methods [HIG02], but other methods include the non-parametric rank and census measures [ZW94]. In the rank and census measures, correlation depends on the relative ordering of intensities in the window, and not on the intensity values themselves, making them less susceptible to noise and outliers. All of these previous methods use intensity (grayscale level) instead of color because of the complexity of color. However, Chambon and Crouzil present three color correlation methods which always improve the results compared to using grayscale images [CC]. The three methods presented are:

1. Compute the correlation with each color component and merge the results
2. Process a principal component analysis and compute the correlation with the first component
3. Compute the correlation directly with the colors

It makes sense that color would outperform grayscale because it's possible to have two uniform surfaces with similar luminance but with completely different colors. Color correlation would be able to detect the difference, but traditional grayscale correlation would not. Regardless of the correlation method used, different correlation methods can be substituted for each other fairly easily, making it simple to determine how the correlation method directly affects the outcome.

Another difficulty in area-based matching is choosing a window size with which to calculate correlation. Since area matching rests on the continuity assumption, we want to avoid placing the match windows at points where they contain pixels of object points at different depths. A change in depth

means there is a change in disparity, so only some of the pixels will correspond. But it's difficult to prevent this from happening, and thus object borders can be blurred and small details can be removed [HIG02]. The smaller the window size, the less likely it is to cross a depth discontinuity, but a smaller window is more likely to find many matches, increasing the possibility of an incorrect match [KO94]. Kanade and Okutomi change the size of the window after evaluating the local variation of the intensity and the disparity [KO94], but this comes at a cost and can no longer be real-time [BVZ97]. In [HIG02], the correlation is calculated by summing some of the best matches from a sequence of surrounding "supporting" windows to reduce depth disparity error. Other methods use the same window size on different levels of image resolution, which somewhat replicates the method used by Kanade and Okutomi but is more efficient [Fua93].

Besides correlation alone, there are other methods to verify a match. When comparing one window in a left stereo image to several windows in a right stereo image, we can create a correlation graph with the location of the right image's window on the x-axis, and the correlation on the y-axis. Assuming that a high correlation value indicates a good match, we could choose the maximum point on the graph as the best match. However, if there are several local maxima, the absolute maximum point may not be the best match [HIG02]. The left/right consistency check in [Fua93] can help solve this problem. It then would fix the right window at the location of the supposed best match, and compare it to several windows in the left stereo image. If the match was correct, it should find that the best match is the same left point used previously. If not, one of the other local maxima can be used.

In feature matching, features are selected from the image, and the matching strategy is applied to those features. It matches more abstract features rather than texture regions which are susceptible to noise. Points along the edges of intensity discontinuities (dubbed "edgels" for edge elements) are usually the best features because they often are matches made with high confidence [CM92]. Other methods extract straight lines as features [HS89]. We can't create a dense depth map from feature matching alone; instead it is usually combined with area matching [XWZ96, Fua93], an interpolation step [Sun02], or a model-based interpretation step [BF82]. When feature matching is combined with area matching, it is usually superior to area matching by itself [BF82]. However, it requires the extra step of detecting features [Sun02], an extra calculation that may or may not be worthwhile.

Uniqueness

The uniqueness constraint states that each item in each image is assigned at most one disparity value, or in other words, is only matched to one specific real-world object [MP77, CHRM96]. This constraint is fairly obvious, but it does imply that everything in the image corresponds to a physical world object and is not something that doesn't exist in the scene being photographed (such as a lens artifact or noise). Also, Cox shows that sometimes it is too strict to initially limit an algorithm to only one match [Cox94].

Continuity

According to the continuity (or smoothness) constraint, most of the model contains continuous surfaces and thus pixels which are close to each other usually represent adjacent, contiguous points on a single object [BF82, HIG02]. This also means that disparity usually varies smoothly [MP77, HS89]. The continuity constraint is the fundamental principle on which area matching is based. It's based on the assumption that most match windows consist of pixels from a single object that has no depth discontinuity. When the continuity constraint doesn't hold, it's difficult to reliably make a match. Objects at different depths have different disparities in the two stereo images, so several objects at different depths appearing in close proximity in one image will not appear close in the other image. Since stereo matching is founded on the ablility to find the same object in two images, we need to be able to inspect small regions of contiguous objects or else we will not be able to find the region in the other stereo image.

Ordering

The ordering constraint states that ordering is preserved across matches [CHRM96, Bak82, BI99]. Suppose A and B are in image 1 and A' and B' are in image 2. If we match A to A' and B to B' and A is to the left of B in image 1, then A' should be to the left of B' in image 2. This holds true if A and B are on a contiguous object, but can fail if A and B are on different objects. Suppose A and B are between the two stereo cameras, and B is close and A is very far away. In image 1, A will be to the left of B, but in image 2, B' will be to the left of A' (see figure 2.1). The ordering constraint is not often violated in the real world [BI99], but like most of the constraints, it's not a

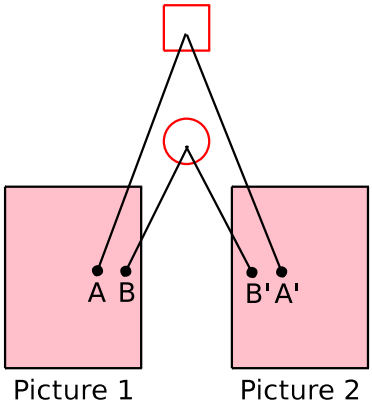


Figure 2.1: The ordering constraint can fail for objects of different depth. A matches A', B matches B', and A is to the left of B in image 1. But B' is to the left of A' in image 2.

steadfast rule.

Epipolar

Perhaps the most useful constraint, the epipolar constraint reduces the search for a matching pixel from two dimensions to one [BF82]. It is based on the geometric constraints of a camera. It helps us find a match because, given a point in one image plane, the corresponding point in another image plane is consrtained to lie along a straight line (see Figure 3.4) [Sch89, HS89]. We can further constrain that line by specifying a minimum and maximum depth for all points in the scene.

2.2.2 Problems

Cochran and Medioni cite four specific problems that occur when doing stereo matching: photometric variation, occlusion, repetitive texture, and lack of texture [CM92]. All of these problems pose serious hurdles for stereo matching algorithms, but fortunately there are many good solutions to these issues.

Photometric Variation

Cochran and Medioni [CM92] and also Ivanov, et al. [IBL00] describe photometric variation as the change of intensities caused by a change in the camera's viewpoint. This happens because the amount of light reflected on the scene and the amount of noise in the image are dependent on the location of the camera. Furthermore, nonlinearities in the camera geometry can cause slight changes when the camera is moved. By performing normalization with respect to the mean and/or the standard deviation, most effects of a change in brightness or contrast between two stereo images can be eliminated [Sun02].

Occlusion

Occlusion occurs when one camera sees a point in space that another camera can not see due to a change in viewpoint [CM92]. Occlusion is a mark of a depth discontinuity and a result of the violation of the continuity constraint [BI99]. If an area is occluded, then it doesn't appear in both stereo images and thus triangulation can't be used to find that area's depth. Bobick and Intille use high-confidence matches to eliminate sensitivity to occlusion and accurately extract large occlusion regions [BI99]. Cox, et al. show that when using more camera views of a scene, occluded regions are reduced or eliminated [CHRM96]. This is obvious because adding another viewpoint can only remove occlusions if it can see areas that one of the other cameras can't see. We plan to take advantage of the fact that we have an array of cameras to help elminate occlusion regions.

Repetitive Texture

Repetitive texture is a problem because it can cause too many good matches. The classic example is a brick wall. It is very difficult for the stereo matching algorithm to tell one brick from another. The problem is compounded when photometric variation alters the two stereo images in such a way that an incorrect match seems like a better match than the correct match [CM92]. Using a large window can help alleviate this problem if the window is large enough to cover an area outside the repetitive texture. In our situation, it may not be a serious problem if we match the wrong part of the same repetitive texture because our objective is not to find the best depth map.

Lack of Texture

If a section of a scene has no distinguishable texture, then an area-based match is not possible [CM92]. For example, a photo of a white wall has no texture, and therefore every point looks the same. It is impossible for even humans to find a specific match, much less a computer. In the real world, it's uncommon to have objects with no texture. In our application, we attempt to minimize this occurrence as much as possible.

Chapter 3

Current Work

The first step was to create a static scene to use as our experimental testbed. In our first experiments, we placed balls on pieces of 6-inch PVC piping on a table. To mimic a sparse camera array, we mounted a camera on a copy stand and took images 10 inches apart to create a 4 by 3 array of photographs of the scene. Using a single camera provided good consistency in the look of the images, reducing or elminating the need for normalization of the photographs. We also wanted the camera axes for each image to be parallel, and we hoped that could be accomplished by using the copy stand setup. Thus, it was easy to do perspective division since the camera location was known. Since the same camera with the same settings was used for all of the images, the perspective division coefficients were equal for each photo so we only had to calibrate one camera.

While taking the images, we paid close attention to the camera settings. We used a 17mm lens (27mm with respect to a 35mm camera), the widest angle possible on our camera, to maximize the depth of field. Using a wide-angle lens also best duplicated the fixed lens in a cheaper camera which might be used for the videoconferencing system. We also set the camera to ISO 400 instead of ISO 100 or 200 to increase the depth of field. We manually focused the camera to about 1 meter distance so that everything was in focus. By using manual focus instead of auto-focus, we guaranteed that all of the images were in the same focus.

I then started by creating basic edge detection and stereo matching algorithms. I used some of those techniques to create an orthogonal projection.

3.1 Edge Detection

Edge detection is simply a threshold applied to a gradient mapping of an image. The gradient for a pixel in the image is the first derivative of image luminance. The luminance, the black-white equivalent of a color, is a linear combination of the color's RGB (red, green, and blue) values. More specifically, a working measure of luminance is the Y component in the YUV color system:

$$Y = 0.3 * R + 0.59 * G + 0.11 * B \text{ [BB82]} \tag{3.1}$$

Vertical and horizontal gradients are estimated for each pixel. If $I(p)$ is the intensity of pixel p, $I'_x(p)$ is the x-gradient of that pixel, and $I'_y(p)$ is the y-gradient of that pixel. To find the x-gradient, we find the average of the difference between pixel p_l (the pixel to the left of p) and p, and the difference between pixel p and p_r (the pixel to the right of p):

$$I'_x(p) = \frac{I(p) - I(p_l)}{2} + \frac{I(p_r) - I(p)}{2}$$

Thus,

$$I'_x(p) = \frac{I(p_r)}{2} - \frac{I(p_l)}{2} \tag{3.2}$$

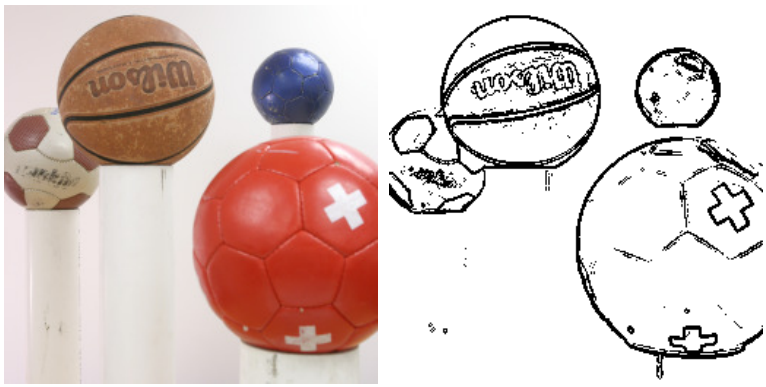
The y-gradient is found similarily. To find the overall gradient for pixel p, we take the square root of the sum of their squares:

$$I'(p) = \sqrt{I'_x(p)^2 + I'_y(p)^2} \tag{3.3}$$

Finally, all pixels with a gradient above the threshold are turned black. All others are turned white. Figures 3.1(a) and 3.1(b) show the results of this edge-detection algorithm with a threshold of 15.

3.2 Interest Operator

Another benefit of the gradient map is to find how "interesting" a point is. Areas with a high gradient are easily identified in other images because the gradient signifies that the area has a unique structure that is unlikely to be found more than once (except for cases of repeating patterns). It is useful to have a way to judge the interest level of a point, because if the point has a low interest level, stereo match candidates might not be trustworthy. Thus, I defined the interest level of a point as the number of edge pixels occurring in a window around that point. The size of that window is



(a) Original image (b) After edge detection

Figure 3.1: The results of a simple gradient-based edge detection algorithm

empirically adjusted based on the scene. To create the edges, I used the Canny edge detector from the OpenCV library [Can83]. I empirically adjusted the parameters for the edge detection algorithm so that the input images had just the right number of edges pixels, and the wall had no edges. Then I created an interest map, where the value for each pixel was the total number of edge pixels in a window surrounding the pixel. I used the same window size used in the area matching algorithm. The resulting interest map looked like a blurry edge detector: the walls were empty (no interest), and the faces were gray and white. Figure 3.2 shows the interest operator applied to one of the input images. Since every pixel that should be matched had a positive interest value, each pixel's interest could be tested before attempting to make a match. If the interest level is zero or below a threshold, the algorithm wouldn't attempt to match it.

3.3 Camera Calibration

The goal of camera calibration in this application is to know where each point in space would project onto the image plane of each camera. This projection is done using the perspective equations. Let (x_e, y_e, z_e) be a point in space relative to the center of the camera (the eye). (x, y) is the point on the image where (x_e, y_e, z_e) projects. An important point to note is that (x, y) is expressed relative to the center of the image with the positive axes pointing up and to the right. So, $(0, 0)$ would be

(a) Before applying interest operator

(b) After applying interest operator

Figure 3.2: The result of applying an interest operator to an image

the center of the image, $(1, 1)$ would be one pixel above and to the right of the center pixel, etc. The perspective equations follow:

$$x = k_x \frac{x_e}{z_e} \tag{3.4}$$

$$y = k_y \frac{y_e}{z_e} \tag{3.5}$$

To use these equations, k_x and k_y must be known. They can be found by measuring the depth of a known point in the scene and finding it manually in two of the images. Let the point in space be (x, y, z), the coordinates of the first camera S be $(S_x, S_y, 0)$, and the coordinates of a second camera T be $(T_x, T_y, 0)$. (x, y, z) is first manually found in S and T. Let (M_x, M_y) and (N_x, N_y) be the locations of (x, y, z) in cameras S and T, respectively. Finally, let Δx and Δy be the x-distance and

y-distance, respectively, between the two cameras. More specifically,

$$\Delta x = S_x - T_x \tag{3.6}$$

$$\Delta y = S_y - T_y \tag{3.7}$$

Then there are two sets of equations:

$$M_x = k_x \frac{x - S_x}{z} \tag{3.8}$$

$$M_y = k_y \frac{y - S_y}{z} \tag{3.9}$$

and

$$N_x = k_x \frac{x - T_x}{z} \tag{3.10}$$

$$N_y = k_y \frac{y - T_y}{z} \tag{3.11}$$

First, we use what we know to deduce:

$$\begin{aligned}
z &= \frac{k_x(x - S_x)}{M_x} \\
&= \frac{k_x(x - T_x - \Delta x)}{M_x} \\
&= \frac{k_x}{M_x}\left(\frac{N_x z}{k_x} - \Delta x\right) \\
&= \frac{k_x}{M_x}\left(\frac{N_x z + \Delta x k_x}{k_x}\right) \\
&= \frac{N_x z + \Delta x k_x}{M_x} \\
&= \frac{N_x z}{M_x} + \frac{\Delta x k_x}{M_x} \\
z - \frac{N_x z}{M_x} &= \frac{\Delta x k_x}{M_x} \\
z\left(1 - \frac{N_x}{M_x}\right) &= \\
z\left(\frac{M_x - N_x}{M_x}\right) &= \\
z &= \left(\frac{M_x}{M_x - N_x}\right)\left(\frac{\Delta x k_x}{M_x}\right)
\end{aligned}$$

Thus,

$$z = k_x \left(\frac{\Delta x}{M_x - N_x}\right) \tag{3.12}$$

The same can be done for y using the same steps. Since z, Δx, M_x, and N_x are known, it is true that:

$$k_x = \frac{z(M_x - N_x)}{\Delta x} \tag{3.13}$$

$$k_y = \frac{z(M_y - N_y)}{\Delta y} \tag{3.14}$$

However, a problem with this method is that it is difficult to measure the depth z manually. If the manual measurement of z is incorrect by even a small ϵ, the k_x and k_y produced will cause incorrect calculations of future z's with an error potentially greater than ϵ. Suppose the correct depth of a point is z, but it is measured to be at depth $z + \epsilon$. Then the incorrect k_x, denoted k_{i_x}, is:

$$\begin{aligned} k_{i_x} &= \frac{(z+\epsilon)(M_x - N_x)}{\Delta x} \\ &= \frac{z(M_x - N_x)}{\Delta x} + \frac{\epsilon(M_x - N_x)}{\Delta x} \\ &= k_x + \frac{\epsilon(M_x - N_x)}{\Delta x} \end{aligned}$$

Thus, when computing z for other corresponding points (P_x, P_y) and (Q_x, Q_y) in the same two images, the incorrect depth z_i will be:

$$\begin{aligned} z_i &= \frac{\Delta x k_{i_x}}{P_x - Q_x} \\ &= \frac{\Delta x}{P_x - Q_x}\left(k_x + \frac{\epsilon(M_x - N_x)}{\Delta x}\right) \\ &= z + \epsilon\left(\frac{M_x - N_x}{P_x - Q_x}\right) \end{aligned}$$

If the depth of the new point is very small, then $P_x - Q_x$ will be large, and the depth error due to finding an incorrect k_x will be small. But if the depth of the new point is very large, then $P_x - Q_x$ will be small and will result in z_i being more incorrect.

If we want an accurate depth measurement, we could measure to a known depth and calculate the distance of the center of the camera from that depth. This is not difficult to do using the aluminum stand I built. It's easy to measure the depth of an object relative to the horizontal bars in the stand. I measured a depth z_1 and z_2 from two points to the horizontal bar, and assumed the center of the camera was a distance of δ from the bar. The true depth of the points relative to the camera would be $z_1 - \delta$ and $z_2 - \delta$, respectively. I also had to find those two points in two cameras, resulting in

two equations:

$$\begin{aligned} k_x &= \frac{(z_1 - \delta)(M_x - N_x)}{\Delta x} \\ k_x &= \frac{(z_2 - \delta)(P_x - Q_x)}{\Delta x} \end{aligned}$$

Thus,

$$\begin{aligned} (z_1 - \delta)(M_x - N_x) &= (z_2 - \delta)(P_x - Q_x) \\ z_1(M_x - N_x) - \delta(M_x - N_x) &= z_2(P_x - Q_x) - \delta(P_x - Q_x) \\ \delta(M_x - N_x) - \delta(P_x - Q_x) &= z_1(M_x - N_x) - z_2(P_x - Q_x) \\ \delta(M_x - N_x + Q_x - P_x) &= \end{aligned}$$

Thus, we can find δ:

$$\delta = \frac{z_1(M_x - N_x) - z_2(P_x - Q_x)}{M_x - N_x + Q_x - P_x} \tag{3.15}$$

Once δ is known, it's easy to find the depth of an object in the scene, and thus the computation of k_x and k_y is much more accurate.

3.4 Stereo Matching

To do stereo matching, I created a basic area-based matching algorithm. Given a pixel in one image (the source pixel), it attempts to find a match for that pixel in another image (the target image). First we must solve the correspondence problem, and then we can use some of the stereo constraints to limit the total area we need to search to find the match.

3.4.1 Stereo Correspondence

The correspondence problem is to find for each pixel in one image its best match in a second image. The correspondence function returns an estimate of match quality. First, windows are centered around the candidate points in both images, and then compared using a match measure. The window size is determined empirically. Two measures we used were sum of absolute differences

(SAD) and cross correlation. To compute the sum of absolute differences, we find a value d by summing the absolute values of the differences between the intensities of the pixels in the two windows and dividing by the number of pixels in the window to find the average difference per pixel. Let A and B be matrices containing the pixel intensities of the first and second windows, respectively. The sum of absolute differences d between two $m \times n$ windows is:

$$d = \frac{\sum_m \sum_n |A_{mn} - B_{mn}|}{mn}$$

Since we're comparing values of d for equal-sized windows, m and n are always the same, so there is no need to divide by mn. Furthermore, we would want to normalize A_{mn} and B_{mn} by the means of the intensities ($\overline{A}$ and $\overline{B}$) of the windows if we used different cameras to take the two images. In images from two different cameras, it's possible for one to be brighter or darker than the other. If we subtract the means of the window from the A_{mn} and B_{mn}, then that should account for any error that occurs if either A or B is uniformly brighter than the other. Therefore:

$$d = \sum_m \sum_n |(A_{mn} - \overline{A}) - (B_{mn} - \overline{B})| \tag{3.16}$$

Since the goal is to find a match in a second image, we would compare the point in the original image with several points in the second image. The point in the second image with the smallest difference value d is the match. Additionally, we could say that if the minimum d is greater than a threshold, then there is no good match.

We can also measure correspondence by finding the cross correlation. Cross correlation measures similarity between two points. I implemented Matlab's corr2 function to find the correlation value c:

$$c = \frac{\sum_m \sum_n (A_{mn} - \overline{A})(B_{mn} - \overline{B})}{\sqrt{\left(\sum_m \sum_n (A_{mn} - \overline{A})^2\right)\left(\sum_m \sum_n (B_{mn} - \overline{B})^2\right)}} \text{ [Inc]} \tag{3.17}$$

The value of the correlation function ranges between -1 and 1, with values close to 1 indicating a good match. With this function, the point with the maximum cross correlation is the best match.

Until now, we have been performing only one search for each pixel, using only one window size. But how large should the window be? If the window is too large, the windows around matching pixels may not have a high correlation because of different point of views between the stereo cameras. If the window contains two objects at different depths, the corresponding window in the other image could look completely different. However, if the window is too small, we may lose the context of

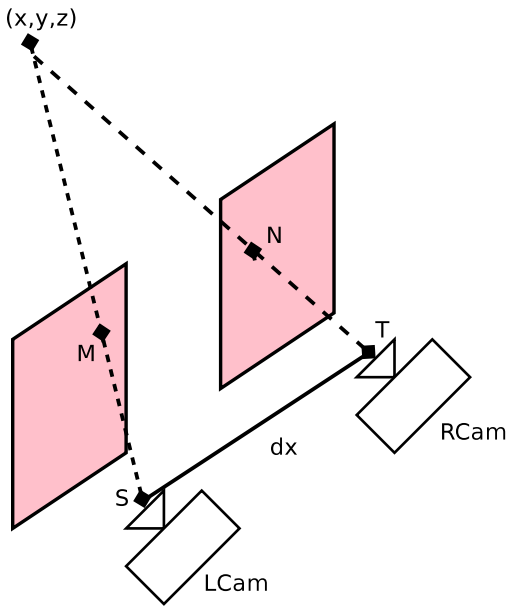


Figure 3.3: Two cameras set up for stereo matching

the point and find too many good matches—especially if the object being matched has a repeating pattern.

A solution is to search with two window sizes. We first find the best match with a large window. This reduces the likelihood we will match the wrong point due to a repeating pattern in the image. Once we find a match, we search around that match with a smaller window so we can drill down to the best matching point.

3.4.2 Finding the Match

For each source pixel, we could search every pixel in the target image. But since we know the location of each camera, we can greatly narrow the area we need to search. Let

$$M = (M_x, M_y)$$

be the coordinates of the source pixel relative to the center of the image. Let

$$S = (S_x, S_y) \text{ and } T = (T_x, T_y)$$

be the location of the source and target cameras. Let Δx and Δy be the x-distance and y-distance, respectively, between the two cameras. More specifically,

$$\Delta x = S_x - T_x$$
$$\Delta y = S_y - T_y$$

The basic arrangement for stereo matching is shown in Figure 3.3. We want to find where M would appear in the target image. If we know the depth of the point in space (x, y, z) projected onto M, then we can find where it projects in the target image. We don't know the depth, but we can bound the depth based on the depth of the scene we are photographing. First, we start with the basic perspective equations:

$$M = (M_x, M_y) = (k_x(x - S_x)/z, k_y(y - S_y)/z)$$
$$N = (N_x, N_y) = (k_x(x - T_x)/z, k_y(y - T_y)/z)$$

Now we can express N in terms of M by manipulating the equation:

$$\begin{aligned} N_x &= \frac{k_x(x - (S_x - \Delta x))}{z} \\ &= \frac{k_x(x - S_x)}{z} + \frac{k_x \Delta x}{z} \\ &= M_x + \frac{k_x \Delta x}{z} \end{aligned}$$

The result is similar for n_y, giving us:

$$N = (M_x + k_x \Delta x/z, M_y + k_y \Delta y/z) \tag{3.18}$$

Since z is now the only unknown, we can find N given a minimum z (N_{min}), and a maximum z (N_{max}). The real location of N will be somewhere along a line drawn between N_{min} and N_{max}.

$$N_{min} = (M_x + k_x \Delta x/z_{min}, M_y + k_y \Delta y/z_{min})$$
$$N_{max} = (M_x + k_x \Delta x/z_{max}, M_y + k_y \Delta y/z_{max})$$

The line between the minimum point and the maximum point is called the epipolar line, and the target pixel should fall along this line [BF82]. We also test several pixels above and below this line to allow for geometric nonlinearities in the camera and/or slight miscalculations. This is illustrated in Figure 3.4.

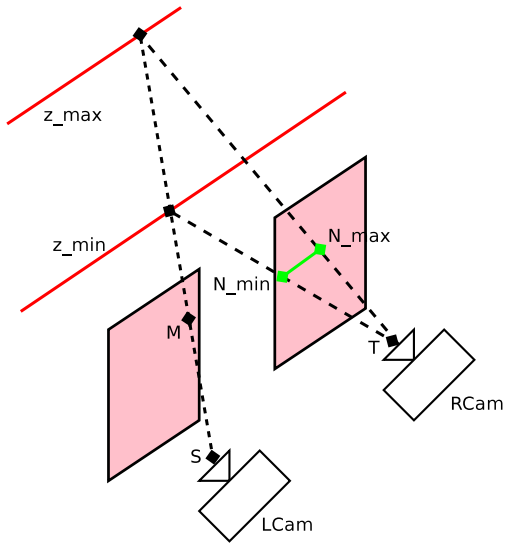


Figure 3.4: Given a point M in the left camera, we can find that point in the right camera by searching along the epipolar line between N_{min} and N_{max}.

Once we find the target point N, we can find the absolute world coordinates (x, y, z) of that point. From Equation 3.12 in §3.3, we know:

$$z = k_x \left(\frac{\Delta x}{M_x - N_x} \right)$$

We also know:

$$\begin{aligned} x - S_x &= M_x z / k_x \\ x &= S_x + M_x z / k_x \end{aligned}$$

We can find y similarily. Now we know (x, y, z), where:

$$z = \begin{cases} k_x \Delta x / (M_x - N_x) & \text{if } m_x \neq n_x, \\ k_y \Delta y / (M_y - N_y) & \text{else.} \end{cases} \tag{3.19}$$

$$x = S_x + z M_x / k_x \tag{3.20}$$

$$y = S_y + z M_y / k_y \tag{3.21}$$

While doing matching in this way worked for very distinguishable points, there were several problems. Specular objects, such as a shiny soccer ball, were difficult to match correctly; often the algorithm

would match the wrong point on the ball. While in doing so it still matched the correct object, matching the wrong point on the correct object causes the depth to be calculated incorrectly. We also had problems with matching borders between balls. In one image, a soccer ball might share a border with a basketball. But in another image, that border might be with the white wall. Finally, it's difficult to match balls with few features or with repeating patches.

3.4.3 Building a Depth Map

By performing stereo matching on every point in an image, we can create a detailed depth map for that image. I developed an algorithm that takes advantage of the many cameras, producing a good depth map for nonzero interest points in all of the images in the camera array. To create the depth map, the algorithm scans through every pixel p in every image in the array. First, that pixel's interest value is determined. If the interest value is zero, no depth is calculated. For pixels of nonzero interest, the algorithm attempts to find that pixel in adjacent cameras. Up to 8 cameras are considered adjacent to any camera: the four cameras above, below, left, and right of the current camera, and the four cameras diagonally above to the left, above to the right, below to the left, and below to the right. There are at least three cameras adjacent to each camera because a camera still has three neighboring cameras if it is in a corner of the camera array.

To find the point p in adjacent cameras, the epipolar line is calculated based on the minimum and maximum depth. The algorithm then uses the sum of absolute differences method to find the point best matching the source point. To accommodate slight errors in camera positioning, the area around the epipolar line is also searched. Once a supposed best match q is found, the left-right consistency check is used [Fua93]. It uses q as the new source point, and searches for a match in the original image. If the match is equal to p within a small error, then it is assumed that q and p correspond to the same point.

Once each neighboring camera is searched, there will be from 0 to 8 matches (0 or 1 match for each neighboring camera) for the source point p. Let the number of matches be n. Using the stereo matching equations, it is then straightforward to find a depth value for each of the n matches. However, some of these matches may be incorrect. Because the matches were confirmed with the left/right consistency check, we assume that most of them are correct. We also assume that the

correct matches will have similar depth values. To decide which matches to keep, a basic clustering algorithm is used. Assuming there are at least two matches, the two points with the closest depth are found. If the difference between those two matches' depths is below a depth threshold, they are kept and a target depth is set to the average of the two closest depths. Then the match with depth closest to the new target depth is found. If the difference between the two depths is less than the depth threshold, then the third match is kept and the new target depth is set to the average of the three depths. The process is repeated until the difference between the target depth and the next closest depth is greater than the depth threshold. The rest of the matches are discarded.

Of the matches that are kept, the depth for all of them is set to their average depth. Then the depth for p and all of its matches is set to the new depth. This process is repeated for each pixel in each camera. If a pixel already has a depth as a result of being previously matched, it is skipped. After all of the images are finished being matched, we will have a depth map for each camera.

This algorithm is extremely slow. It takes several hours to complete using a window of size 21 and images of size 500×750. However, we can increase the speed by skipping every third pixel. We can assume that in a 500×750 image, a 3×3 block of pixels represent the same point. Thus, for each point that we match, we set the depth of the surrounding 3×3 window to the same depth as the matched point. Then, each row only needs to search for the depth of every third pixel, and only every third row needs to be checked. This cuts the run time by 9. The same effect can be produced by reducing the size of the images used to create the depth map, and searching every pixel. This method proved to be more straightforward and resulted in better depth maps (Figure 3.5). First, I created depth maps from 250×375 images. Then, the program was able to save the depth map image and load it for use in a later run. To create orthogonal projections from images with a different size than those used to create the depth maps, the depth maps are first loaded into the program, Gaussian smoothed, and resized to match the new size. Figure 3.6 compares a depth map created from a 500×750 image to a depth map created from a 250×375 image but smoothed and resized for matching with 500×750 images.

Another problem is that even though we average depths for matches found in multiple cameras, different cameras can still record slightly different depths for the same object. This happens mainly because the cameras' alignment is not exactly accurate. We assume that each camera is at a specific position, but if it is not at exactly that position, pointing in that exact direction, the depth will be

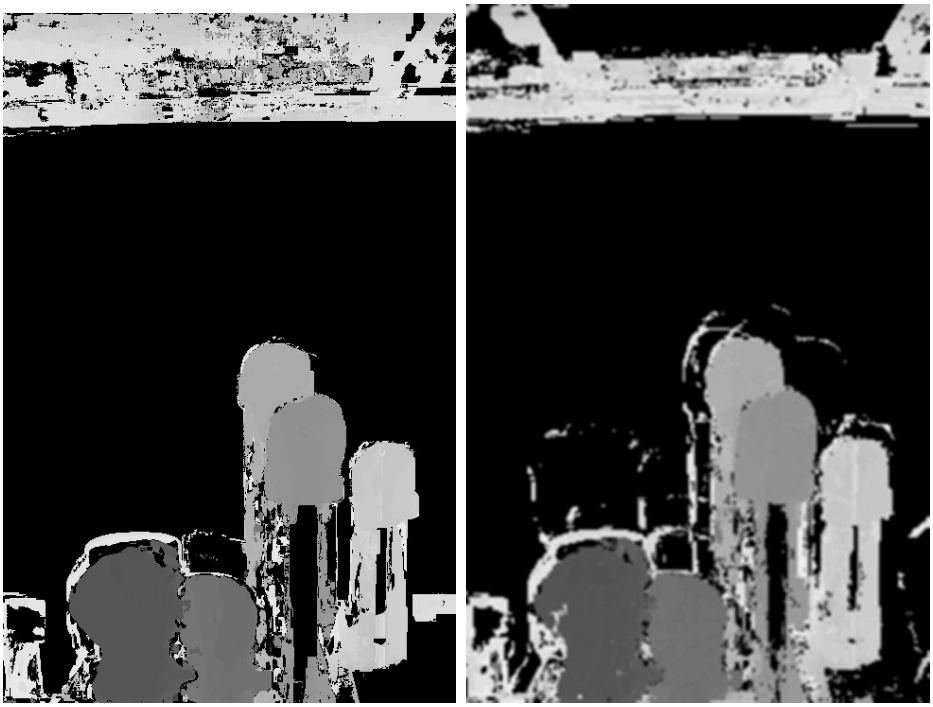
Figure 3.5: Depth map from 250×375 image

calculated slightly incorrectly.

3.5 Orthogonal Projection

I have written two algorithms that create an orthogonal projection from a sparse camera array. The area we will try to project orthographically is the area inside our 4×3 grid, bounded by the outside cameras. Since the centermost pixel of each image is an orthogonal projection of that point in space, it makes sense that the top-left-most pixel in our orthogonal projection is the center pixel of the top-left camera. The same can be said for all of the corner pixels and cameras. For simplicity, we let the center of the bottom-left camera be the origin $(0, 0, 0)$.

Before searching, we have to decide how many pixels per inch (ppi) we would like to sample. The greater the resolution, the slower the run-time, and the larger the orthogonal projection. If our resolution is lower than the resolution of an object in one of the images, our orthogonal projection will become blurry. In our orthogonal projection, the sampling resolution is uniform throughout the image, but that is not the case in perspective images. Since equal-sized objects become smaller as they get farther away in a perspective projection, they are represented by fewer and fewer pixels. Therefore we have to be careful not to make our desired sampling rate lower than the most distant objects in our perspective images or they will be blurry.



(a) Depth map from 500×750 image

(b) Depth map from 250×375 image, smoothed and resized for matching with 500×750 images

Figure 3.6: Depth map for the large image compared to the depth map created from a small image and resized

3.5.1 Algorithm A

In Algorithm A, instead of matching given pixels in a source image with ones in a destination image to create a 3D model, the algorithm starts with an (x, y) point in space and tries to find the object in the scene at (x, y) with the smallest z. The color of that point is projected to a pixel in the orthogonal projection. For example, suppose we are sampling at 10ppi and we find an object at $(25, 10.5, 20)$ and there is nothing at $(25, 10.5)$ closer than 20 inches away. Then that point will be projected onto the pixel (in image coordinates) $(250, h - 105)$ in the orthogonal projection. (h represents the height, in pixels, of the orthogonal projection, since the origin of an image is the top-left corner and x and y are always positive). The algorithm does this search for every pixel in the orthogonal projection.

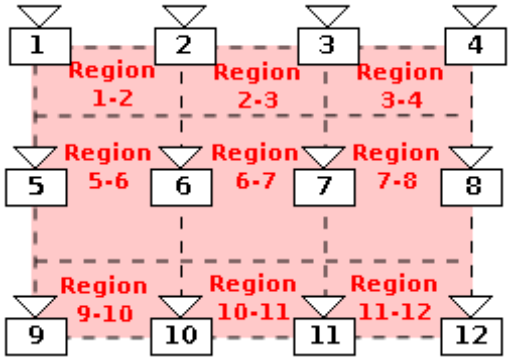


Figure 3.7: Camera pair regions in a 4×3 camera grid

The algorithm is based on the idea that given any point in space and a camera with known parameters, we can find out where an object at that point in space would project onto that camera's image. We can find this by doing simple perspective division. Furthermore, let the function $\Phi_i(x, y, z)$ be coordinates of the projection of (x, y, z) in camera i's image. If there are two cameras photographing the point p in space, we assume an object is at p if $\Phi_1(p)$ and $\Phi_2(p)$ have a high correlation. Since this algorithm includes the stereo matching step, it doesn't depend on the prior creation of depth maps.

Depending on which point we are trying to sample, the algorithm will choose only two cameras from which to match. It is possible that there are points that one of these cameras doesn't see, but we're currently limiting each point to two cameras for simplicity's sake. Because of this, the output orthogonal projection can be divided into camera pair regions. All of the pixels in each camera pair region are matched using the same camera pair. Figure 3.7 shows the regions in a 4×3 camera grid.

Let ortho[i][j] be the pixel at the ith row and jth column in the orthogonal projection. Then the orthogonal projection algorithm is roughly given below:

```
rows:=number of rows of cameras
cols:=number of columns of cameras
dx:=distance between columns
dy:=distance between rows
```

```
ppi:=ppi of orthogonal projection

p:=(0,0)

for i:=rows*dy*ppi-1:0
  for j:=0:cols*dx*ppi
    ortho[i][j]:=color of closest object at p
    p:=(p.x+1/ppi, p.y)
  end for
  p:=(0, p.y+1/ppi)
end for
```

We still need to know how to find the color of the closest object at p. We could just loop through from (x, y, z_{min}) to (x, y, z_{max}) and stop when we find a good match. But that requires computing $\Phi_1(p)$ and $\Phi_2(p)$ each time we increase z. Instead, we varied our approach slightly. We consider the pixels $\Phi_1(x, y, z_{min})$ and $\Phi_1(x, y, z_{max})$ in image 1, and $\Phi_2(x, y, z_{min})$ and $\Phi_2(x, y, z_{max})$ in image 2. Figure 3.8 shows (x, y, z_{min}) projecting to L_{min} and R_{min} in the left and right images, respectively, and (x, y, z_{max}) projecting to L_{max} and R_{max} in the left and right images, respectively. An epipolar line runs between L_{min} and L_{max} in the left image, and between R_{min} and R_{max} in the right image. Then, we run a window along each epiline, calculating the correlation and then moving the window along the line by a distance proportional to the length of that window's epipolar line. This is done once with a large window, and then a second time with a small window in the region of the best match with the large window. The pixel at the center of the small window with the best match is the pixel that is used to sample our target point. Figure 3.9 shows epipolar lines for two images. The white box in the image is the location of the best match using a large window 21 pixels wide.

The algorithm can use two window sizes in an attempt to fix the problem we had with repeating patterns on the soccer balls. The result is much better than the original attempt, although it is not perfect. I also tried changing the correspondence function to use the correlation value instead of the difference value. I was surprised to find out that the correspondence function was not necessarily better, and it might be worse. Figure 3.10 shows an orthogonal projection from a 4×3 camera grid using correlation to find correspondence, and Figure 3.11 shows the same image created using the

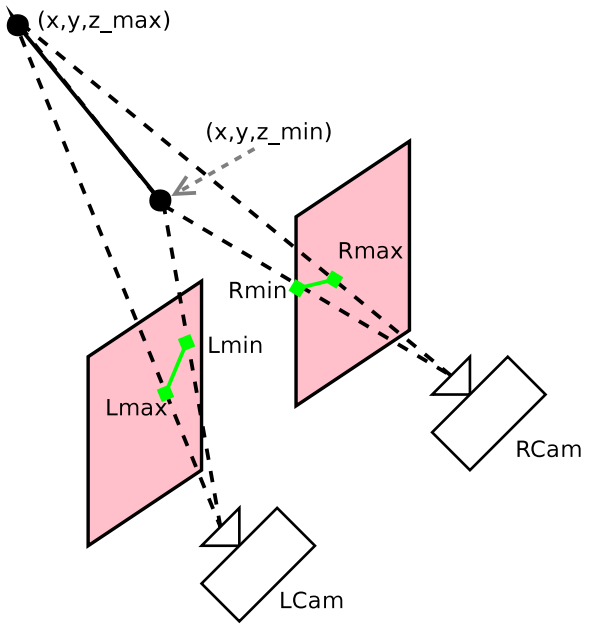


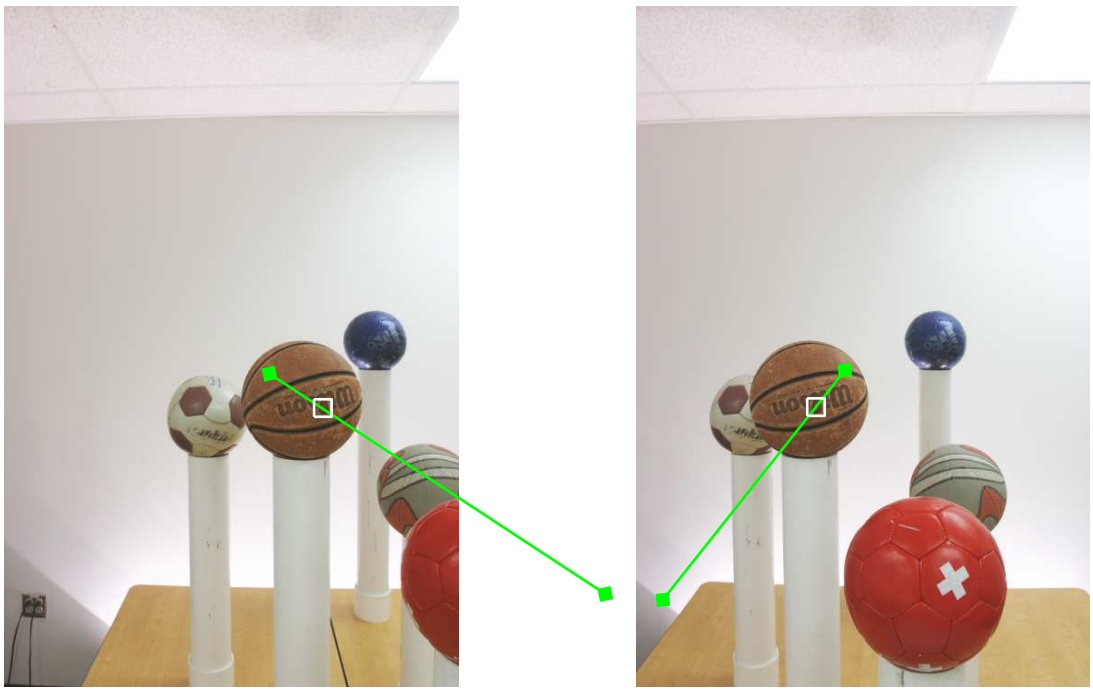
Figure 3.8: Projection of (x, y, z_{min}) and (x, y, z_{max}) onto the image plane and the corresponding epipolar lines

SAD method.

Algorithm Runtime

This algorithm is by no means real-time in its current state. It takes a long time to run, and optimization will be discussed later. There are many variables that affect the algorithm's run-time. These are detailed below:

- Rows (R) - The number of rows of cameras in the grid
- Columns (C) - The number of columns of cameras in the grid
- Δx - The distance between columns
- Δy - The distance between rows
- PPI - The number of pixels per inch in the orthogonal projection



(a) Left stereo image

(b) Right stereo image

Figure 3.9: Two stereo images are shown with their epipolar lines. The best match with a large correlation window is shown as a white square.

- Depth Resolution (d) - The number of iterations for each inch of z-depth—for example, if the depth resolution is 5, the algorithm will test depths of Min Z, Min Z+1/5, Min Z+2/5, etc.
- Max Z (z_{min}) - The largest Z value to test
- Min Z (z_{max}) - The smallest Z value to test
- Large window size (m) - The dimension of the large window
- Small window size (n) - The dimension of the small window

Since the algorithm loops over every pixel in the output orthogonal projection, the run-time is $\#pixels *$ time per pixel. The number of pixels are:

$$(R * \Delta y * PPI)(C * \Delta x * PPI)$$

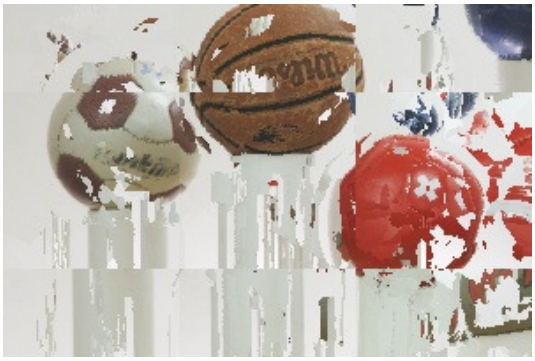


Figure 3.10: Orthogonal projection from a 4×3 camera grid, created with correlation matching

Figure 3.11: Orthogonal projection from a 4×3 camera grid, created with sum-of-difference matching

For each pixel, the algorithm then searches within a large window and a small window. When it searches with a large window, it iterates and finds a correspondence value $(z_{min} - z_{max})d$ times. When it searches with a small window, it actually finds a correspondence value m times, because it searches with a small window in the vicinity of the large window's best match. Finding the correspondence value is an $O(n^2)$ operation, because it has to loop through all of the pixels in the square window. The actual run-time is $2m^2$ and $2n^2$ for the large window and the small window, respectively, because it has to loop through the window to find the mean, and then loop again to either calculate the difference or the correlation. Putting it all together, we have $\#pixels * (\#iterations * 2m^2 + m * 2n^2)$, or:

$$\text{Run-time} = (R * \Delta y * PPI)(C * \Delta x * PPI)((z_{min} - z_{max})d * 2m^2 + 2mn^2)$$

Optimization

Since the run-time of this algorithm is so long, efficiency is crucial. The fewer tasks that are performed in the correspondence function, for example, the better because it's the most called function. In the original implementation of this algorithm, I opened and closed the stereo images each time the correspondence function was called. This really caused a slowdown because of the overhead of loading the files. Now, the images are opened early on and the image file handles are passed as pointers.

The algorithm also used to loop through pixels in the orthogonal projection row by row. In other words, it would find the first pixel in the first row, the second pixel in the first row, etc. The problem is that doing so would require opening and closing different pairs of cameras for each row of pixels. If there are three camera pair regions on each row, and 200 rows, then images are opened and closed 600 times. We could open all of the images into memory, but a very large camera array would use too much memory. Instead, the algorithm finds all of the pixels in each camera pair region before moving on to the next region. Thus, in a 3×4 camera grid with 9 regions, camera pair images are only opened and closed 9 times.

Because of the specific application and the relatively small size of the images, I eventually decided that it would be best to load all of the images into memory at once. That allows nearly instant access to information about any of the 12 cameras, and allows more complicated multiple-camera matching.

Dropping Calibration Requirement

After developing this algorithm for a while, I realized that it's possible to achieve the same results without calibrating the cameras and finding k_x and k_y from the perspective equations. Recall the perspective equations 3.4 and 3.5 in §3.3:

$$\begin{aligned} x &= k_x \frac{x_e}{z_e} \\ y &= k_y \frac{y_e}{z_e} \end{aligned}$$

We want to find the epipolar line from $\Phi(x_e, y_e, z_{min})$ to $\Phi(x_e, y_e, z_{max})$. As $z \to \infty$, x and y approach 0. Therefore, the epipolar line is a segment of a ray pointing from the center of the image out in one direction. Furthermore, in photographs, k_x should equal k_y or else there would be distortion. For example, if $k_x \neq k_y$, then a 5-inch horizontal line would be represented by a different size in the image than a 5-inch vertical line (assuming equal depth and foreshortening). So, we can refer to k_x and k_y as simply k. Then, $\Phi(x_e, y_e, z_e) = (kx_e/z_e, ky_e/z_e)$. Since k is constant, and z_e is applied equally to both x_e and y_e, the slope of the epipolar line is determined by x_e and y_e alone. This makes sense: a point located at $(5, 5, z)$ will be projected somewhere in the image along the line $x = y$ with $x, y > 0$, regardless of the camera parameters such as focal length, which determine k. The position along that line depends on z and k. Thus, we can find the epipolar line with (x_e, y_e) alone.

While we don't know where along the line (x_e, y_e, z_{min}) and (x_e, y_e, z_max) fall without knowing k, all that matters is that the beginning and ending of the epipolar lines in the two images are at the same z. We always set the end of the line at $z = \infty$, which is always $(0, 0, 0)$. So, we must make the beginning of the lines correspond to the same z-value. To do this, start by creating the epipolar line for the camera farther from the point in the scene, because it will have a longer epipolar line. The closer a point is to the center of the camera (assuming a fixed z), the smaller the epipolar line will be, because (x, y, z_{min}) will be closer to $(0, 0)$. If the epipolar line for the farther camera is created first, the epipolar line for the second camera will be completely inside the image plane.

Since (x_e, y_e) alone determines the slope of the epipolar line, it's easy to calculate it. We will let the epipolar line run from the center of the image to the border. Then, it will either intersect with either the top/bottom border or the side border. Let w and h be the width and height of the images, respectively. If $|x_e/y_e| > w/h$, then the epipolar line will intersect with the side of the

image. If $|x_e/y_e| < w/h$, then the epipolar line will intersect with the top or bottom of the image. If $|x_e/y_e| = w/h$, then the line will intersect with the corner of the image. The beginning of the epipolar line is chosen as follows:

```
if |x_e/y_e| > w/h
  if x_e > 0
    x_min = w
  else
    x_min = -w
  end if

  y_min = x_min(y_e/x_e)

else
  if y_e > 0
    y_min = h
  else
    y_min = -h
  end if

  x_min = y_min(x_e/y_e)
end if
```

The epipolar line runs from (x_{min}, y_{min}) to $(0, 0)$. To make the epipolar line in the other image start at the same z-depth as the epipolar line in the first image, recall that a camera translation along an axis only moves pixels along that same axis. So, if the two images only differ by a horizontal translation, then the minimum point on the epipolar line in the second image will have a y-value of y_{min}. Then the x-value can be calculated based on the ratio of x_e to y_e for the new camera. Likewise, if the images differ by a vertical translation, the minimum point on epipolar line in the second image will have the same x-value as the minimum point on the epipolar line in the first image.

Once the two epipolar lines are set, they are used just in the same way in the algorithm. The only

difference is that k no longer needs to be known. As long as the cameras are properly aligned, no calibration needs to take place.

3.5.2 Algorithm B

Because of problems described in §3.5.3, I decided to investigate an alternative algorithm to create the orthogonal projection. It is simpler, but requires creating a good depth map of all the images in the array—a time-consuming process. The idea is that with an accurate depth map for all of the images, we know the locations in the scene represented by any pixel assigned a depth. This would create a 3D model of the scene, and then an orthogonal projection could be created.

Algorithm B loops through every pixel in each image. If the depth is known, it calculates the coordinates for the point in the scene which the pixel represents. This is done using a form of the perspective equations. Recall that M is the location of a pixel in the images S representing the point (x, y, z) in space (from Equations 3.4 and 3.5 from §3.3):

$$M = (M_x, M_y) = (k_x(x - S_x)/z, k_y(y - S_y)/z)$$

And recall Equations 3.20 and 3.21 in §3.4.2, where given M and z, we can find (x, y, z):

$$\begin{aligned} x &= M_x z/k_x + S_x \\ y &= M_y z/k_x + S_y \end{aligned}$$

Once we know the coordinates (x, y, z) of a point, we can calculate its coordinates in the orthogonal projection. Let $O = (O_x, O_y)$ be the point on the orthogonal projection corresponding to (x, y, z), and let h be the height (in pixels) of the orthogonal projection. Let r be the sampling resolution (pixels per inch). Assuming that the leftmost column of the orthogonal projection contains points at $x = 0$ in the scene, and the bottom row contains points at $y = 0$ in the scene, we can find O in terms of x, y, h, and r:

$$(O_x, O_y) = (xr, h - yr) \tag{3.22}$$

O_y is expressed as $h - yr$ because O is in image coordinates, where pixel $(0, 0)$ is at the top-left of the image and the positive y axis points downward. We multiply the coordinates by r because x inches * r pixels/inch = xr pixels.

Once we calculate O, we check the depth of the current point projected to the same pixel O. We color O the same as M in S if and only if O hasn't yet been colored or it is colored but its depth is greater than z. This process is repeated for every pixel in every image.

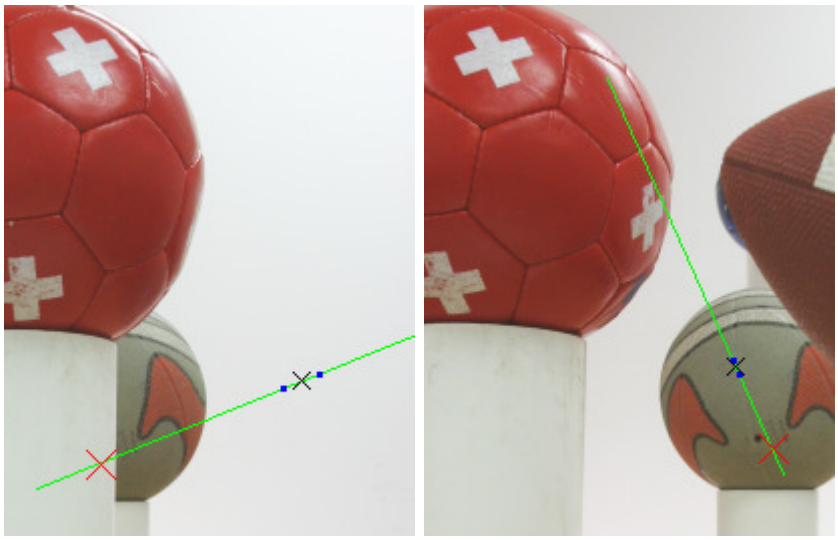
3.5.3 Problems

After implementing the algorithms, several problems arose that made it difficult to create a high-quality orthogonal projection. First, occlusion, a problem common in stereo matching, caused areas in the orthogonal projection to be missing. Another problem, repetition in the images, can cause errors in the orthogonal projection. Finally, slight errors in camera alignment can prevent the orthogonal projection from looking good at all.

Occlusion

The problem of occlusion was a big issue in Algorithm A. We made the assumption that every point in the ideal orthogonal projection is visible from at least two cameras in the array, but that doesn't mean that every point is visible by the same two cameras. To save computation time, the algorithm can be made to use only two cameras for each pixel in the orthogonal projection. But when the point in the scene that should project onto the orthogonal projection isn't visible from both of the chosen cameras, the pixel is colored incorrectly.

An example of this is shown in Figures 3.12 and 3.13. Figures 3.12(a) and 3.12(b) show the left and right images used for stereo matching in the current region. The epipolar lines are green, and the black X shows the match that the algorithm found. The actual point that should be matched is marked with a red X. But in the left image, that point is occluded by the white pipe. The result is that the best match only has a correlation of about 0.3 (using the correlation coefficient) on a scale from -1 to 1, with 1 meaning the two points are equal. Figure 3.13 shows a section of the resulting orthogonal projection. The white blob in the middle of the ball is the area that could not be matched well because of occlusion. If the right camera was used in conjunction with the camera above it to do the matching for this area, it would have found the correct match.



(a) Left stereo image (b) Right stereo image

Figure 3.12: A stereo pair used to match a region: the green line is the epipolar line, the blue squares bound the region where the algorithm will search using a small window, and the black X is the best match. The red X shows where it should have found a match if the ball wasn't occluded in the left image.

Repetition

Another problem specific to Algorithm A is that it is prone to errors if there is any replication in the scene. Replication here is defined as any surface in the scene which is duplicated somewhere else in the scene. It specifically becomes a problem in this algorithm when the epipolar lines from the two stereo images cross a replicated area. Figure 3.14 illustrates this problem. Two similar things on either side of an object appear to be in front of that object. When using more complicated objects

Figure 3.13: The orthogonal projection of a ball that was partially occluded by one of the cameras

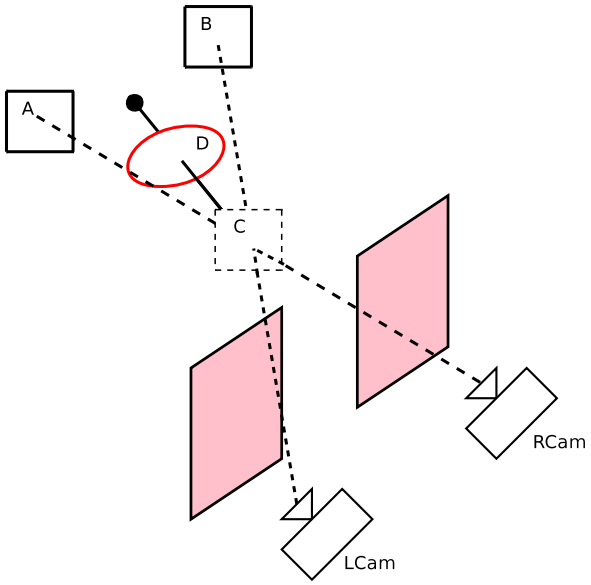
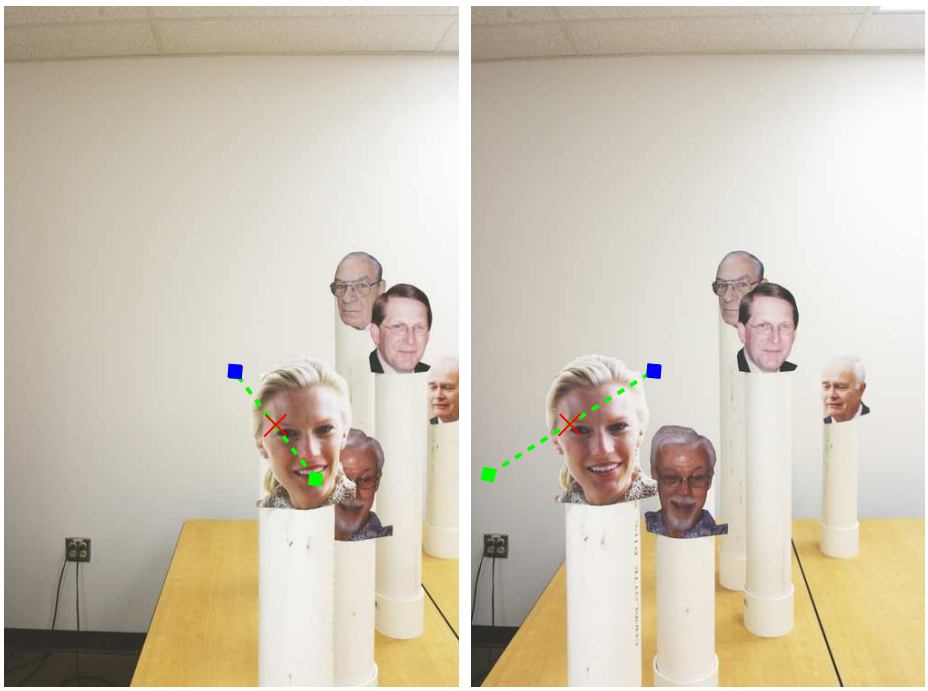


Figure 3.14: An example of replication: the intersection of object D with the bold line should be the point which is projected onto the orthogonal projection. But because of the locations of symmetrical objects A and B, the algorithm thinks there is an object C in front of D.

such as faces, there is less of a problem with replication. While faces are symmetric, the objects on the face which are replicated are small enough that they usually are not a problem, and we just need to use a large enough window size to rule out most cases of symmetry. The background poses a much bigger problem. Since the background is uniform, it is easy for the background to be matched by mistake. Figure 3.15 shows a real example of how replication caused half of the face to be missing in Figure 3.16.

Camera Alignment

Perhaps the biggest obstacle in the way of creating the orthogonal projection is error in camera alignment. It is extremely difficult to keep all of the cameras precisely in known positions and pointed exactly perpendicular to the orthogonal projection plane. In most stereo matching applications, slight errors in alignment are forgiveable because the algorithm searches above and below the epipolar line. However, Algorithm A doesn't search for a target point given a fixed source point. It searches



(a) Left stereo image (b) Right stereo image

Figure 3.15: A stereo pair exhibiting error due to replication: the green lines are the epipolar lines, and the blue square is the point where the algorithm made an incorrect match. The red X shows where the match should have been made.

two lines in two images in hopes that the lines intersect exactly one point in the scene. If the camera alignment is slightly incorrect, they might intersect at the wrong point, or they might not intersect at all. In our experiment, it turned out that cameras that were supposed to be above each other were slightly misaligned. Most points were shifted to the side by a significant number of pixels (often 12 or more) when translating the camera up and down on a supposed vertical line. However, cameras to the left and right of each other had had good matches because when translating the camera from left to right, points did not shift up and down on the image very much.

In Algorithm B, I attempted to avoid the alignment problem by first creating a depth map. Stereo correlation is less sensitive to alignment problems because matching points in two images can be found without knowing camera geometry. Camera geometry is still used to find epipolar lines, but

Figure 3.16: An example of replication errors: half of the woman's face is missing because the white wall was matched instead.

the algorithm searches around the epipolar line to compensate in slight alignment errors. However, the problem lies in computing the depth. Camera geometry is needed to compute the depth, and again, slight errors in alignment can cause depth inconsistencies for the same object in different depth maps. It would be straightforward if the goal was to find a 3D model from two stereo cameras. Even if the two cameras weren't perfectly aligned, the depth errors would be consistent throughout the image, and there would be no other camera pairs to "disagree" with the calculated depth. But this algorithm uses images from almost all of the cameras, so it becomes crucial for the depth maps to be consistent across all cameras.

3.6 Improvements

3.6.1 Algorithm A

Problems of occlusion and repetition in Algorithm A can be solved using more than two cameras to find each point in the orthogonal projection. While there are some points in the orthogonal projection which would be occluded in some cameras, I believe that every point is visible in at least two of the cameras in the array. So, I improved the code to search in several camera pairs for each point in the orthogonal projection.

Four cameras surround each point. A natural sequence of camera pairs to use for stereo matching for that point would be top-left and top-right, bottom-left and bottom-right, top-left and bottom-

Figure 3.17: The result of Algorithm A when only stereo camera pairs with a vertical translation were used

left, and top-right and bottom-right. Of course, there are $\binom{4}{2} = 6$ possible combinations if the two diagonal pairs are added, but four pairs should be enough.

For each camera, the best match is found. With each match comes a depth, so it is possible to have the four camera pairs "vote" on which match to use. If at least two of the matches agree that the object is at a certain depth, then that is the depth used. Once the depth for a point is known, pixel can be projected from any of the images that agreed on that depth.

3.6.2 Alignment

While there were decent results using Algorithm A, problems of repetition and occlusion prevented the creation of an acceptable orthogonal projection. I should be able to avoid these problems by matching each point with several cameras and using the result from the camera with the best match, but errors in camera alignment prevented any accurate vertical stereo matches. Figure 3.17 shows the result of Algorithm A when only vertical camera pairs were used. The poor quality of this result means that vertical matches are useless to augment the horizontal matches.

The need for properly aligned images of the scene became apparent. One of the biggest problems with our setup was that the vertical shaft of the copy stand did not correctly translate the camera vertically. Objects in the images shifted to the left or right by as many as 12 pixels when only raising

or lowering the camera on the copy stand. A vertical translation shouldn't cause any horizontal pixel shifts, just as a horizontal translation shouldn't cause any vertical pixel shifts. This problem was unique to vertical translation: pixels in the images didn't shift vertically when the camera was translated horizontally. This fact explains why using Algorithm A created reasonable results only when performing left-right stereo matches. The same error has also been demonstrated with other cameras, making it unlikely that the error was due to that particular camera.

To attempt a properly aligned camera array, I bought a piece of foam-core board the size of the orthogonal projection (20" $\times$ 30"). I created a square grid on it, with 12 points at grid intersection points. I then mounted it on the background wall using a level to ensure that the grid lines were exactly horizontal and vertical. The goal was to have the camera at the exact (x, y) location as the points in the wall grid, and pointing straight at the dots so that the dots were in the exact center of the image. I attached the camera to the copy stand, and used a laser level to align it. I used a vertical line on the laser level to project a line along the first grid point, and then I could translate the camera horizontally until the vertical laser ran directly through the center of the camera. Of course, I had to use a large carpenter's square to make sure that the vertical laser was perpendicular to the wall. I used the horizontal laser to make sure the camera was at the correct height from the floor. I also measured the height with a ruler to double-check that the height of the first grid point matched the height of the camera.

With the camera at the correct (x, y) location, if the grid point was not at the center of the image, then the error was due to rotation about the vertical or horizontal axis. So, I took an image of the scene, and with the camera still attached to the copy stand, I plugged in the cable and downloaded the image. By inspecting the image I knew whether I should rotatate the table about the vertical axis (moving the camera center horizontally), or rotate the camera about the horizontal axis (moving the camera center vertically). By repeating this process several times, I was able to properly align the camera so that it was pointing perpendicular to the wall, and at the same (x, y) location as the target grid point.

I assumed that once the camera was initially aligned, that it would be trivial to take the remaining images. However, when I cranked the copy stand up 10 inches to the next spot and downloaded a new image, I noticed that the center of the camera was several pixels to the right of the next grid intersection point. If pixels shift horizontally with a vertical translation of the camera, it means

that either the camera was either translated horizontally, rotated about the vertical axis, or both. By using the vertical laser, I could tell that the camera had not been translated horizontally when I cranked the camera vertically. But just to make sure, I shifted the camera horizontally until the center of the image was correct. It took a horizontal shift of over an inch to compensate for the center of the image moving several pixels to the right. I would have been able to detect a horizontal shift of this magnitude using the vertical laser. Therefore, the error must have been due to a rotation about the vertical axis.

I could compensate for error due to a vertical axis rotation by rotating the table on which the copy stand sits, but it would be impossible to measure this movement, and it would be very difficult to continually rotate the table to different but precise locations based on the copy stand crank height to compensate for the rotation error.

I realized a different tool would be needed to take correctly aligned images, so I built the aluminum stand described in the introduction in §1.2. This seemed to fix the alignment problem, but the resulting orthogonal projection wasn't great. These results are described in Chapter 4

Chapter 4

Results

Using the aluminum stand did not fix vertical stereo matching problems. While the images were aligned, vertical matching still did not work properly. Furthermore, with almost every set of images we've taken, it is possible to get good results, but usually with different parameters. Some of the parameters that I've been changing include:

- Which camera pair to use to match a point
- Which camera of the pair use when grabbing the pixel
- Window size
- Using one or two window sizes
- Interest window size
- Interest threshold
- Correlation threshold

4.1 Parameters Affecting Orthogonal Projection

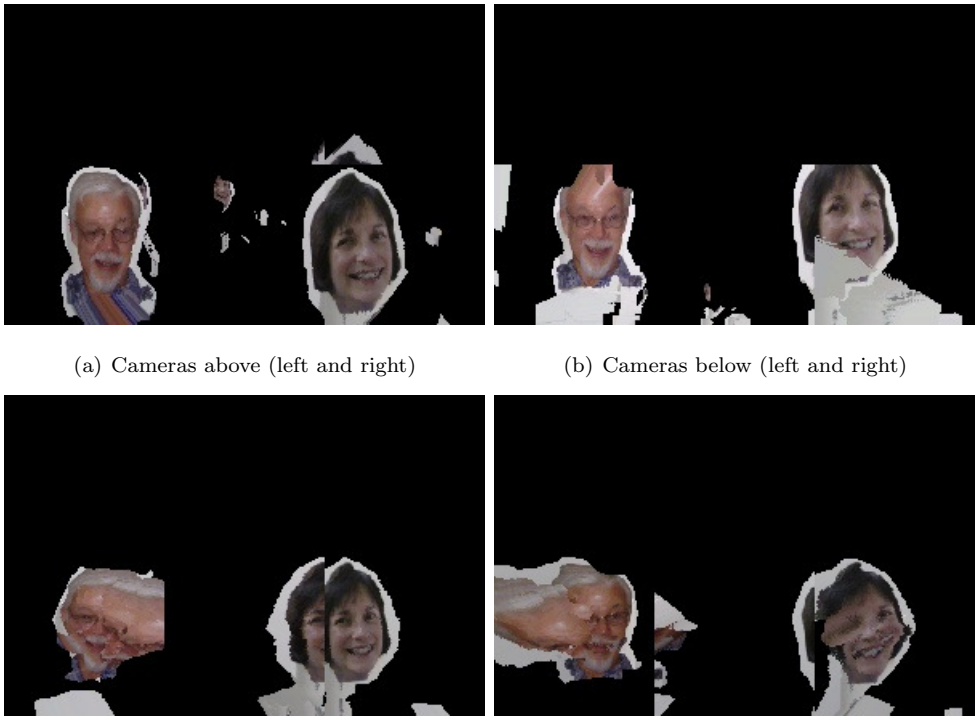
4.1.1 Which camera pairs to use

For each scene photographed, different parameters resulted in better images. The most difficult parameter to set is which camera pair to use. I intended to make use of four camera pairs to find a good match for each point. If one camera pair couldn't find the match, then surely one of the other three camera pairs would have a good match. I tried creating four orthogonal projections for each run, one for each camera pair used. For example, the first orthogonal projection uses the cameras above (left and right) each point to do the match, the second projection uses the cameras below (left and right) each point, and so on. The problem was, however, that usually only one or two of the orthogonal projections looked good. I couldn't reliably switch between different camera pairs to supply a good match. If each of the projections looked equally distorted, then there might be a chance of combining them and taking the best match for each pixel. Usually one camera pair provided really good matches, and two or three are very bad. Figure 4.1 shows an example of four orthogonal projections of two heads. Furthermore, usually the vertical matches were bad, as shown in the figure. I was not able to discover why the vertical matches were so poor, even after drawing the epipolar lines from the stereo pair. The epipolar lines don't seem to be in the correct place for them to intersect at the right point, but after doing manual calculations, the lines are at the mathematically correct coordinates. Figure 4.2 shows this comparison.

Another problem with using alternating between different camera pairs is that each individual camera pair produces an orthogonal projection in a slightly different place. If one camera pair thinks that the point projecting to $(3, 4)$ is the eye, and another camera pair thinks that point should contain a nose, then the algorithm can't alternate between using using different camera pairs for the same orthogonal projection, even if both have good matches, because they could all have good matches for different points.

4.1.2 Which camera of the pair to use when grabbing the pixel

Another choice to make is which camera in a pair to use when grabbing a pixel. When a camera pair returns a match, it has a left camera match and a right camera match. Then the pixel can



(a) Cameras above (left and right)

(b) Cameras below (left and right)

(c) Cameras to the left (top and bottom)

(d) Cameras to the right (top and bottom)

Figure 4.1: Four orthogonal projections from the same camera array using different camera pairs to do stereo matching

be grabbed from either camera. Ideally, if the camera is a true match, it shouldn't matter which camera to use. But the match isn't always correct. However, it's an arbitrary decision to choose one camera over another. I thought the results might be better if I grabbed the pixel from the camera closer to the pixel, but the results didn't generally improve or get worse.

4.1.3 Window size

The size of the match window has a big effect on the orthogonal projection quality. A large window size usually produces more accurate matches, except in the case of a depth discontinuity. Figure 4.3 shows an orthogonal projection created using window sizes of 11, 21, 31, and 41. In this case, it appears that a window size of 31 is the best. As the window size decreases, there is more noise

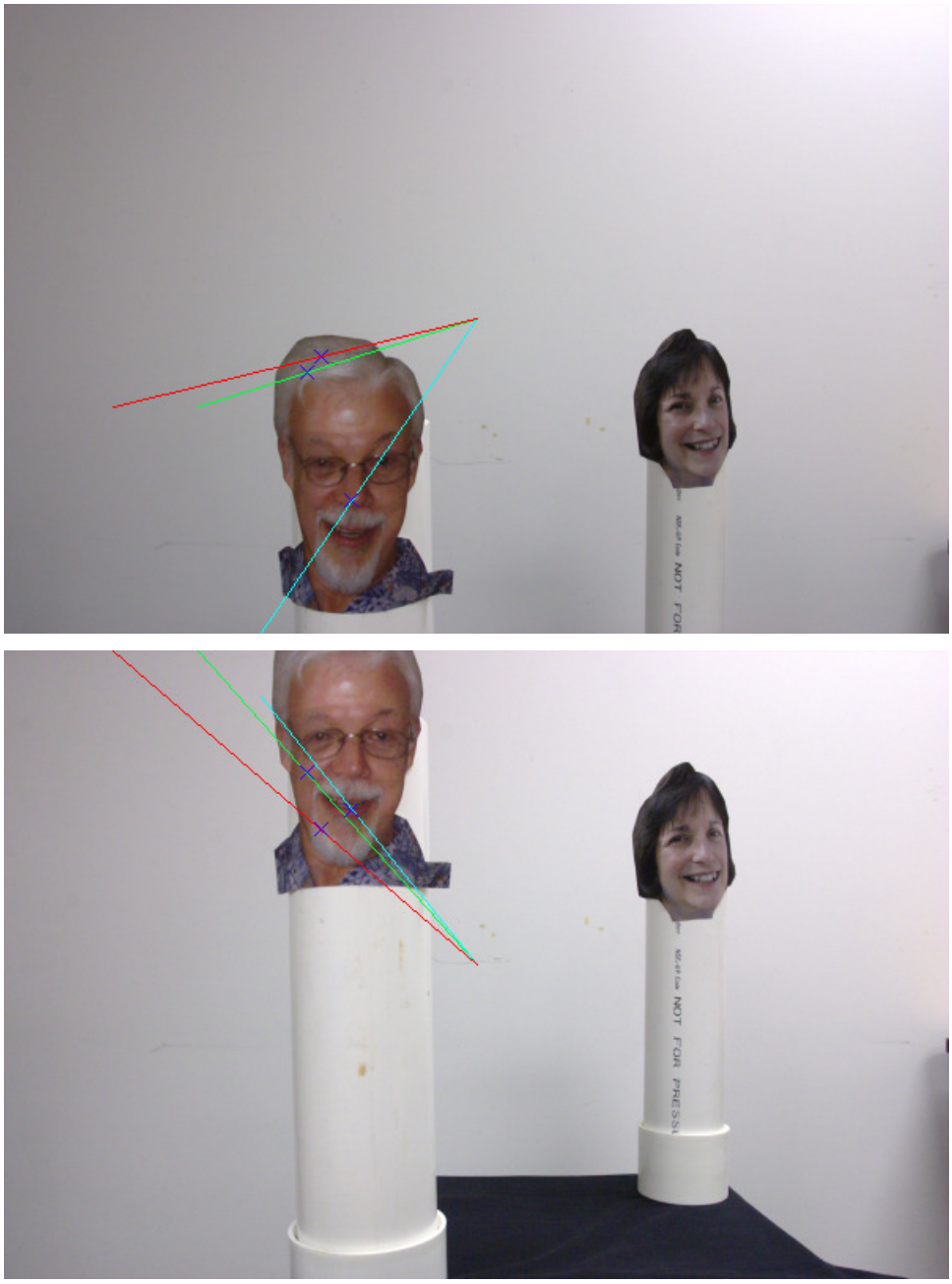
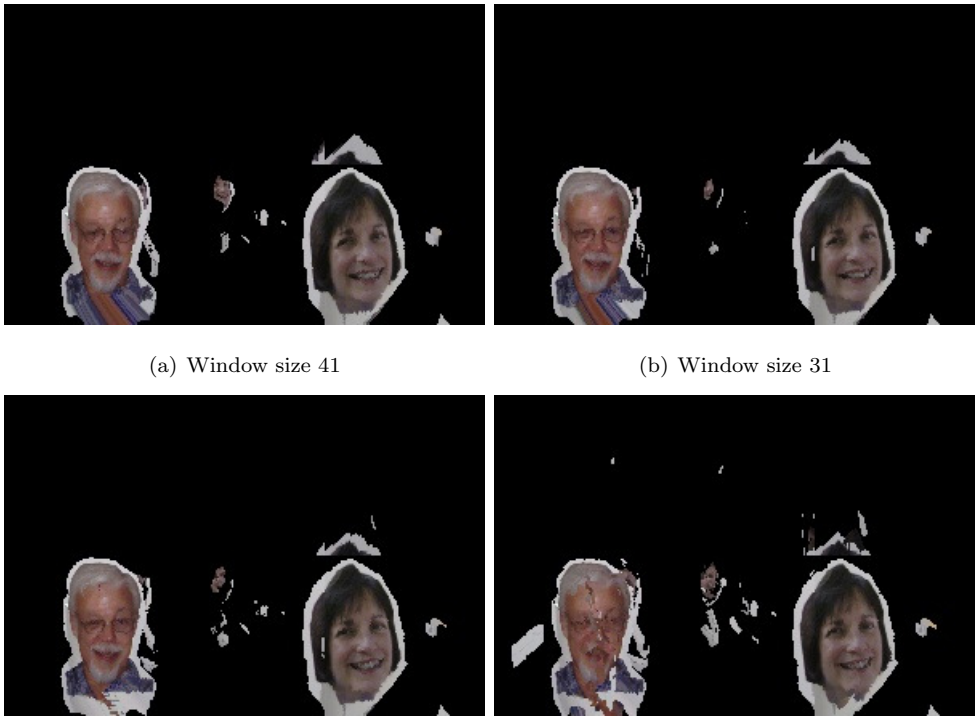


Figure 4.2: The epipolars line and the best match for two cameras attempting to perform three different vertical stereo matches



(a) Window size 41 (b) Window size 31

(c) Window size 21 (d) Window size 11

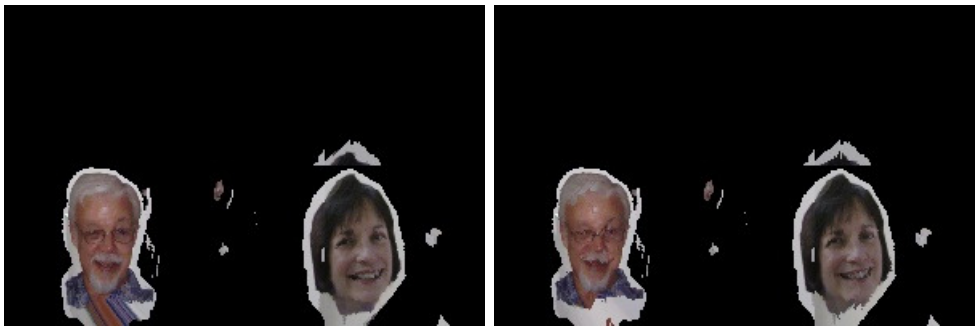
Figure 4.3: Four orthogonal projections created with different window sizes

in the image. A window size here of 21 still has acceptable results, and finally a window size of 11 produces bad matches because it is too small. Large window sizes also increase computation time significantly, since correlation (the most called function) is done on the window. Large windows generally produced better results, but when there are edges with depth discontinuities, it can prevent good matches near those edges (Figure 4.4).

4.1.4 Using one or two window sizes

Two window sizes can be used to increase accuracy around the borders of depth discontinuities, with a tradeoff of increased noise. However, if the scene doesn't have any depth discontinuities, then there is little gain in using two window sizes (Figure 4.5).

Figure 4.4: When a large window is used, depth discontinuities contain errors. Note especially the edge between the two men's heads.

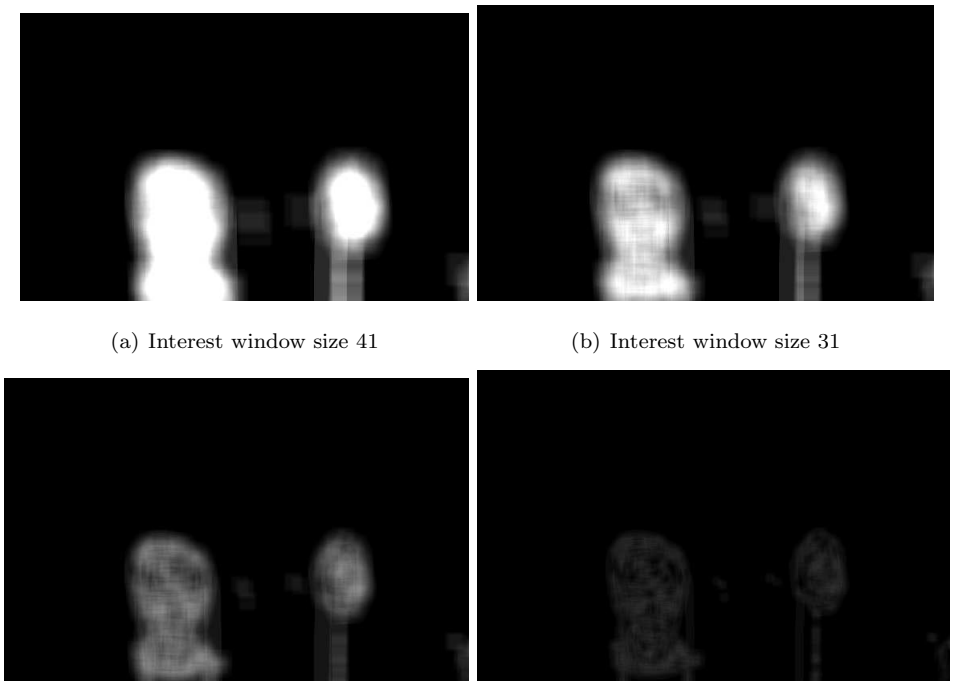


(a) Only one window, size 31 (b) Two windows, sizes 31 and 11

Figure 4.5: Two orthogonal projections, one created with one window and another with two

4.1.5 Interest window size

For each point, the number of edge pixels in a window around that point are summed to create the interest value of a point. The size of that window is the interest window size. Figure 4.6 shows the resulting interest image based on different interest window sizes. I chose an interest window of size 21. If the interest window is too large, it will cause too much of the surrounding background to have a high interest value, leading to poor matches and poor orthogonal projections. The interest value should be the lowest possible value while also giving each point that should be matched an interest value above the interest threshold.



(a) Interest window size 41

(b) Interest window size 31

(c) Interest window size 21

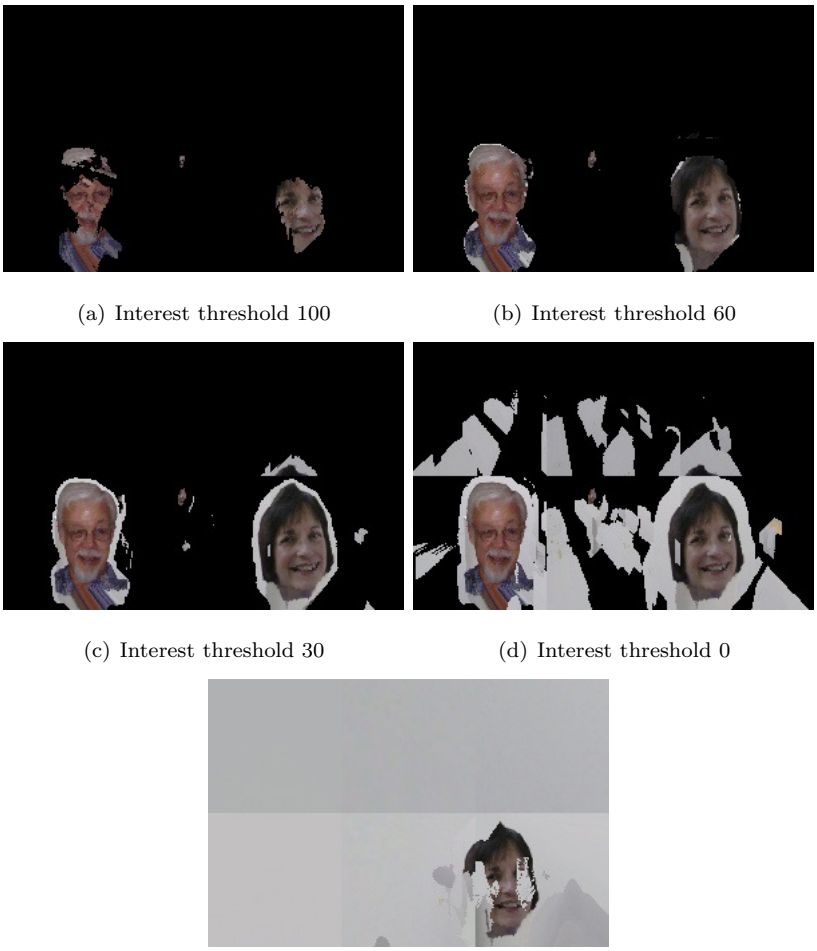
(d) Interest window size 11

Figure 4.6: The interest image for an image, given different interest window sizes

4.1.6 Interest threshold

Points in the images which have an interest value below the interest threshold are ignored and not matched. The interest threshold needs to be empirically adjusted to a good value for the input images. Figure 4.7 shows the result of adjusting the interest threshold for an orthogonal projection. The interest window size used is 21. The results are intuitive: the higher the interest threshold, the fewer low-interest points are matched. However, if the interest threshold is too high, then part of the image is not matched that should be matched. If the interest threshold is too low, then the wall starts appearing in the scene. We don't want to match the wall because it is difficult to get good matches on the wall, producing errors in the orthogonal projection. When no interest threshold is used, the interest feature is effectively turned off and a match is attempted for every point. This results in the wall being matched almost everywhere and the scene nearly being erased. The interest threshold should be as high as possible without cutting off important parts of the images, because

the algorithm runs much faster when it can quickly decide to ignore a point due to a low interest value.



(a) Interest threshold 100

(b) Interest threshold 60

(c) Interest threshold 30

(d) Interest threshold 0

(e) No interest threshold

Figure 4.7: The results of changes in interest threshold

4.1.7 Correlation threshold

The correlation threshold specifies that the correlation value should be either at most or at least a certain value, or else it is not a good match. If SAD is used, then a high correlation value means that the difference between the two points is large. If cross correlation is used, a high correlation

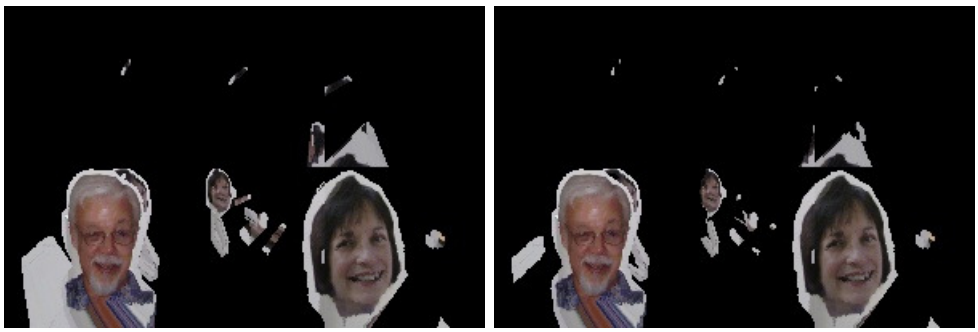
value indicates a good match. For example, using SAD correlation, if the best match for a particular point in the orthogonal projection has a difference value of 50, and the threshold is 60, the match is accepted. If the best match had a difference value greater than the threshold, then no match would be accepted for that point on the projection.

Instead of pre-defining a correlation threshold, I define a correlation threshold per pixel. If the window size changes, the correlation threshold needs to change in proportion to the number of pixels in the window. Obviously, a correlation value of 50 in a window of size 11 has a different meaning than a correlation value of 50 in a window of size 51. By setting the correlation threshold per pixel, the correlation value returned for a point is compared to the correlation threshold per pixel multiplied by the number of pixels in the window. Figure 4.8 shows an orthogonal projection created using different correlation thresholds.

4.2 Optimal Configuration

The optimal configuration changes depending on the input images. For a final run, I took images of two heads separated to eliminate occlusion problems. The input images were 600x399, using the same previously discussed camera settings. I used the orthogonal projection created by the above-left and above-right cameras, sampling from the left camera. I found that matching with one window of size 31, an interest window of size 21, an interest threshold of 50, and a correlation threshold per pixel of 31 produced the best results.

The sampling resolution, or pixels per inch of the orthogonal projection, doesn't have a large affect on the quality of the orthogonal projection. We want a large resolution, but the size is limited by the size of the input images. Figure 4.9 shows the input images, and Figure 4.10 shows the orthogonal projection with the aforementioned configuration with three different sampling resolutions.



(a) No correlation threshold

(b) Correlation threshold per pixel 60

(c) Correlation threshold per pixel 31

(d) Correlation threshold per pixel 20

(e) Correlation threshold per pixel 10

Figure 4.8: The results of changes in correlation threshold per pixel

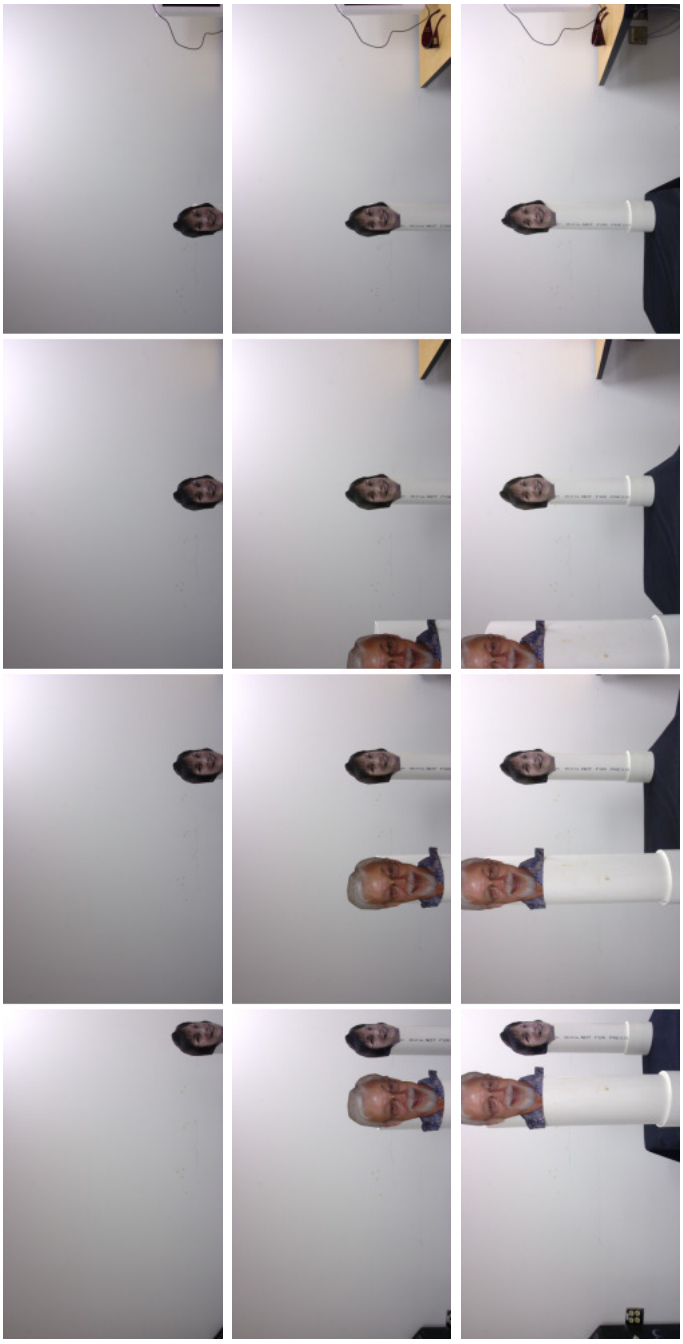
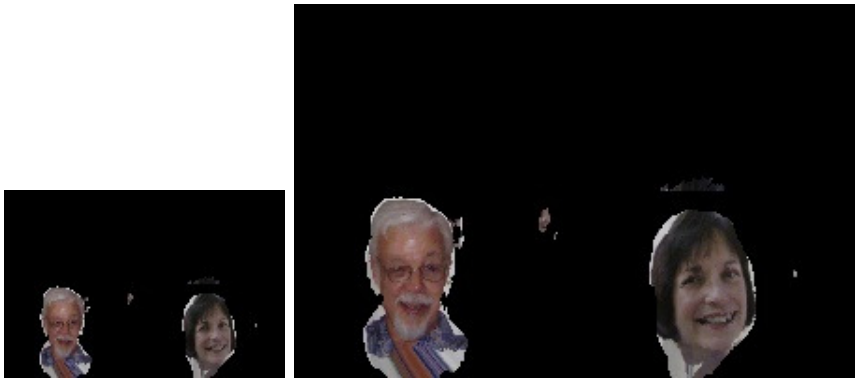
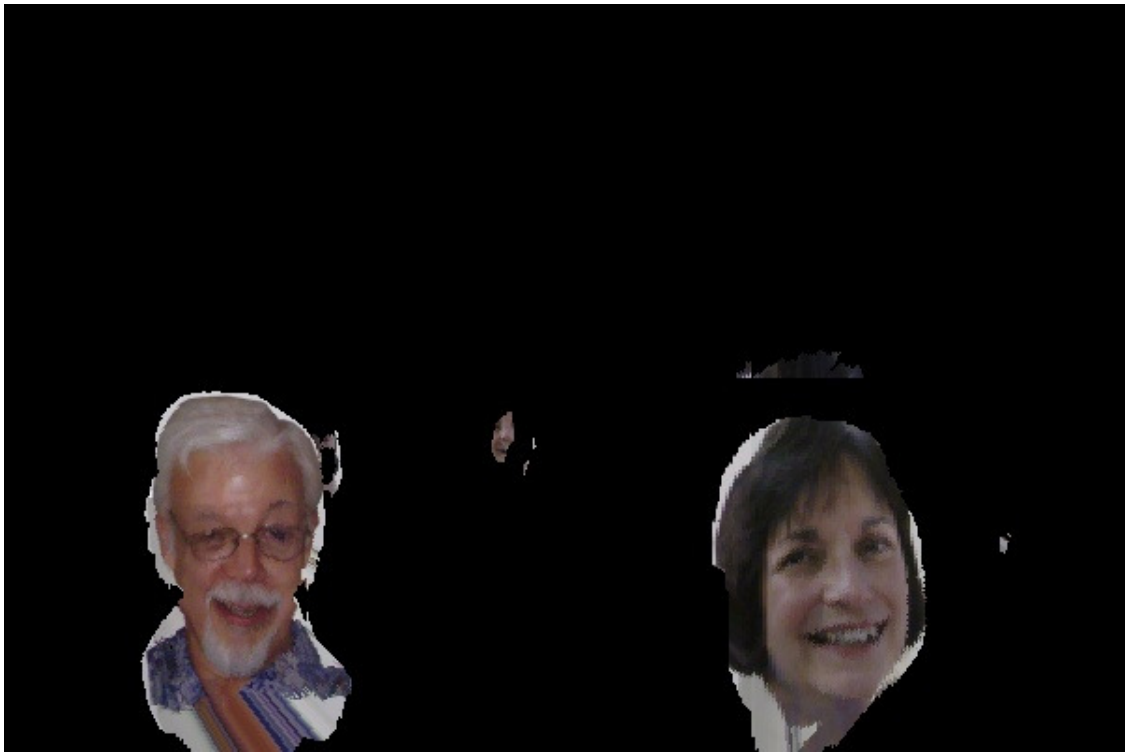


Figure 4.9: The input images for the orthogonal projection



(a) 5ppi (b) 10ppi



(c) 20ppi

Figure 4.10: The final orthogonal projection with three different sampling resolutions

Chapter 5

Future Work

To develop these algorithms into a working large-screen videoconferencing system, this project will need future work in the areas of stereo matching, calibration, and optimizations towards real-time speed.

5.1 Stereo Matching

While area-based matching usually found good matches, depth discontinuities continually posed problems. The final orthogonal projection was a good result because there was no overlap between the two faces used in the scene. Using multiple window sizes for area-based matching should improve results around depth discontinuities, but it didn't help much, possibly because a large window was initially used. As long as a large window is used at all, large depth discontinuities will continue to pose a big problem.

One way the algorithm might be improved is by taking advantage of known information about the scene. If a very good match is found in two images, it's likely that the areas around that point will also match and be at the same depth. This kind of information could allow matches that aren't as strong, but strong enough when combined with the high likelihood of neighboring matches.

Likewise, a region growing algorithm could be implemented. It would use confident matches to

create contiguous regions of similar depth. The regions then grow on the basis of adjacent pixels of similar color and depth, partitioning the scene into distinct objects with known depths. A region growing algorithm would allow the system to use fewer matches, leaning more heavily on only the best matches.

The use of multiple cameras will introduce new problems. Since our setup uses the same camera, all of the images are consistent in their white balance, color, and exposure. Switching to multiple cameras would introduce new matching problems and necessitate normalization before area matching.

Other constraints might be used in the future as well, such as the ordering constraint. Certain matches could be thrown out or made stronger based on the ordering constraint for an object, leading to a more confident depth map and orthogonal projection. The similarity constraint could be improved through the use of Chambon and Crouzil's color correlation methods. Sometimes two grayscale regions will be incorrectly matched if they have a similar structure but different colors. Color correlation can help eliminate potential false matches by forcing correct matches to have the same color.

Matching could also be improved if the background was accurately removed. The interest operator helped to eliminate most of the background, but the background immediately surrounding the objects in the scene still qualified for matching. A "green screen" type of background removal would work, but would be impractical for the application of a large-screen videoconference. However, in the event that this project changes from still-image to video, the background could be detected and removed by sensing which objects in the scene move.

5.2 Calibration

Alignment was the biggest problem with the orthogonal projections. While some of the alignment problems were overcome by building the aluminum stand, the assumption that all cameras are precisely positioned and aligned will need to be dropped in any practical application. If small cameras are placed at the intersections of the monitors on a large display, the location of the camera could be positioned precisely. But it would be impossible to align their rotation to be exactly the same. Points in space will need to undergo a transformation to find their location relative to each camera. The problem won't be finding k in the perspective equations, but it will be finding the

location relative to each camera. Relative points change as the camera rotates, because the points at $(0, 0)$ are along the line coming straight out of the camera's lens. Consequently, camera calibration will be a key problem.

5.3 Optimizations

For videoconferencing to work, the algorithm needs to run faster. Parallel computation would probably be mandatory for this to happen. If one computer or processor was dedicated to processing each rectangular region between four cameras, the each image could be generated quickly. It's also possible that the program could be optimized to take advantage of continuity between video frames. The scene doesn't change much from frame-to-frame, so matches made in the previous frame could be used to help find the orthogonal projection for the current frame quickly.

REFERENCES

[ACR96] M. Adjouadi, F. Candocia, and J. Riley. Exploiting Walsh-based attributes to stereo vision. *IEEE Transactions on Signal Processing*, 44:409–420, February 1996.

[Bak82] H. Baker. Depth from edge and intensity based stereo. Technical report, Stanford University, September 1982.

[BB82] D. Ballard and C. Brown. *Computer Vision*. Prentice-Hall, Inc., 1982.

[BF82] S. Barnard and M. Fischler. Computational stereo. *ACM Comput. Surv.*, 14(4):553–572, 1982.

[BI99] A. Bobick and S. Intille. Large occlusion stereo. *Int. J. Comput. Vision*, 33(3):181–200, 1999.

[BVZ97] Y. Boykov, O. Veksler, and R. Zabih. A variable window approach to early vision. Technical report, Cornell University, Ithaca, NY, USA, 1997.

[Can83] J. Canny. Finding edges and lines in images. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[CBVY00] J. Cassell, T. Bickmore, H. Vilhjálmsson, and H. Yan. More than just a pretty face: affordances of embodiment. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 52–59, New York, NY, USA, 2000. ACM Press.

[CC] S. Chambon and A. Crouzil. Color stereo matching using correlation measures. http://www.irit.fr/ Sylvie.Chambon/PUBLICATIONS/csimta2004.ps.gz. Last accessed 17 May 2007.

[CHRM96] I. Cox, S. Hingorani, S. Rao, and B. Maggs. A maximum likelihood stereo algorithm. *Comput. Vis. Image Underst.*, 63(3):542–567, 1996.

[CM92] S. Cochran and G. Medioni. 3-D surface description from binocular stereo. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(10):981–994, 1992.

[Cox94] I. Cox. A maximum likelihood n-camera stereo algorithm. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 773–779, 1994.

[Fua93] P. Fua. A parallel stereo algorithm that produces dense depth maps and preserves image features. *Machine Vision and Applications*, 6:35–49, 1993.

[GY05] M. Gong and Y. Yang. Fast unambiguous stereo matching using reliability-based dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:998–1003, June 2005.

[HIG02] H. Hirschmüller, P. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *Int. J. Comput. Vision*, 47(1-3):229–246, 2002.

[HS89] R. Horaud and T. Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1168–1180, 1989.

[IBL00] Y. Ivanov, A. Bobick, and J. Liu. Fast lighting independent background subtraction. *Int. J. Comput. Vision*, 37(2):199–207, 2000.

[Inc] The Mathworks Inc. `http://www.mathworks.com/access/helpdesk/help/toolbox/images/corr2.html%`. Last accessed 17 May 2007.

[KO94] T. Kanade and M. Okutomi. A stereo matching algorithm with an adaptive window: Theory and experiment. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(9):920–932, 1994.

[Lev06] M. Levoy. Light fields and computational imaging. *Computer*, 39:46–55, August 2006.

[LH96] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, New York, NY, USA, 1996. ACM Press.

[MP77] D. Marr and T. Poggio. A theory of human stereo vision. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.

[RC98] S. Roy and I. Cox. A maximum flow formulation for the n-camera stereo correspondence problem. *Computer Vision*, 27:492–499, January 1998.

[RGL04] A. Roman, G. Garg, and M. Levoy. Interactive design of multi-perspective images for visualizing urban landscapes. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 537–544, Washington, DC, USA, 2004. IEEE Computer Society.

[Rui04] Y. Ruichek. A hierarchical neural stereo matching approach for real-time obstacle detection using linear cameras. In *IEEE '03: Proceedings of Intelligent Transportation Systems*, volume 1, pages 299–304, 2004.

[Sch89] R. Schalkoff. *Digital Image Processing and Computer Vision*. John Wiley & Sons, Inc., 1989.

[SS97] K. Swaminathan and S. Sato. Interaction design for large displays. *Interactions*, 4(1):15–24, 1997.

[Sun02] C. Sun. Fast stereo matching using rectangular subregioning and 3d maximum-surface techniques. *Int. J. Comput. Vision*, 47(1-3):99–117, 2002.

[XWZ96] Y. Xiong, D. Wang, and G. Zhang. Integrated method of stereo matching for computer vision. In Andrew G. Tescher, editor, *SPIE Proc. Applications of Digital Image Processing XIX*, volume 2847, pages 665–676, 1996.

[ZW94] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual corresponence. In *Proceedings of the European Conference of Computer Vision 94*, pages 151–158, 1994.

VITA

Ryan Edward Silva
300 McDonald Street
Apartment 24B
Blacksburg, VA 24060

Education

Mr. Silva graduated in spring 2006 from Virginia Tech with a bachelor's degree in computer science, and a double major in mathematics (applied discrete concentration). From fall 2005 to spring 2006, he was a 5-year B.S./M.S. student in computer science, taking four graduate courses while finishing his bachelor's degree. After finishing his undergraduate work, he spent his final year as a student working on his thesis and finishing the rest of his required graduate courses.

Honors and Activites

Mr. Silva was a member of Intervarsity Christian Fellowship for five years and served as their web administrator for four. He is a member of Upsilon Pi Epsilon, where he served as president in spring 2007. He also spent three years as a member of Computer Science Community Service. After four years as an undergraduate, he received nine scholarships, was on the dean's list every semester, and graduated summa cum laude. At graduation, he received the Department of Computer Science service award. He is an Eagle Scout in the Boy Scouts of America, and a member of the National Eagle Scout Association. He enjoys playing soccer, hiking, camping, and cooking.

Personal

Mr. Silva was born in Yonkers, New York on December 6, 1983. He lived in Yonkers for five years before his family moved to Virginia. He spent most of his youth at his home in Midlothian, Virginia. There, he was homeschooled from fourth through eighth grade and afterwards attended Midlothian High School. In fall 2002, he began his undergraduate career at Virginia Tech, where he was a student until 2007. He is currently employed by IBM and will be moving to Boston, Massachusetts. He is marrying Anna-Laura Kroesen on June 22, 2007.