Stereo Vision using Block Matching

Christopher Munroe

Christopher_Munroe1@student.uml.edu

***Abstract*— Digital imagery is everywhere in today's modern world. With cameras everywhere, and on everything, stereo vision is all the more usable. In this paper, I discuss my implementation of a block matching algorithm, and go over the fundamentals of stereo vision. My approach is very slow, but provides reasonably accurate results in typical environments.**

I. Introduction

Computer stereo vision is the extraction of 3D information from digital images. Depth information has an endless amount of applications in all fields. It is an image, but with additional information. For example, robotics uses depth information to construct 3D maps of areas using SLAM algorithms (Fig. 1). Self driving cars depend on depth information to navigate the road, however, its gathered with LIDAR and not stereo. Google Maps allows you to explore the world in 3D using satellite imagery. Moving forward, virtual reality could be the next big consumer of depth information!

While there are sensors such as LIDAR which are specifically designed to detect 3D information, digital imagery has an overwhelming presence in today's modern world. Because the camera population vastly outnumbers the LIDAR population, creators prefer solutions using digital imagery, so they don't have to attach a new sensor to their product. Not only are they cost effective due to their ubiquity, cameras have more uses outside of depth sensing. The XBOX Kinect, a structured light sensor which uses a modified version of stereo, is a great example of the power of price because of its widespread adoption in the research community.

II. Background

My take on stereo vision is not novel. I have always been fascinated by stereo vision, and wanted to learn more about it. The stereo vision algorithm I attempted to recreate uses a method called block matching [2]. The basic version of this algorithm I created has some noise, and state of the art algorithms find ways to smooth things out. For example, Ohta and Kanade discovered that through the use of dynamic programming, pixels can be fit into the line for the best possible ordering to further reduce noise in block matching algorithms [3]. Many algorithms have different ways of improving the speed of the algorithm, with the main solution being depth map density reduction. Unfortunately, I did not manage to implement dynamic programming or significant speed optimizations outside of the standard block matching algorithm. I made an attempt at the dynamic programming smoothing, but I failed to finish the implementation because I had trouble internalizing the concept fully.

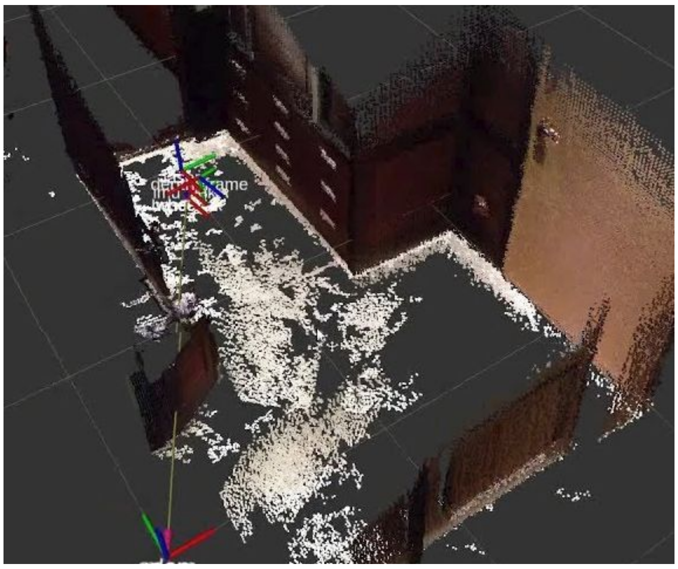
Fig. 1 3D mapping using XBOX Kinect [1]

III. Approach

When two cameras are viewing the same scene from similar perspectives, the images share some pixels. With these matched pixels and the known

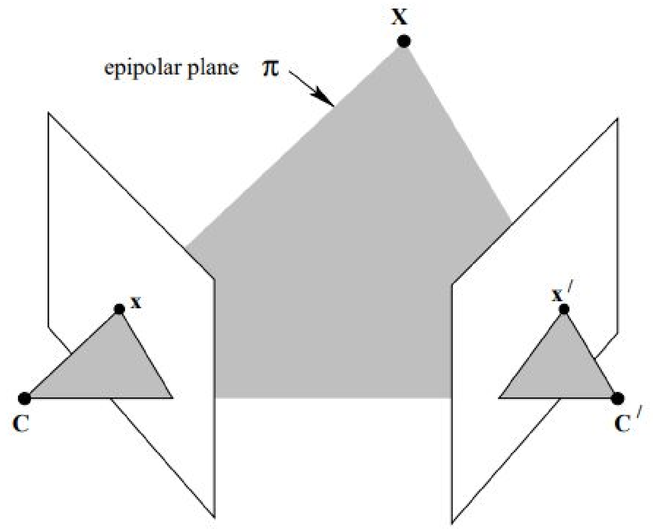

Fig. 2 Epipolar plane visualized

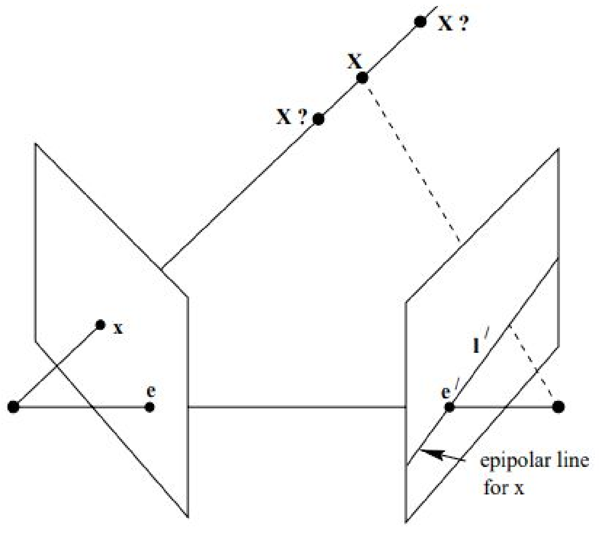

Fig. 3 Epipolar line visualized

intrinsic properties of the stereo setup, we can triangulate the distance from the cameras to the correspondence. Of course, there are also a lot of pixels that are not shared between the images, so the real task is finding the correspondences. Naively, you may first think to scan through the whole image for each pixel in the original image, but with high resolution images it is very costly. Epipolar geometry will help us narrow down our search.

A. Epipolar Geometry

Epipolar geometry is the intrinsic projective geometry between two views. By analyzing the epipolar geometry using the cameras' internal parameters and relative poses, we can ease our search problem immensely. Rather than scanning through the entire image, we can use the intrinsic projective geometry to find a scan line that the pixel must fall under.

In Fig. 2, two cameras **C** and **C'** can see the point **X** in three-dimensional space. Pixels **x** and **x'** are the location of the point within each camera's two-dimensional image. The rays projected by **x** and **x'** intersect and form what's called the epipolar plane $\boldsymbol{\pi}$.

But practically speaking, what if we don't know **X** and the corresponding pixel **x'**? Fig. 3 illustrates this geometry problem. Despite having two unknowns, we can still form an epipolar plane from **C**, **C'**, and **x**. If **X** must be along the ray casted by **x**, then **x'** must lie along the intersection of $\boldsymbol{\pi}$ and the image plane of **C'** which is called the epipolar line. With this method, for each pixel in the left image, we calculate the epipolar plane, then calculate the epipolar line, and lastly we search along the epipolar line for a corresponding pixel. Calculating the intersection for each pixel and traversing the image with a crooked line segment is messy and inefficient. Diagonal lines touch a lot more pixels than a perfectly straight horizontal line which slows everything down.

Instead of calculating and traversing diagonal epipolar lines we can transform the images before doing any correspondence search. The goal of the transformation is to make every epipolar line in the image be perfectly horizontal. That way, when we pick a pixel in the left image to perform our correspondence search, we can skip the plane

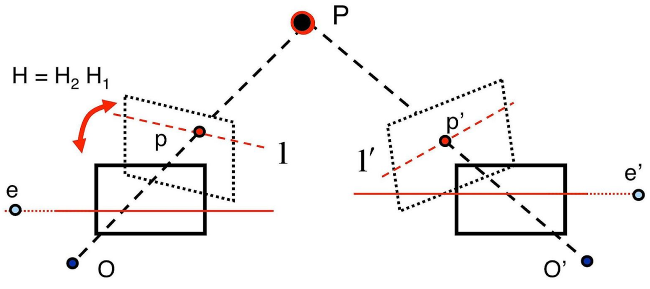

Fig. 4 Image rectification

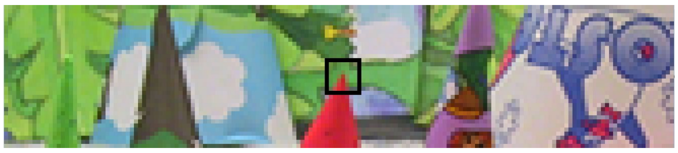
Fig. 5 Left Image

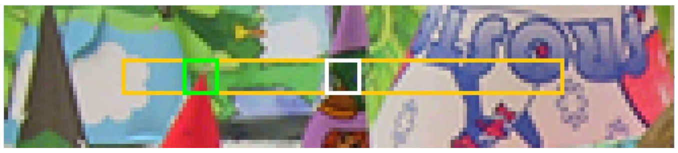
Fig. 6 Right image block matching search

intersection part and assume the epipolar line is the corresponding row in the right image. This transformation process is called image rectification (Fig. 4). The subtleties of the transformation are beyond the scope of this work, but basically we can use the rotation matrices of camera perspectives to apply an affine transformation on the images.

B. *Block Matching*

Now that the images are rectified, we can go into the block matching algorithm I implemented. The basic concept is: instead of comparing a single pixel to another to find correspondences, we compare a rectangle of pixels to another. In Fig. 5 we have selected a block in the left image, denoted by the black rectangle, and now need to find the corresponding block in the right image. In the right image (Fig. 6), we outline the scan area in orange. The scan area is centered on the original coordinates (indicated by the white box) and extends left and right for a variable amount of distance. The user of the algorithm can tweak this distance, and it is called the maximum disparity. We compare the original template from the left image with the sliding window block in the right image using an operation called SAD (Fig. 7).

SAD, or sum of absolute differences, is a fast and effective method for scoring correspondences. The operation subtracts each pixel in the template from the corresponding pixel in the block, and then takes the absolute value. Then we sum all the differences and are left with the score. The lower the SAD, the more similar the blocks are. To find the best match, we compute the SAD for each block along the scan region, and choose the block that has the lowest SAD for our correspondence.

To find the disparity of the correspondence, we find the difference between the horizontal coordinates of the blocks. The disparity distance uses pixels as units. The larger the disparity, the close the object is. After performing this entire process for each pixel in the left image, we end up with a two-dimensional array of disparities. To visualize the disparities, we divide each disparity by the maximum disparity and then map those intensities to colors (Fig. 8). To convert a disparity into an actual distance, the formula is:

$$Z = \frac{Bf}{disparity}$$

Where B is the baseline, or the distance between the cameras, and f is the focal length (assuming the focal length is the same in both cameras).

IV. Dataset

This algorithm does not use a dataset to learn from. The only requirement for my block matching algorithm is that the images provided are rectified.

The evaluation dataset is the 2014 Middlebury stereo datasets from Middlebury College. The set includes 33 rectified image pairs with the intrinsic properties of the stereo setup, such as focal length and baseline distance.

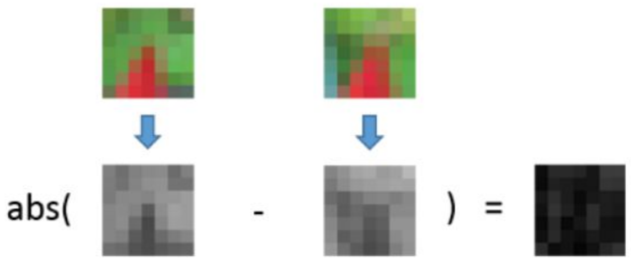

Fig. 7 Block comparison using SAD

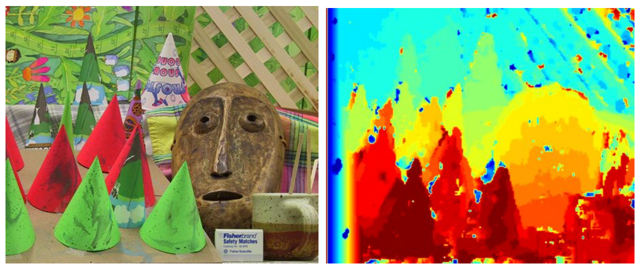
Fig. 8 Colorized disparities

V. Evaluation

To evaluate my algorithm on the aforementioned datasets, I hooked up my code to Middelbury's stereo vision evaluator [5]. The evaluator runs my algorithm on the stereo pair dataset and compares the disparity map against a ground truth. The main metrics the evaluator tracks is time, and "badX", which is the percentage of bad pixels with disparity error > X pixels. I evaluated my algorithm with the metric bad1.0 because that seemed to be the standard threshold. I ran my algorithm for 7 stereo pairs which had dimensions around 700x500 and recorded the results in Fig. 9. I have also provided the stats for a state of the art algorithm, ELAS, for reference.

ELAS is nearly 500x faster than my algorithm, so my implementation of block matching certainly has many optimizations that could be done. As far as accuracy goes, my algorithm can compete reasonably well with the state of the art for particular scenes (Fig. 10). A lot of my disparity error in this scene appears to be created by my failure to smooth in occluded areas. This could probably be solved with the dynamic programming smoothing.

For other scenes, my algorithm fails terribly. In the PianoL stereo pair, one of the images has altered lighting. As you can see in Fig. 11, my block matcher struggles to find any correspondences in the altered lighting unless it's very obvious. All the images with altered lighting end with a capital L or E in Fig. 9, and on these images you can see the algorithm scores very poorly for disparity accuracy.

VI. Conclusion

Overall, I think my basic block matcher held up adequately in terms of accuracy for images with similar lighting. To improve the accuracy further, dynamic programming would be the next step. In the images with different lighting, perhaps I could compare the overall lighting between the images, and apply a filter to make the lighting more similar. As for the speed of my algorithm, there is a lot of work that needs to be done. The speed of my block matching implementation is its biggest shortcoming. To address this, image pyramiding could be the first step towards optimization. However, ELAS must be doing something radically different to achieve such an impressive speed.

VII. Team roles

There was only one author for this project who did everything, Christopher Munroe.

References

[1] Robotics Weekends, *RTABMAP 3D mapping on mobile robot with Kinect 360 - first SLAM test results,* https://www.youtube.com/watch?v=sMx7N5rYUFQ

[2] Koschan, A., Rodehorst, V., & Spiller, K. (1996, August). Color stereo vision using hierarchical block matching and active color illumination. In Pattern Recognition, 1996., Proceedings of the 13th International Conference on (Vol. 1, pp. 835-839). IEEE.

[3] Ohta, Y., & Kanade, T. (1985). Stereo by intra-and inter-scanline search using dynamic programming. IEEE Transactions on pattern analysis and machine intelligence, (2), 139-154.

[4] Lucas, B. D., & Kanade, T. (1981). An iterative image registration technique with an application to stereo vision.

[5] http://vision.middlebury.edu/stereo/submit3/

Mine	**Adrion**	**ArtL**	**Jadepl**	**Motor**	**MotorE**	**Piano**	**PianoL**	**Pipes**
time (sec)	53.82	16.20	49.00	47.50	48.90	41.89	40.67	48.86
bad1.0	61.67	73.33	43.12	28.72	85.26	37.31	83.16	30.09

ELAS	**Adrion**	**ArtL**	**Jadepl**	**Motor**	**MotorE**	**Piano**	**PianoL**	**Pipes**
time (sec)	0.14	0.04	0.13	0.15	0.16	0.14	0.13	0.14
bad1.0	12.07	19.80	34.68	12.49	13.45	22.04	37.52	15.30

Fig.9 Benchmarking for 7 stereo image pairs

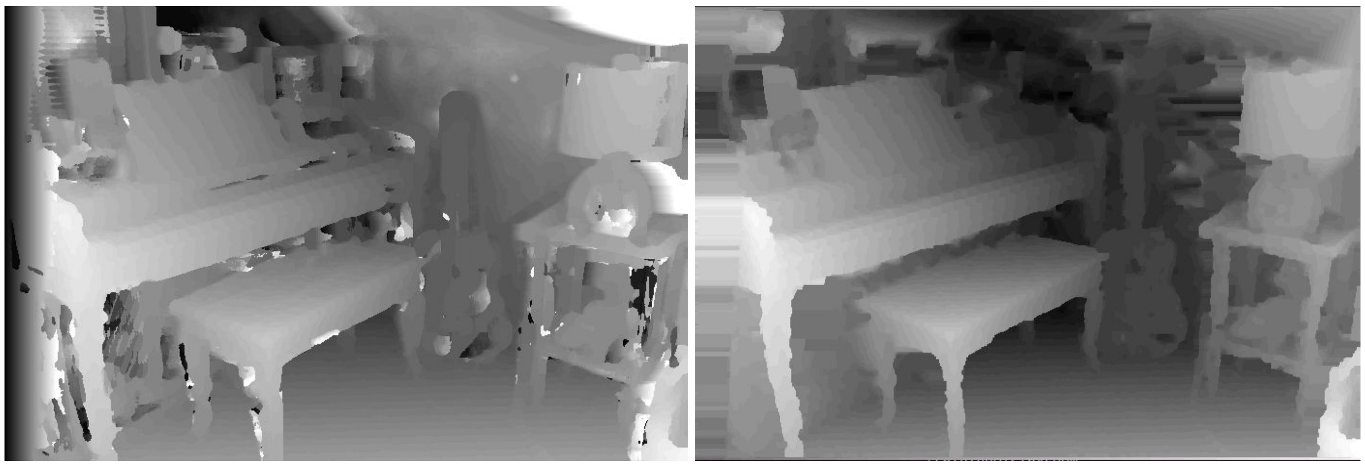
Fig.10 Disparity map for Piano, LEFT: My algorithm, RIGHT: ELAS

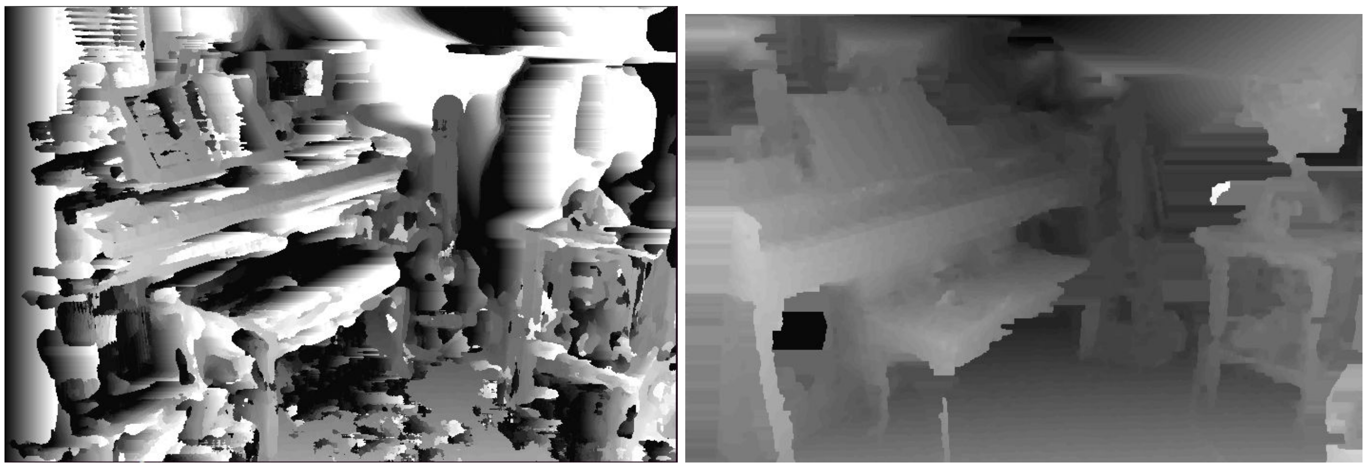
Fig.11 Disparity map for PianoE, LEFT: My algorithm, RIGHT: ELAS