Final Project: **In Placid Depths, Latent Light Sparkles**

By Andres Santiago, Anil Parthasarathi, and Gage Mariano

Introduction

The idea for this project originated from a medium article simply titled "How to generate a normal map from an image." Through these normal maps, the author wanted to add the illusion of depth to famous paintings allowing for the users to more closely inspect the brushstrokes and to apply dynamic lighting to the scenes. For normal maps to be generated, a height map (on which to perform finite differences) of the image is required. To circumvent this problem, the author considered the brightness of a pixel to be its height and calculated the normal map based on this, as the author called it, "quick and dirty" method. The results were sufficient for the essentially flat paintings (although even the author admits that it struggles with images with both dark foregrounds and backgrounds); however, we wanted to see if we could recreate a similar or better effect with scenes containing meaningful depth.

Our primary goal was to allow for an arbitrary dynamic light to move throughout the scene, illuminating the crevices and shapes of the scene as if a real flashlight was shining at the given point. To accomplish this, we knew we would need to improve beyond the naive approach of equating height to brightness and that our height map would need to reflect the actual depth of the scene. Our discussion of the possible solutions happened to closely coincide with the lectures on stereo images. Not one to ignore a clear deus ex machina, we readily decided to use stereo images to calculate a disparity map which we would then employ as the height map. The following sections will discuss the details of our project structure and implementations.

Implementation

Our project was implemented in the form of a web application for viewing 2.5D renders alongside a tool for computing disparity maps from stereo images. We used HTML, CSS, JavaScript, and WebGL for the website and Python for the disparity calculation tool.

Our implementation of the disparity calculations began with the most naive approach of performing a linear search along the epipolar line for every pixel in the image to find the minimum SSD of all the patches and establishing a match. Following this concept we achieved the results shown below, ideally closer points should be brighter and points further away from the camera should be darker.

Before discussing the results, we will detour briefly and discuss the different designs we developed to perform this naive calculation. As this is an O(n^3) algorithm, it was very important to vectorize (via Numpy) as many of the operations as possible. Following this logic, we iteratively reduced the number of Python for loops required in the computation. Our initial function used numpy's sliding_window_view to create a view of each image of shape (HEIGHT, WIDTH, window_size, window_size). This allowed us to perform "element-wise" SSD for a given left window against all windows in the right image along the corresponding epipolar line. Argmin could then be used to find the disparity for the given left point. Repeating this for every pixel in the left window, we calculated disparity at each point with only two python for loops. To build upon this, we repeated the exact same sliding_window_view process except, when performing the "element-wise" SSD operation, we handled every pixel in a left image row at once. This was done by using numpy.newaxis to format the left row to have an additional dimension in comparison to the right row, allowing for every pixel in the left to be compared against every pixel in the right patch within one operation, achieving the same output as previously with one Python for loop. The final Python for loop was eliminated by further abusing this numpy.newaxis. Giving both images a new dimension, we are able to compare every pixel in the left image to every pixel along the corresponding epipolar line in the right image, completing our goal of optimizing the number of operations performed via Numpy.

Despite this, experimentally we discovered that, contrary to expectations, the algorithm with the most for loops often performed the best, outperforming the single for loop by several seconds and the zero for loop by almost one minute. The issue and oversight stemmed from the intermediary calculation during SSD which first performs (left - right)^2 before summing. This operation meant that for each left pixel being considered an entire corresponding row of patches was initialized. This led to an array of the shape (HEIGHT, WIDTH, WIDTH, window_size, window_size) in the worst case. This of course pushed memory constraints which the runtime reflects.

Returning to the discussion of the above results generated using these algorithms, we see that high frequency areas (including edges and textured areas like the pottery and parts of the wall) were captured very well. Inversely, low frequency areas such as the face and cone are all reporting a very low disparity. This is because there was no enforced order of pixel matching,

meaning that multiple pixels in the left image could claim to match to one pixel in the right image and also a leftward pixel could match with a rightward pixel while a rightward pixel matched with a leftward pixel. As a result, once the edges have been iterated through and the pixels being handled were firmly within a low frequency area in both images, a pixel often found its match at the same index as itself. To correct this, we need to require that matches are always increasing; this ensures a minimum disparity within low frequency components and a more accurate match.

During our initial research for the project, we came across these lectures slides which cover the seminal stereo algorithm described in this paper. Differing from our previous implementation, this "new" algorithm ensures that all matches are unique and ordered. Following the relative failure of the previous approach, we began implementing this more demanding (of us and the matches) method.

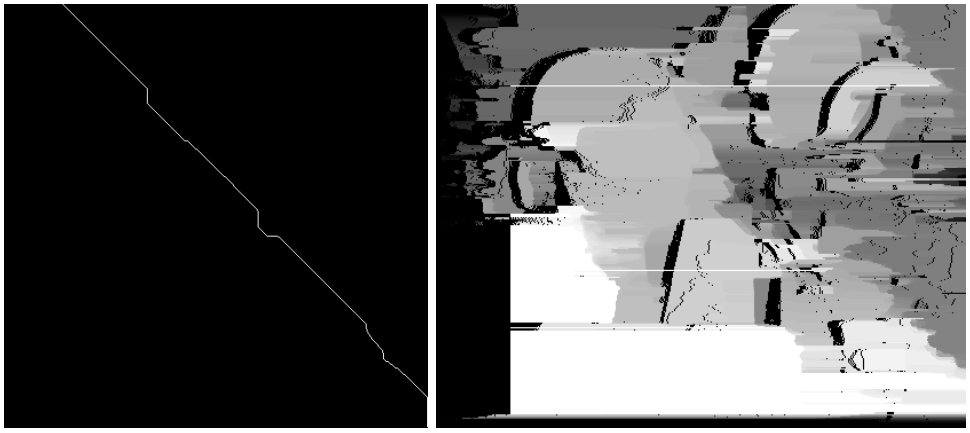
With a dynamic programming strategy, the algorithm works row by row (finding the disparity of every pixel in a row before moving to the next) and consists of two primary stages, building the "score" matrix and backtracing. The score matrix is initialized with a (WIDTH, WIDTH) shape where each column represents a pixel in the left image and each row represents a pixel in the right image; thus, at column=3 and row=5, the distance value between pixel 3 in the left image and pixel 5 in the right image is stored. We originally decided to use SSD for our distance function (as discussed later). To implement this, we utilized numpy's sliding_window_view to create a view where each pixel index is associated with its surrounding patch. With these windows, we used numpy's newaxis and sum to subtract every right patch from each left patch and square the result before neatly summing into the scoring matrix shown below.

We can now begin the actual dynamic programming aspect of the algorithm, but first we must mention the idea of occlusions. Sometimes a left pixel cannot find a reasonable match (or all the reasonable matches have already been taken by preceding pixels). This could be due to the true match being hidden in the other image (a.k.a. occluded). When this happens, we don't

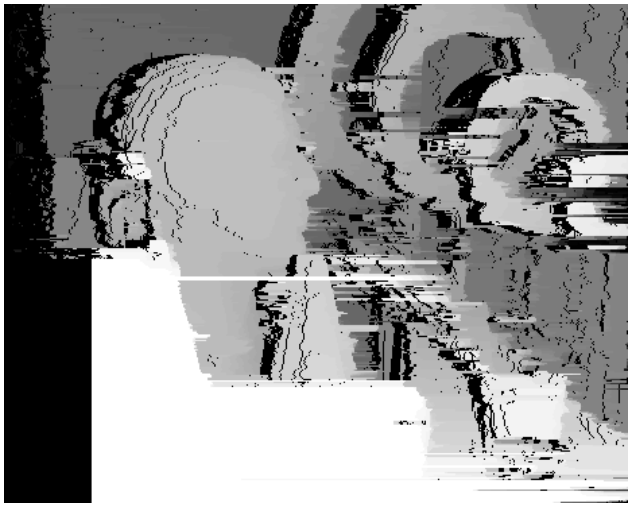
accept a poor match score, instead we simply mark it as occluded (either from the left or right depending on in which image the pixel is hidden), so as not to consume what could be a good match for another pixel. These occluded pixels incur a occlusion cost (discussed later)

With the scoring matrix constructed, we can begin constructing the final scores. Due to the iterative nature of the algorithm, we cannot effectively leverage numpy and must use python for loops. We will work row by row. Within each row, we will only consider a reasonable disparity between the two images. As it is unlikely that the disparity value is the majority of the image. For us, we followed the lectures guidance and (usually) used a range of 64 (roughly 15% of the width). This reduces false positives and runtime. First we initialize the first (base) row with the occlusion cost multiplied by index (this is because these pixels cannot be determined to be right occluded). We now, since we are enforcing an ordering of matches, work diagonally down to the next row, meaning for the second row we start at column = 1 and go to column = 64. At each index, we compute and compare three possibilities: that it is a match, that it is left occluded, and that it is right occluded. To do this, we check whether the preceding left pixel matched with correspondingly preceding right pixel (the pixel directly to the left of the one we are matching), calculating match cost to be the computed dissimilarity at the current pixel plus the score of the previous row and col. To be left occluded means that the matching pixel is already taken, so we take the score of the previous column (same row) plus the occlusion cost. Similarly for the right occlusion, we take the score of the previous row (same column) plus the occlusion cost. The minimum of these three values are stored in the score matrix. This continues until the entire score matrix is populated.

Allowing us to now backtrace and determine the minimal path and the optimal disparity assignments that obey the given constraints. We of course start at the final calculated score, in the bottom-right corner. At each index we determine whether the minimal path would consider the current index to be a match, a left occlusion, or a right occlusion. This is determined by reading and comparing the corresponding scores from the score matrix. The action that leads to the minimal score is followed and recorded. This is repeated until every row (every pixel) has a determined disparity (or is marked as an occlusion). The values along this path (like the one shown below) are returned back to the depth matrix for the given row. The process then repeats for the next row until all disparities are computed.



Clearly the results from this approach are leagues above the previous results. Because of the enforced ordering, low frequency areas have near constant disparities and the accuracy appears higher and more consistent overall. Surprisingly, in exchange, it seems that we lost the one success of the previous implementation and now edges are not always captured. In an attempt to correct this problem, we switched the distance function to NCC (implemented in much the same way as SSD except additionally using np.std and np.mean). This improved the edge detection considerably. This is likely due to how NCC is more resistant to noise and differences in lighting.



In the given result, the occlusions are extremely visible. Around every left edge there is a patch of black values that report no information. To correct this we can fill these occlusions by copying the nearest valid value in the direction they are occluded from. This is performed column by column and direction by direction. First for left occluded elements in each column, we copy the disparity value of the pixel directly to the left. We do the same for right occluded elements, except iterating in reverse and copying from the right.

This approach produces some artifacts when there is a long line of occlusions, but now every pixel (for which there can be a disparity value as some edges values cannot be matched) contain a reasonable disparity.

A lot of the challenge for this (beyond implementing the algorithm itself) came from determining the occlusion cost. The paper mentions a very complex equation that was simply beyond the scope of this project. Initially we tried to set it equal to 1/64 so that it would accumulate at an equal rate to the width of the valid disparities. This often led to an abundance of occlusions. Fortuitously, switching to NCC helped to standardize the constant as NCC always returns a value between [-1, 1] which made testing and reusing occlusion constants across scenes much more plausible. Eventually, we settled on 0.5 experimentally.

Additionally, we found that the maps we produced were prone to overly sudden transitions in disparity. This would result in unnatural effects in the normal maps that did not accurately capture the surfaces within the image. To help overcome this issue, we ran the output of the algorithm through a gaussian blur filter (we tried several low-pass filters including using skimage.transform.resize and subtracting the high frequency components via skimage.filters.sobel, but we found that a simple gaussian had the best results), to smooth it out. This allowed us to reduce the “stepping” effect and improve the later results.

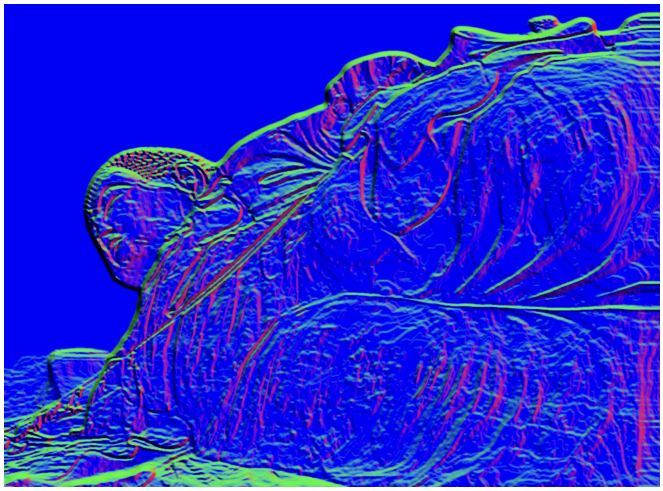
After calculating disparity maps, we had the information necessary to form normal maps while rendering. For the sake of simplicity, we used our Python tool to pre-generate disparity maps for a collection of stereo images found online. However, the rest of our process was able to be handled in real time on the GPU using a WebGL shader.

Rendered inside a JavaScript canvas, this shader takes in several uniform variables. First are a collection of parameters that influence the output. The renderStyle parameter determines whether the canvas will display the default 2.5D render, an image of the disparity map which we generated with our tool, an image of the normal map that is created in the shader, a silhouette shader render made using the normal map, or a cel shaded version of the 2.5D render. The "NIGHTMODE" parameter, determines whether the image should be dark, emphasizing the ability of the light to illuminate the image, or a brighter color that more closely matches the original. The "zFactor" parameter is used to set the z value for the generated normals, with lower values allowing for more depth, and more detailed normal maps – at the cost of occasionally introducing artifacts if set too low. The "DEPTHFACTOR" parameter is the constant value corresponding to focal length * that is divided by each of the disparity values when converting the disparity map into a depth map. Next there is the resolution of the canvas, used for indexing texture coordinates, and then both the original image (the first of the pair) and the disparity map are sent in as textures.

Within the shader we set up constant starting values for specular, ambient, and diffuse. These were determined based on what ended up making the lighting look best. When Night Mode is not enabled, the ambient color is set to the default ambient plus the color of the original image. The diffuse was set to the color of the original image and then slightly dimmed in order to emphasize the color of the light. The position of the light is set to wherever the mouse is placed displaced by 70 units above the canvas in the z axis.

After these rendering details were set up, we began the process of computing the normal map. Taking advantage of the massive parallelization provided by the GPU, the code within the shader handles the work for one pixel within the image. After referencing the position of the current pixel, we were able to generate the normal map by convolving the depth map with Sobel filters. To accomplish this convolution we would first check if each neighboring pixel was within

the image boundary. Then, if so, we sampled the disparity map texture at the index of the neighboring pixel and used this to divide DEPTHFACTOR, then multiplied this by the Sobel filter value for that neighbor and added it to cumulative gradient. Both the gradients in the x and y directions were found in this manner using Sobel filters. Once this was done, the final gradients for the horizontal direction and vertical direction were used as the x and y values for the normal at the given pixel with the aforementioned “zFactor” used as the z value. Then this was normalized to achieve the final normal vector.



A normal map made using an AI generated depth map during testing

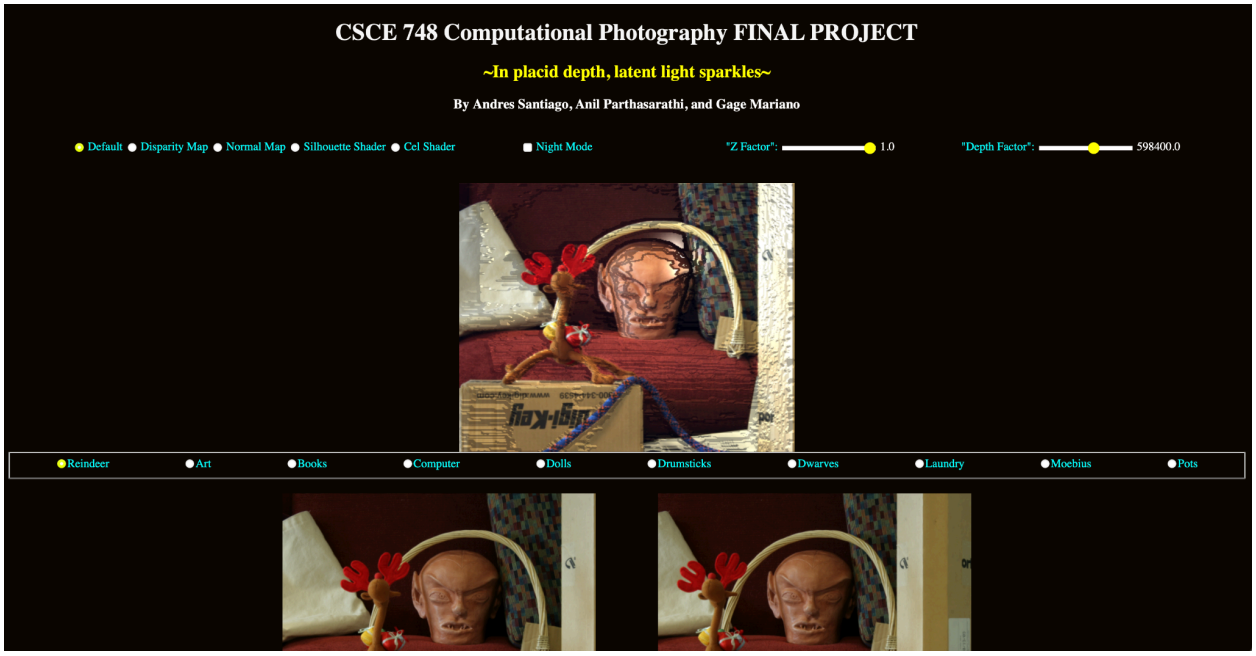
With the normal vector in hand, all that was left was standard rendering logic. If the silhouette shader was enabled, the dot product was computed between the normal and the eye vector. If this value was less than 0.3, then it would be colored black, otherwise it would be colored white. This accomplished an effect where the image is drawn as if it were a black and white sketch that emphasizes important details such as edges.

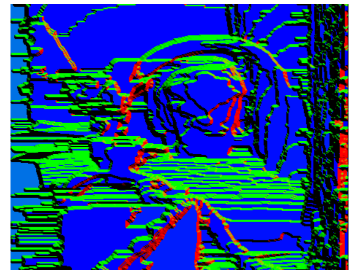
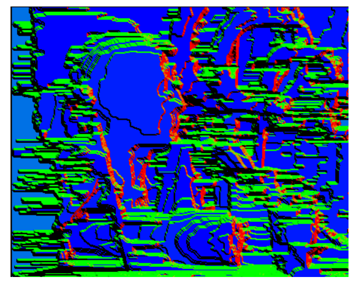
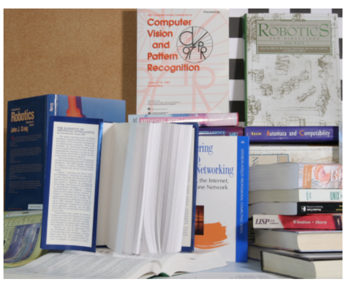
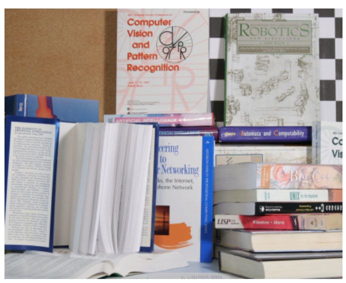
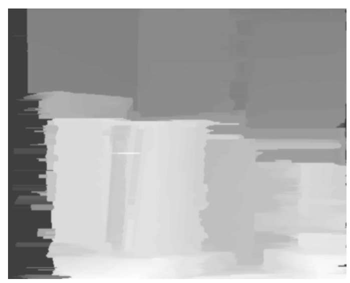
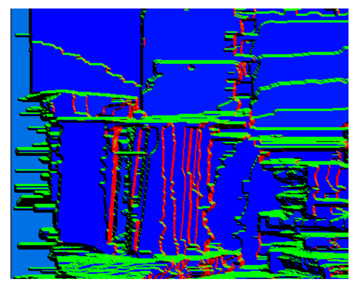
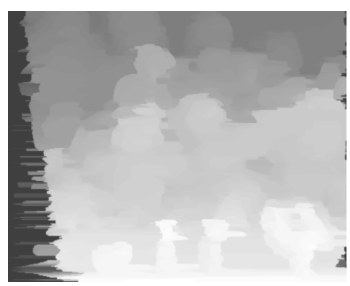
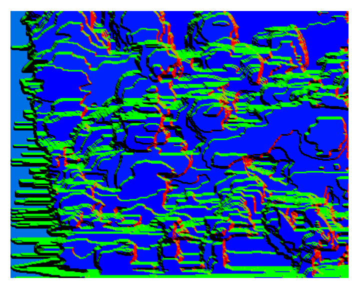
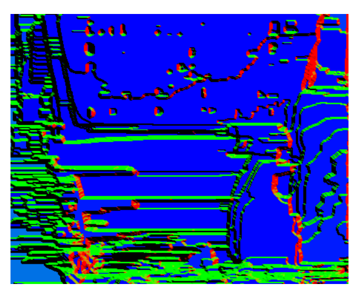
An example of cel-shading in action using the normal map above

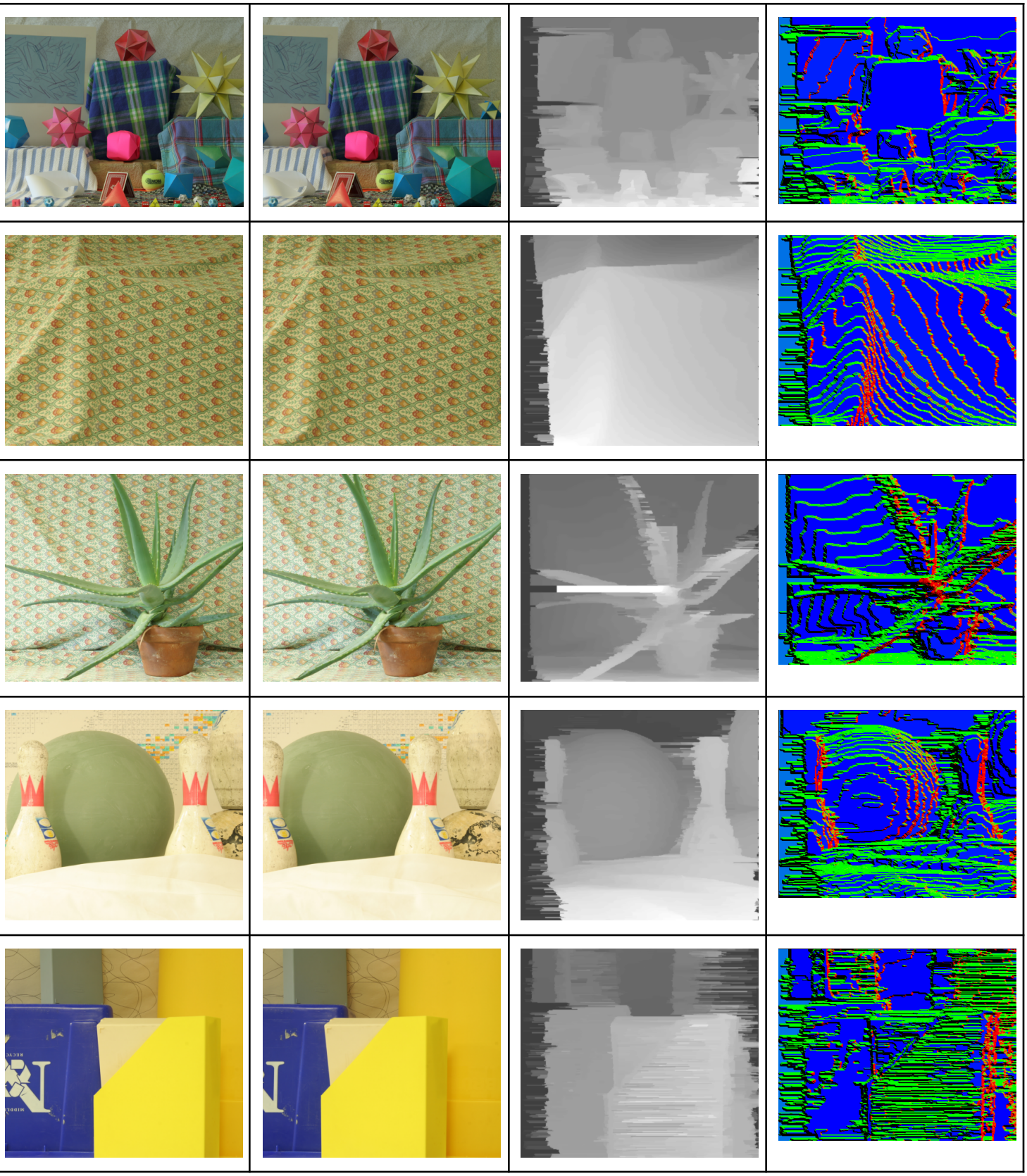
Otherwise, Blinn-Phong shading is employed using the color values, the normal, the eye position, and the position of the light to color in the pixel and apply lighting effects. For the default render mode, the final pixel color is set to the result of the Blinn-Phong computation. Otherwise, when cel-shading is applied, a similar approach to silhouette shading occurs where the dot product of the normal and the eye is calculated then depending on the resulting value, the pixel is colored differently. For low values (such as edges), the pixel is colored black, which outlines all the figures within the image. For higher values there are a series of thresholds where the color gradually transitions. This ultimately achieves a cartoonish, stylized, effect where the image looks more like a 2D painting.

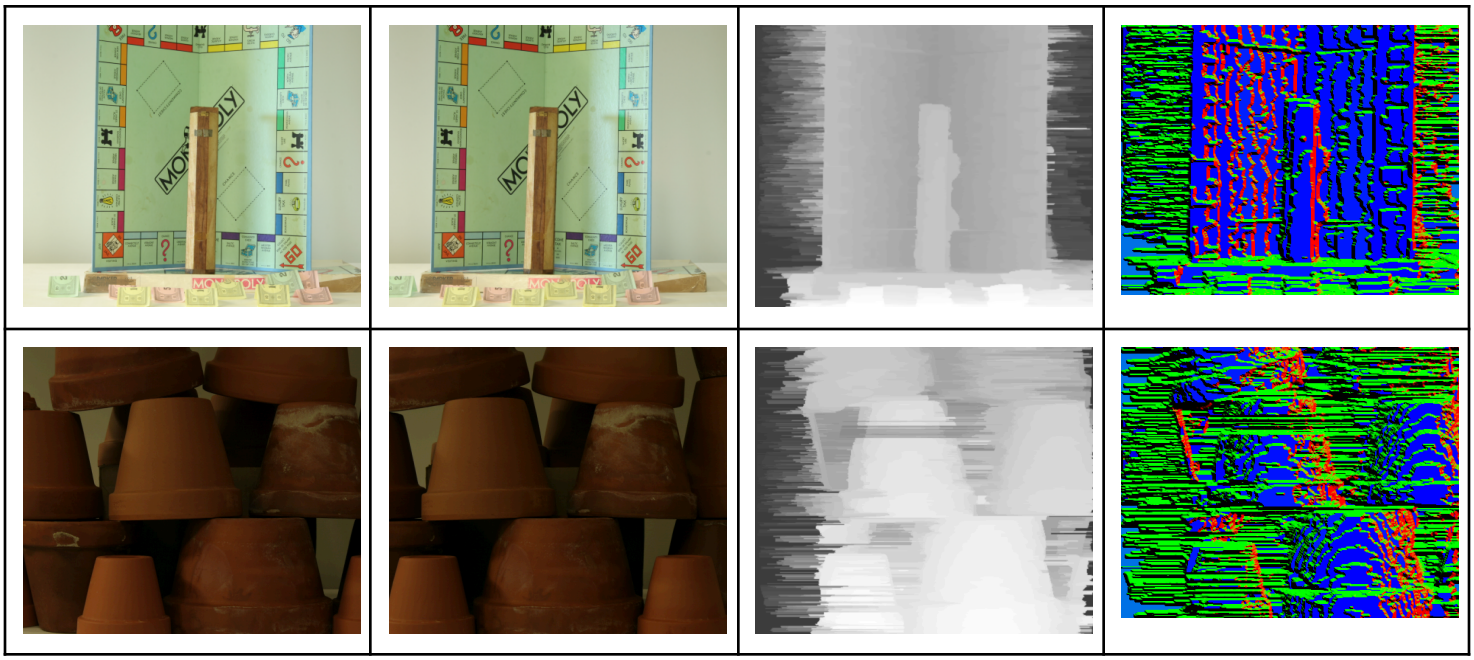
Results

While we have included pictures further down for the sake of conciseness, we strongly believe that the end results of our project are best seen in motion. Please check out this short video to see how it turned out: [2.5D Rendering Demonstration Video]



Left Image	Right Image	Disparity Map	Normal Map
			
			
			
			
			





How to Use

If you would like you can try out our project yourself:

Visit ipdlls.netlify.app/frontend/ to interact with our application. It utilizes precomputed disparity images to shade the scene. The top inputs let you change how the scene is shaded:

- Default - Renders the scene with a reflective light source using the computed normal map
- Disparity Map - Displays the precomputed disparity image
- Normal Map - Displays the computed normal map
- Silhouette Shader - Shades edges and enclosed regions
- Cel Shader - Shades edges and enclosed regions using cel shading
- Night Mode - Dims Default and Cel Shader scenes
- “Z Factor” - Adjusts depth of light to scene, 0 = parallel to scene, 1 = unit distance from scene
- “Depth Factor” - Normalizes scene depth from [0,Depth Factor) to reflect real world measurements.

The bottom input allows you to switch between preselected image pairs and their corresponding disparity map in the rendering canvas.

To run the disparity map calculation to get your own disparity map, you first need a pair of parallel images of the same scene named left.png and right.png. Then run the following to generate the map in the same directory as the input image pair:

```
python3 main.py “path to directory containing left.png/right.png” “window size (must be an odd integer greater than 3)”
```

References

Collins, R. (n.d.). *Lecture 9: Surface orientation and reflectance* [Lecture notes]. Penn State CSE486. https://www.cse.psu.edu/~rtc12/CSE486/lecture09.pdf

Cox, I. J., Hingorani, S. L., Rao, S. B., & Maggs, B. M. (May 1996.). *A maximum likelihood stereo algorithm*. NEC Research Institute; Carnegie Mellon University. https://www.cs.umd.edu/~djacobs/CMSC426/Slides/stereo_algo.pdf

Kruschwitz, A. J. (2021, October 9). *How to generate a normal map from an image: The quick and dirty way*. Medium. https://medium.com/@a.j.kruschwitz/how-to-generate-a-normal-map-from-an-image-the-quick-and-dirty-way-36b73a18f1f1

Wang, Y., Gonen, O., & Akleman, E. (n.d.). *Gradient domain rendering*. Texas A&M University. https://people.tamu.edu/~ergun/research/2Drendering/article.pdf