Depth map estimation for plenoptic images

Michael Hansen Eric Holk

December 16, 2011

Abstract

Depth map estimation is a prerequisite for useful plenoptic rendering algorithms, such as all-in-focus rendering and depth-based matting. We provide a simple GPU-based algorithm that estimates the depth of each pixel in the rendered image as well as the uncertainty of that estimate. We perform several rendering experiments using depth information. Finally, explore what factors harm the accuracy of depth estimation and discuss potential improvements to our algorithm.

1 Introduction

Plenoptic photography enables exciting opportunities for the post-processing of digital photos and videos. Using special software, it is possible to do real-time refocusing and viewpoint adjustment after the photo has been taken. Plenoptics also enable effects that are impossible with a traditional camera, such as rendering with an effectively infinite depth of field. These effects are made possible by capturing the scene's full radiance function; plenoptic cameras capture color as a function of position *and direction*, whereas traditional cameras only capture a single color per position. While the benefits to traditional photography are obvious, plenoptic imaging techniques have also attracted the attention of movie producers to expand the possibilities in 3D filmmaking.

High quality plenoptic rendering often depends on having depth information for a scene. Traditional stereo depth estimation, like that done by human eyes, uses the difference between the images in both eyes to judge distance. Since it is possible to render at least two viewpoints in a plenoptic image, traditional stereo depth estimation applies easily. Yet, this discards the rich information present in the scene's full radiance function. By using the many *microimages* present in the plenoptic *lightfield*, it is possible to do more sophisticated depth estimation. In this paper, we explore techniques for estimating the depths of all pixels in a rendered image, and experiment with several depth-based rendering algorithms.

We make the following contributions.

- We demonstrate that doing GPU-based computations from the start enables rapid experimentation without requiring significantly more effort than CPU-only approaches. (Section 3.1)
- We generalize a traditional stereo depth algorithm to plenoptic lightfield images and show that even this simple method produces usable results. (Section 3.2)
- We present several rendering experiments that incorporate depth information. (Section 4)
- We explore areas in which our algorithm produces inaccurate results. Based on this, we discuss techniques for determining uncertainty in our depth maps and look towards the potential for using it to develop more accurate depth maps. (Section 5)

2 Background

In this section we introduce the basic ideas behind plenoptic photography and depth estimation.

2.1 Plenoptic Photography

Traditional photographs capture only a single color for a given position in a scene. This discards the directional component of light. Objects reflect light in many different directions, and thus light rays from multiple objects may arrive at the same point on a sensor, causing blurring in the final image for objects that are not in the plane of focus. In this paper, we consider a plenoptic camera, which is designed to capture the directional variation of light as well. The representation of the light in a scene in terms of position and direction is known as the scene's *radiance*.

We can represent the radiance as a function $r(q, p)$, which is the value of light at position q and direction p. Value is typically the red, green and blue color components, but may represent other aspects such as polarization. The parameters q and p are both two dimensional vectors, representing the position on the camera sensor and the direction of the given light ray. It is common to use a one dimensional simplification, where q and p are both scalar values.

Using this characterization, we can represent an image like would be captured by a traditional camera as follows:

$$I(q) = \int_p r(q, p) \tag{1}$$

By capturing the full radiance function instead of a single image, we can computationally simulate various optical elements and render many images from a single lightfield. For example, we can computationally change the plane of focus or the point of view. The ability to render different points of view leads to the ability to calculate depth information for the rendered image.

Capturing full radiance function is accomplished by a plenoptic camera, which is illustrated in Figure 1. The camera is similar to a traditional camera, except that a *microlens array* has been inserted immediately in front of the sensor. The additional lenses allow us to recover the directional variation in the captured light. Figure 2 shows a close-up of the raw radiance capture. Notice how the same object appears in multiple microimages but in slightly different positions. This *disparity* between corresponding objects in the microimages is what allows us to estimate a depth map for the scene.

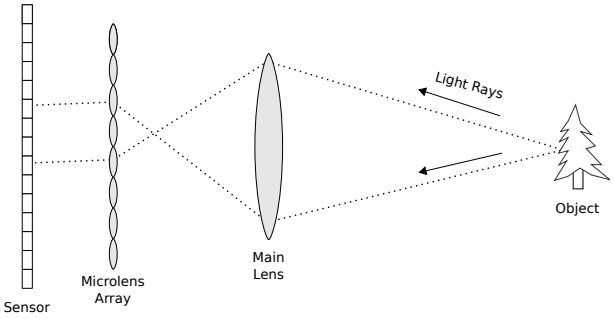


Figure 1: A plenoptic 2.0 camera with the microlens array behind the main len's focal plane.

Figure 2: A crop of a raw lightfield capture. Each of the nine smaller images are referred to as a microimage.

There are two main variants of plenoptic cameras. The "Plenoptic 1.0" camera focuses the microlenses at infinity. This causes the microimages to have a uniform color for objects that are at the plane of focus, which needlessly sacrifices spatial resolution. The "Pleonoptic 2.0" camera moves the camera sensor such that the microimages are focused, thereby enabling the rendering of much higher resolution images [5].

We can model camera lenses and other optical el-

ements using a matrix. These matrices represent various shearing transforms on the 4D lightfield (q_x, q_y, p_x, p_y). By computationally shearing the lightfield after the fact, we can refocus the image. This is most easily accomplished by "rendering at an angle" through the lightfield, which in four dimensions corresponds to blending together patches from adjacent microimages (see Figure 3 for an example). The amount by which the position of the patch varies between microimages is known as the slope, and different slopes correspond to different image planes in focus.

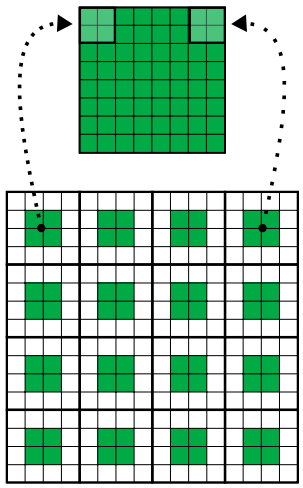


Figure 3: Normal plenoptic rendering.

As shown in Equation 1, rendering an image corresponds to integrating over a range of directions at each point. When we apply this integration process to the full lightfield capture from which Figure 2 was taken, we can produce an image like the one in Figure 4. This image has a shallow depth of field because the integration included a wide range of directions.

2.2 Stereo Depth

Humans use stereo vision to approximate the depth of objects in scene. This is modeled by computing the *disparity*, or the relative separation, between objects in each eye. Far-away objects will appear at virtually the same place in each eye (a low disparity), while objects that are close-up will be in very different positions on each eye (a high disparity). We illustrate this process in Figure 5. In a digital image, disparity is computed on a pixel-by-pixel basis, resulting in a pixel-wise depth map. Successfully determining the disparity between pairs of pixels in two digital images, one from the "left eye" and one from the "right eye," requires an approximate solution to the *correspondence problem*, which is to determine what pixels from the left and right images correspond to the same point in the scene.

Figure 4: An image rendered from a plenoptic lightfield capture.

There is a circular dependence between disparity and correspondence: computing the disparity between two pixels requires knowing which pixels correspond to each other, and determining which pixels correspond requires knowing their disparities! Depth estimation algorithms address this issue by testing for the strength of correspondence at varying disparities. The strongest correspondence is considered to be the correct disparity, and from this disparity the correct depth is derived.

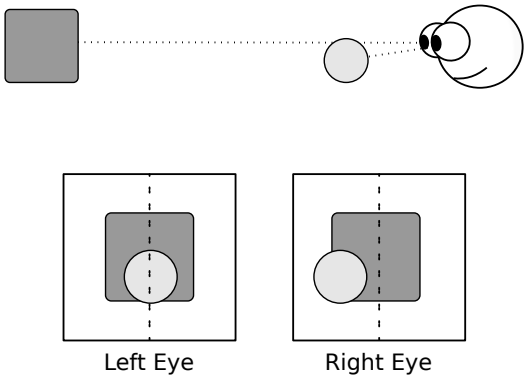


Figure 5: Traditional stereo vision. The disparity between the same object in the left and right eyes can be used to estimate the object's distance or depth.

```
void main() {
  vec2 q =
    gl_TexCoord[0].st / micro_size;
  vec2 p = floor(q);
  vec2 offset = q - p;
  vec2 shift = vec2(shift_x, shift_y);

  gl_FragColor = texture2DRect(
    lightfield,
    p*micro_size+shift+(slope*offset)
  );
}
```

Figure 6: A basic pinhole rendering shader. The parameter `slope` controls which plane is in focus, and the `shift_x` and `shift_y` parameters control the point of view.

3 GPU-based Plenoptic Depth Mapping

3.1 Rapid development with GLSL

We have implemented a plenoptic rendering program using Python and OpenGL (with the pyopengl bindings [6]). The program generates depth maps on the graphics card (GPU) and uses these to enable different rendering techniques, such as those demonstrated in Section 4. We found this approach to be very advantageous. The conventional wisdom is to do a pure CPU implementation first, and then later optimize by running on the GPU. Our experience has been that GLSL (the programming language for OpenGL GPU programs) is well-suited for graphical tasks and thus it was not significantly more difficult to implement rendering algorithms with the GPU. Furthermore, Python simplifies many of the details such as loading images and setting up an OpenGL environment. The payoff is immense; the GPU versions run in seconds at worst, while a CPU implementation could easily take minutes. Although we did not achieve interactive performance, we did not spend much time waiting on rendering, and thus we could quickly try slight variations in our rendering algorithms.

Rendering is implemented as a GLSL fragment shader, which is a small program that runs on the GPU. Shaders are by nature output driven; the main purpose of a fragment shader is to determine what color a given output pixel should be. Plenoptic rendering works by determining the microimage that best fits the current pixel, and then selecting a patch from that and adjacent images to blend together. A simple pinhole shader is given in Figure 6, where `slope` controls the focus. Theoretically, in a pinhole camera, everything should be in focus, so this parameter actually controls which plane has no artifacts.

3.2 Generating depth maps

There are two obvious ways to generate a depth map for a plenoptic image. The first is to calculate depth for each lightfield pixel, while the second is to calculate the depth of each rendered pixel. We have chosen the second approach. The lightfield approach has the advantage that the depth map can be reused even when rendering a new point of view, but is disadvantageous when rendering all-in-focus because each rendered pixel will have many candidate slopes.

The shader in Figure 6 can be modified to generate a depth map. The idea is to find similar regions in adjacent microimages, and determine the disparity that results in the best overall matches. For a given pixel, we sample an 6×6 pixel patch around it, and then compare this with patches at various disparities in several adjacent microimages. We compare patches using the Euclidean distance in RGB

Figure 7: An example depth map.

Figure 8: An image rendered with an infinite depth of field, made possible using a depth map.

color-space, but more sophisticated measures, such as cross-correlation, could be used as well. After considering patches at various slopes, the algorithm picks the best score (lowest color distance) and reports this as the correct slope for the given pixel. An example depth map is given in Figure 7. The GLSL shader program used to calculate this depth map is given in Appendix A.

To compare patches, we consider the eight neighboring microimages, rather than a single pair. This gives us more points to fit to one slope, which should smooth out much of the noise. The fact that our algorithms must simultaneously match disparities in several directions suggests that we should outperform a traditional stereo approach in some cases.

4 Rendering Experiments

As mentioned earlier, the use of Python and OpenGL enabled rapid experimentation with rendering techniques. In this section, we discuss three effects that are enabled by depth maps.

4.1 All-in-focus

Once we know the depth of each output pixel, rendering with an infinite depth of field (which is not possible using a traditional camera) is relatively straightforward. Rather than passing the slope in as a parameter, we simply have the shader read the desired slope out of a depth map. An example is given in Figure 8. Note that in this image, we have performed some manual filtering on the depth map to give smoother results (both a median and Gaussian filter were used).

In our experience, all-in-focus rendering is fairly tolerant of inaccurate depth maps. The reason is that our depth maps tend to be the most inaccurate in regions of low texture, since these regions do not have much variation in match quality. Fortunately, artifacts introduced by rendering at the wrong slope are significantly less noticeable in low texture regions.

4.2 Green Screen Insertion

Many visual effects in the movie industry rely on matting videos together. This is normally done by filming one scene in front of a green screen, and then

Figure 9: An example of depth-based green screen insertion. This could easily be extended to do depth-based matting.

Figure 10: An example of adding fog to a scene using depth information. Notice that the more distant regions are darker.

using computer software to replace the green background with another image or video. Movie makers and actors alike find green screens cumbersome to use, and could benefit from the ability to do matting without a green screen. As a proof of concept, demonstrate the *insertion* of a green screen, by simply drawing all regions beyond a certain depth in green. Figure 9 shows an example of this, which also uses a manually filtered depth map.

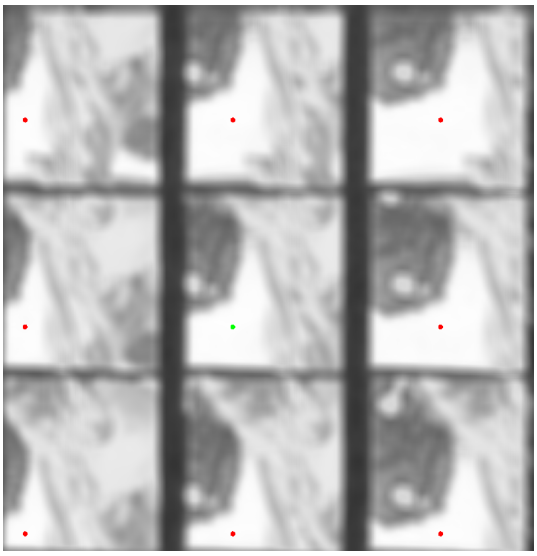
4.3 Atmospheric Effects

While matting can be accomplished without depth information, other effects are impossible without depth. Many atmospheric effects fall into this category. For example, in computer graphics, fog is typically done by blending the fog color and the image color with a weight determined by the distance from the viewer. Generating depth maps allows us to insert fog in photographs as well, which we demonstrate in Figure 10.

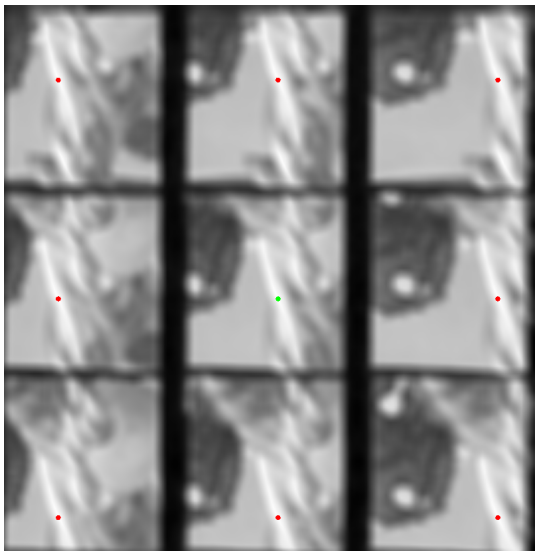
5 Uncertainty Estimation

Our simple depth map estimation algorithm does well in highly-textured regions, where false matches between patches are unlikely. In regions with little texture, however, it becomes much more difficult to find the correct slope for a given pixel in the rendered image. For these low texture regions, we found that our rendering experiments came out much better when we filtered the generated depth maps by hand. Hand-tuning each depth map is not a scalable solution, however, especially for multiple viewpoints of a single image or plenoptic video.

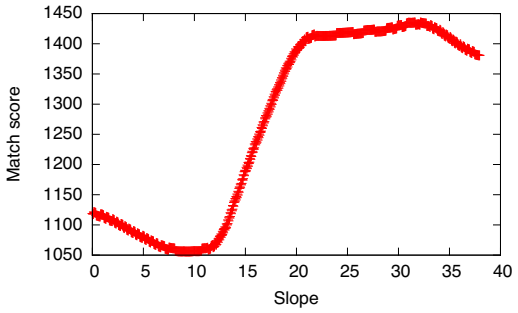
When estimating our depth maps, we determined the "correct" slope for each pixel in the rendered image by choosing the slope with the best score. Slopes for each pixel were scored by computing the summed color distances between adjacent patches along those slopes. The slope with the lowest score (smallest total color distance) was assumed to be correct. In order to estimate the uncertainty of that assumption, we must compute some function on the slope score *distribution* for each rendered pixel.



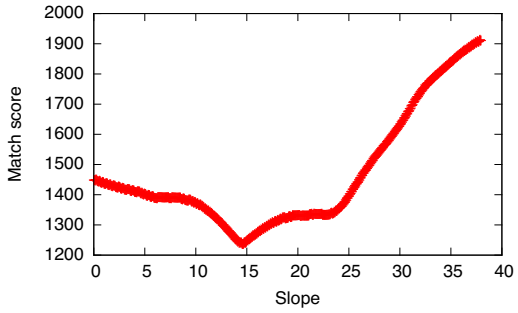
(a) Slope estimation in a low textured region.



(b) Slope estimation in a high textured region.



(c) Slope distribution for a low textured region (lower is better). There is no clear best slope.



(d) Slope distribution for a high textured region (lower is better). There is a clear minimum score at about slope 14.

Figure 11: Example score distributions in low and high textured regions.

Figures 11(a) and 11(b) demonstrate how score confidence varies in low and high texture regions. In both cases, we are trying to determine the correct slope for the green point by matching a patch around the green point with the patches around each of the red points. The amount by which the red points spread out across adjacent microimages is controlled by the slope. The shading represents how strong of a match each point is with the green point; white represents a very strong match, while black represents a poor match. In Figure 11(a), the point under consideration is in the background (see Figure 2), and thus any of the background points will match about as well. This is apparent from the large white regions in the image, and also the wide valley in Figure 11(c). On the other hand, in Figure 11(b) we consider a point in a high texture region. Here the regions of strong matches are much smaller, and there is only one slope that produces a strong match in each of the adjacent microimages. In Figure 11(d), the strong match shows up as a sharp peak around the minimum score.

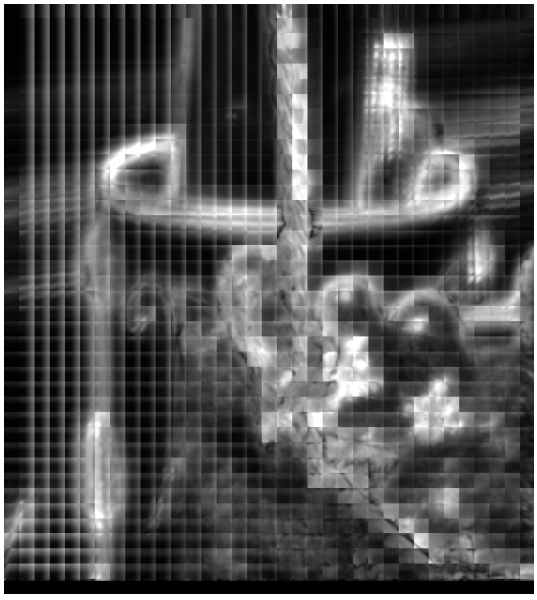


Figure 12: Uncertainty in an estimated depth map. Low-textured regions have higher uncertainty (black).

In Figure 12, we estimate the uncertainty in an estimated depth map using the distance between the slope distribution's minimum and average score (black means a smaller distance and thus more uncertainty). Other approaches to estimating uncertainty are possible as well. For example, we could directly determine low texture areas by looking at the derivative of the image, or by computing some other measure of the "peakness" of the score distribution. Figures 11(c) and 11(d) suggest that another good metric would be to find areas where the match score changes dramatically with small changes in slope. A thorough evaluation is needed to determine the best method of determining uncertainty.

6 Future Work

The depth maps generated with our simple algorithm worked surprisingly well for our rendering experiments, but a more sophisticated algorithm will be required for "real world" use. Specifically, the amount of noise present in our depth maps is too high for scenarios where a human cannot perform fine-tuning by hand. We propose to use a Bayesian filtering process that will take into account our uncertainty estimation in the depth map and edge information about the image itself.

Bayesian filtering has been used by researchers in the past to do image restoration [3]. Edges in the image are detected using standard means, and this information is used to inform the filtering process by restricting smoothing across edges. The process is Markov-based, and can be viewed as an energy-minimization problem.

For our project, we plan to use both depth uncertainty and edge information to restrict the flow of depth values while smoothing the depth map. This will be beneficial in portions of the image where regions with low and high texture are connected and at the same depth. Regions with low texture will have a high depth uncertainty due the multitude of possible slope matches (see Section 3), whereas high texture regions have likely been assigned the correct slope with low uncertainty. We would like the highly certain depth estimates to flow from certain to uncertain regions, with edges inhibiting the flow.

7 Related Work

GPUs have already been used for high performance rendering of plenoptic 2.0 images [2]. This work was primarily about rendering different focal depths and stereo 3D rendering. Our work draws inspiration from this, yet focuses more on depth map generation and the rendering effects that are enabled by depth maps.

Other work, such as [4], has considered uncertainty estimation as a means to improve depth maps. This work demonstrates that confidence information greatly improves depth maps and converges quicker on a high quality depth map.

Early on, [1] demonstrated the recovery of depth information from plenoptic images. The authors report that considering several microimages produces better results than simple stereo algorithms. They additionally perform confidence estimation using the image's spatial derivative.

8 Conclusion

Depth map estimation using a plenoptic lightfield is an interesting extension of the standard stereo depth map problem. In the plenoptic lightfield, we have multiple images of each point in the scene from different directions. With this extra information available, it is possible to estimate a depth map without as many heuristics to guess at missing information.

In this paper, we demonstrated that a simple algorithm can be used to generate surprisingly good depth maps. We expect future improvements can incorporate more of the 4D nature of plenoptic images to generate even more accurate depth maps. Our GPU-based implementation allows for rapid experimentation with a variety of rendering algorithms, such as all-in-focus and green-screen insertion. We proposed including edge and uncertainty information in a Bayesian filtering process to further refine the generated depth maps.

References

[1] E.H. Adelson and J.Y.A. Wang. Single lens stereo with a plenoptic camera. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):99–106, 1992.

[2] G. Chunev, A. Lumsdaine, and T. Georgiev. Plenoptic rendering with interactive performance using gpus. *SPIE Electronic Imaging*, January 2011.

[3] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 6:721–741, 1984.

[4] J. Jachalsky, M. Schlosser, and D. Gandolph. Confidence evaluation for robust, fast-converging disparity map refinement. In *2010 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1399–1404. IEEE, 2010.

[5] A. Lumsdaine and T. Georgiev. The focused plenoptic camera. In *2009 IEEE International Conference on Computational Photography (ICCP)*, pages 1–8. IEEE, 2009.

[6] PyOpenGL – The Python OpenGL Binding. `http://pyopengl.sourceforge.net`.

A Depth Map Shader

```
// -*- c -*-
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect texture;
uniform float micro_size;
uniform vec2  texture_size;
uniform float shift_x;
uniform float shift_y;
uniform float patch_size;
uniform int patch_width;

void main() {
  vec2 num_micro_images = texture_size / micro_size;

  vec2 p = floor(gl_TexCoord[0].st / micro_size);
  vec2 shift = vec2(shift_x, shift_y);
  vec2 offset = gl_TexCoord[0].st / micro_size - p;

  int num_patches = 3;
  int num_images = 1;

  float best_slope = 0.0;
  float best_match = 10000000000.0;

  for(float patch_size = micro_size / 2.0; patch_size >= 0.0; patch_size -= 0.5) {
      vec2 left_base = p * micro_size + shift - offset * patch_size;

      float score = 0.0;
      for (int i = 0; i < num_patches; i++) {
          for(int j = 0; j < num_patches; j++) {
              vec2 pixel_shift = vec2(i, j);
              vec4 left = texture2DRect(texture,
                                        left_base
                                        + pixel_shift);

              for(int m = -num_images; m <= num_images; m++) {
                  for(int n = -num_images; n <= num_images; n++) {
                      if(m == 0 && n == 0) continue;

                      vec2 right_base =
                          left_base + vec2(m, n) * (micro_size + patch_size);

                      vec4 right = texture2DRect(texture,
                                                 right_base
                                                 + pixel_shift);

                      score += distance(left, right);
                  }
              }
          }
```

```
        }
        if(score < best_match) {
            best_slope = patch_size;
            best_match = score;
        }
    }

    float color = best_slope / (micro_size / 2.0);
    gl_FragColor = vec4(color, color, color, 1.0);
}
```