

# Plenoptic Rendering With Interactive Performance Using GPUs

Andrew Lumsdaine<sup>a</sup>, Georgi Chunev<sup>a</sup>, and Todor Georgiev<sup>b</sup>

<sup>a</sup>Indiana University, Bloomington, IN, USA;

<sup>b</sup>Qualcomm, San Diego, CA, USA

## ABSTRACT

Processing and rendering of plenoptic camera data requires significant computational power and memory bandwidth. At the same time, real-time rendering performance is highly desirable so that users can interactively explore the infinite variety of images that can be rendered from a single plenoptic image. In this paper we describe a GPU-based approach for lightfield processing and rendering, with which we are able to achieve interactive performance for focused plenoptic rendering tasks such as refocusing and novel-view generation. We present a progression of rendering approaches for focused plenoptic camera data and analyze their performance on popular GPU-based systems. Our analyses are validated with experimental results on commercially available GPU hardware. Even for complicated rendering algorithms, we are able to render 39Mpixel plenoptic data to 2Mpixel images with frame rates in excess of 500 frames per second.

**Keywords:** Focused plenoptic camera, real-time rendering, GPU, GLSL

## 1. INTRODUCTION

Integral photography has more than 100 years of history, starting with Ives<sup>1</sup> and Lippmann.<sup>2</sup> Lippmann motivated his work in this area by observing that even the “most perfect photographic print only shows one aspect of reality; it reduces to a single image fixed on a plane, similar to a drawing or a hand-drawn painting.” With integral photography he attempted instead to render infinitely more—“the full variety offered by the direct observation of objects.” Unfortunately, because of inherent limitations of technologies available at the time, the potential of integral photography was not realized.

Integral photography re-emerged in the 1990s with the introduction of the plenoptic camera, originally presented as a technique for capturing 3D imagery and for solving computer vision problems.<sup>3</sup> It was designed as a device for capturing the distribution of light rays in space, i.e., the 4D plenoptic function. The lightfield<sup>4</sup> and lumigraph,<sup>5</sup> introduced to the computer graphics community, established a theoretical framework for analyzing the plenoptic function. A hand-held plenoptic camera, along with new methods of processing, was introduced by Ng.<sup>6</sup>

Making integral photography practical required two final steps. First, the images rendered from a plenoptic camera needed to be at a resolution acceptable to modern photographers. The traditional plenoptic camera rendered 2D images that were quite small (e.g., 90k pixels<sup>6</sup>). Recent work on the focused plenoptic camera greatly increased the available spatial resolution of plenoptic cameras to a level comparable to that of non-plenoptic digital cameras.<sup>7</sup>

The final step, which we present in this paper, is to enable processing of lightfield data at interactive frame rates. Because of the infinite variety of images that can be produced from a captured lightfield, interactivity is an essential feature for the end user. We achieve this level of performance by taking advantage of the power and programmability of modern Graphics Processing Units (GPUs). The massively-parallel high-performance architectures of modern GPU platforms are well matched to lightfield processing and rendering. In addition, modern GPUs can be programmed in a number of ways, making this computational power directly available to end users.

In this paper we describe our GPU-based approach for lightfield processing and rendering, with which we are able to achieve real-time performance for well-known lightfield tasks such as novel-view generation, refocusing, and stereo rendering. The plan of the paper is as follows. Section 2 contains an overview of related work. In Section 3 we provide background information about lightfield processing and rendering. Our approach to implementation of lightfield processing rendering tasks using the OpenGL Shading Language is given in Section 4. The results of performance experiments are given in Section 5 and we conclude in Section 6.

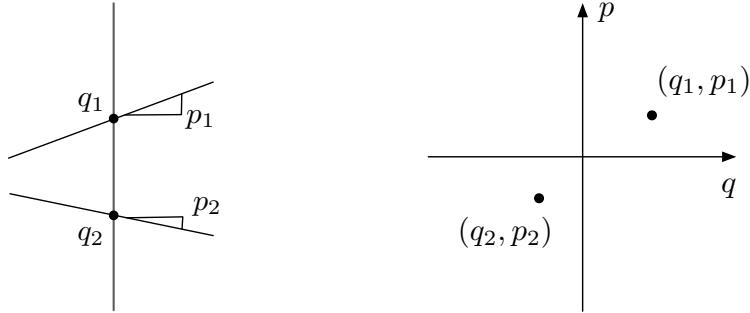


Figure 1: Rays are represented with coordinates  $q$  and  $p$ . The space of all rays comprises the  $q$ - $p$  phase space.

## 2. RELATED WORK

Interactivity has long been recognized as an important feature for lightfield rendering (e.g., see papers by Antunez et al.,<sup>8</sup> Isaksen et al.,<sup>9</sup> and Ng et al.<sup>6</sup>). At the same time, the computational requirements of lightfield rendering were an impediment to interactivity, particularly as lightfield imagery grew into the gigapixel range.

Particularly as CPU performance has plateaued, there is interest in using GPUs in almost all areas of computing where performance is a concern. Computer graphics and computer vision are no exception.<sup>10</sup> For example, Todt et al. apply GPUs to the problem of accelerating image-based rendering in computer graphics.<sup>11</sup> In an unpublished technical report, Meng et al. tested the performance of several plenoptic 1.0 rendering algorithms on various hardware: single core processors, multiprocessors, and a GeForce 7 series GPU.<sup>12</sup> They identified the underutilization of the GPU's resources, due to the fixed pipeline, as its drawback, an issue that has been removed entirely since the introduction of unified shader architecture GPUs by Nvidia.<sup>13</sup>

Luke et al. focus on developing practical technological solution for 3D TV.<sup>14</sup> They suggest a part of the solution will involve plenoptic (1.0) video cameras, which have the same drawbacks as plenoptic cameras, namely low-resolution images due to spatio-angular tradeoffs. A GPU implementation is described for the super-resolution and distance extraction algorithm presented by Owens et al.<sup>10</sup>

## 3. BACKGROUND

### 3.1 Radiance

The plenoptic function<sup>3</sup> (also called the *lightfield*<sup>4</sup> or *radiance*<sup>15</sup>) is a density function describing the light rays in a scene. Since the radiance is a density function over the ray space, we describe radiance transformations via transformations applied to elements of the underlying ray space. Rays in three dimensional space are represented as four dimensional vectors: Two coordinates are required to describe position and two are required to describe direction. Following physics-based conventions,<sup>16</sup> we denote the radiance at a given plane perpendicular to the optical axis as  $r(q, p)$ , where  $q$  describes the location of a ray in the plane and  $p$  describes its direction. (These coordinates are also used in optics texts such as those by Gerrard<sup>17</sup> and Wolf.<sup>18</sup>) For illustrative purposes, and without loss of generality, we adopt the convention of a two-dimensional  $q$ - $p$  plane in this paper.

A two-dimensional image is created from the four-dimensional radiance by integration over all rays incident at a given  $q$ . Given a radiance  $r(q, p)$  at the image plane of a camera, an image  $I(q)$  is rendered for a given range of the available  $p$  values according to

$$I(q) = \int_p r(q, p) dp. \quad (1)$$

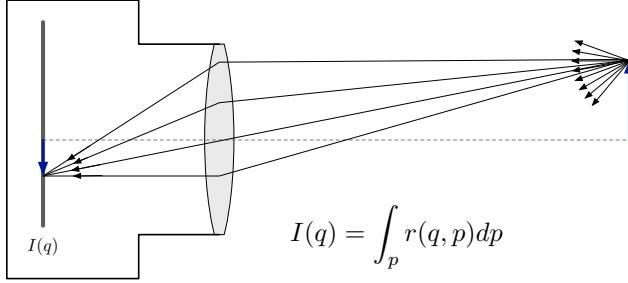
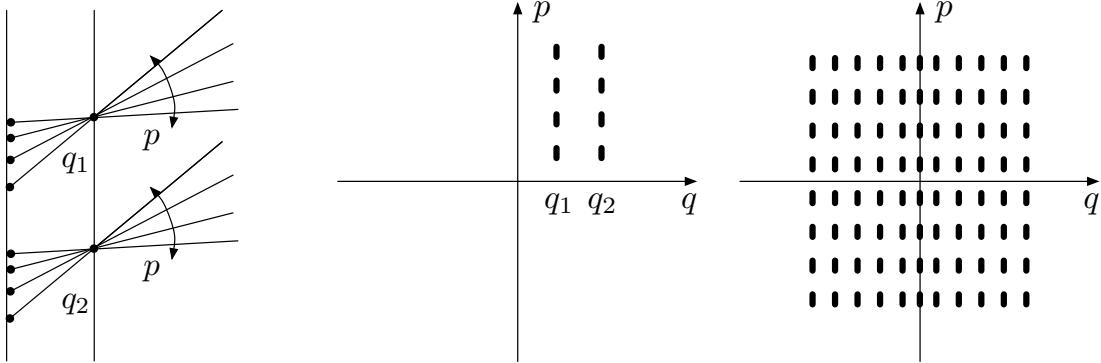


Figure 2: The sensor pixels in a traditional camera capture an image by physically integrating the intensities of all of the rays impinging on them.



(a) Rays converging at a pinhole will separate from each other as they travel behind it and can be captured individually by a sensor behind the pinhole.

(b) Representation of Figure 3a in phase space. Individual pinholes sample one position in the  $q$ - $p$  plane, while individual pixels sample different directions.

(c) A single image captured behind the pinhole thus samples a vertical stripe in the  $q$ - $p$  plane, while an array of pinholes samples a grid.

Figure 3: Radiance sampling with pinholes.

### 3.2 The Lippmann Sensor

As shown in Figure 2, a traditional camera captures an image by physically integrating the intensities of all of the rays impinging on each sensor pixel. A plenoptic camera, on the other hand, captures each ray separately. One approach to separating, and then individually capturing, the rays in a scene is to put a pinhole where the sensor pixel would be while placing the sensor itself some distance  $b$  behind the pinhole. In this case, the rays that converge at the pinhole will diverge as they propagate behind the pinhole. The separate pixels in the sensor now capture separate rays. I.e., the intensity as a function of position at the sensor represents the radiance as a function of direction at the position  $q$  of the pinhole. A single image captured behind the pinhole thus samples a vertical stripe in the  $q$ - $p$  plane, while an array of pinholes samples a grid. This process is illustrated in Figure 3

Although the ideal pinhole makes an ideal “ray separator,” in practice microlenses are used instead to gather sufficient light and to avoid diffraction effects. Figure 4 shows a diagram of such a sensor, which we will refer to as the “Lippmann sensor” (after Gabriel Lippmann who first proposed it<sup>2</sup>). In the diagram,  $b$  is the distance from the sensor to the microlens plane and  $a$  is the distance from the microlens plane to the main lens image plane. The microlens focal length is  $f$ ;  $a$ ,  $b$ , and  $f$  are assumed to satisfy the lens equation  $1/a + 1/b = 1/f$ . Sensor pixels have size  $\delta$  and without loss of generality we take  $d$  to be the microlens aperture and the spacing between microlenses.

A specific instance of the Lippmann sensor was proposed by Ng,<sup>6</sup> in which the distance  $b$  was taken to be equal to the microlens focal length  $f$ . In this case, as has been derived elsewhere,<sup>6</sup> we have the following expression for how the image

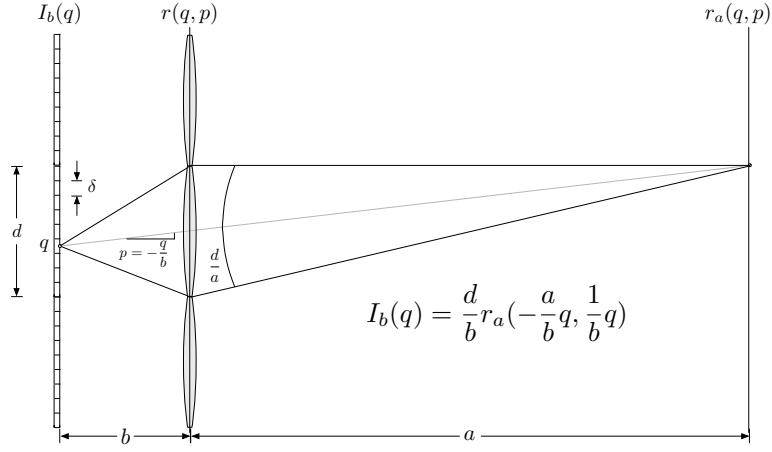
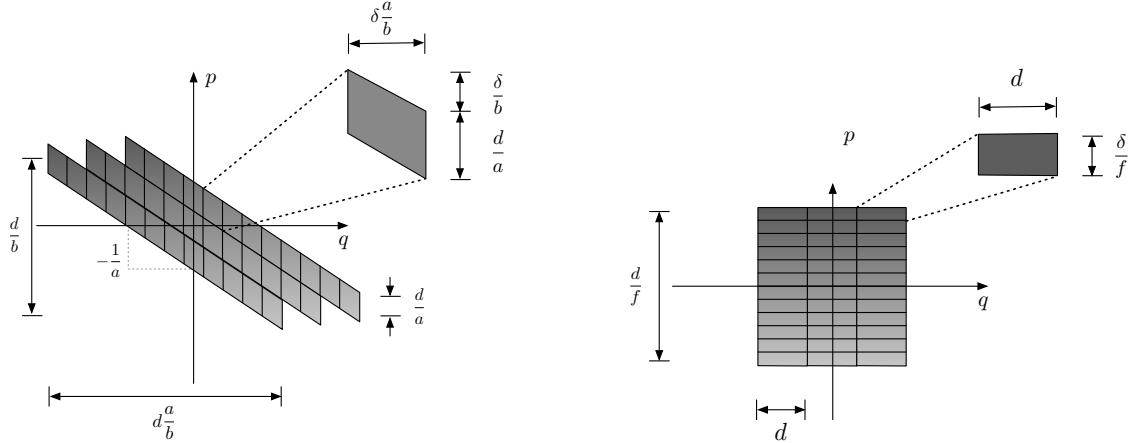


Figure 4: Geometry of Lippmann sensor for a plenoptic camera. The CCD (or CMOS) sensor is placed at a distance  $b$  behind an array of microlenses. In one notable limit,  $b \rightarrow f$  and  $a \rightarrow \infty$ .



(a) Sampling by the Lippmann sensor at a finite distance in front of the microlenses, where  $a$ ,  $b$ , and  $f$  satisfy the lens equation. Note that shearing changes the width of the spatial region sampled by a single pixel.

(b) Sampling by the Lippmann sensor at the microlens array when the distance between the microlens array and the sensor is equal to the microlens focal length (i.e., when  $b = f$ ).

Figure 5: Radiance sampling by the Lippmann sensor in different plenoptic cameras. The geometry of a single pixel for each case is shown in the upper right.

captured on the sensor ( $I_f$ ) samples the radiance at the microlens plane ( $r$ ):

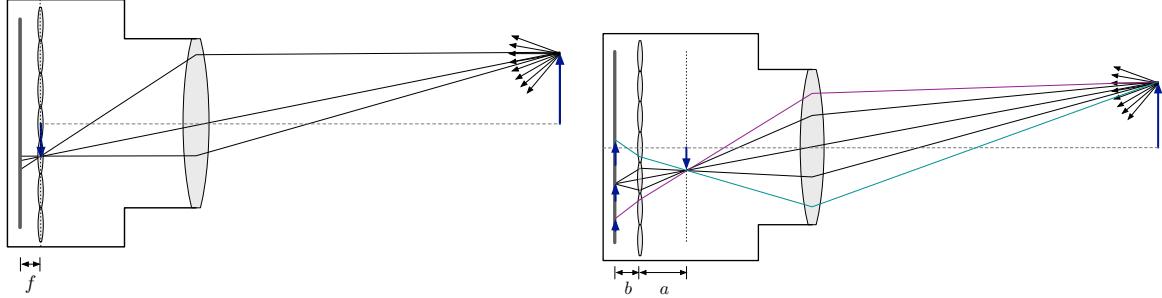
$$I_f(q) = \frac{d}{f} r(0, \frac{1}{f} q). \quad (2)$$

This sampling is shown graphically in Figure 5b.

A different realization of the Lippmann sensor was proposed by Lumsdaine and Georgiev,<sup>7</sup> in which the distance  $b$  was chosen not to be equal to  $f$  in order to form a relay system with the camera main lens. In this case, as derived by Lumsdaine and Georgiev,<sup>7</sup> we have the following expression for how the image captured on the sensor ( $I_b$ ) samples the radiance at the microlens focal plane ( $r_a$ ):

$$I_b(q) = \frac{d}{b} r_a(-\frac{a}{b} q, \frac{1}{b} q) \quad (3)$$

This sampling is shown graphically in Figure 5a.



(a) The traditional plenoptic camera focuses the main lens on the microlens array of the traditional Lippmann sensor.

(b) The focused Lippmann sensor in the focused plenoptic camera acts as a relay system with the main lens.

Figure 6: A comparison of the traditional and focused plenoptic cameras.

As reported by Lumsdaine and Georgiev,<sup>7</sup> each microimage captured by the focused Lippmann sensor represents a slanted stripe of the radiance at distance  $a$  in front of the microlens array, allowing significantly higher spatial resolution when rendering.<sup>7</sup>

### 3.3 Plenoptic Cameras

A plenoptic camera can be constructed by substituting a Lippmann sensor for the traditional sensor in the camera body. The principle design issue then becomes the main lens optics. The main lens of the camera will map the light in the exterior world into the light in the interior world of the camera. This mapping will present a certain plenoptic function inside the camera. In the case of Ng's hand-held plenoptic camera, the main lens focal plane (the primary plane of photographic interest) was mapped to the plane of the microlens array. In the case of the focused plenoptic camera, the main lens focal plane was mapped to a distance  $a$  in front of the microlens array. A comparison of the traditional plenoptic camera with the focused plenoptic camera is shown in Figure 6.

### 3.4 Rendering

Equation (1) defines rendering for the continuous case. In the discrete case, rendering becomes a summation:

$$I[k] = \sum_j r[k, j] \quad (4)$$

where the square brackets indicate indexing into discretized quantities. We note however, that in general, a plenoptic camera does not provide us directly with an array containing  $r[k, j]$ . For example, the focused plenoptic camera captures an image representing the radiance according to Equation (3). In that case we cannot simply sum up along a single array axis to obtain an output pixel.

One approach to rendering would be to suitably interpolate the captured plenoptic function to a grid so that Equation (4) could be applied—a process that is both memory and compute intensive—and one which can introduce noise and other unwanted artifacts. An alternate approach, and the one we use in this paper, is to *render directly from the captured plenoptic function*. To accomplish this, we substitute the discrete form of Equation (3) into Equation (4) to obtain the *discrete plenoptic rendering formula*:

$$I[k] = \sum_j I_b[k - \frac{a}{b}j, j]. \quad (5)$$

In other words, rendering is accomplished by summing up pixels along a slanted line corresponding to the minification factor, as illustrated in Figure 7. We note that due to linearity, shearing transformations of the radiance will correspond to different slants for summation in the captured plenoptic function, and phase space volume of each pixel is conserved.

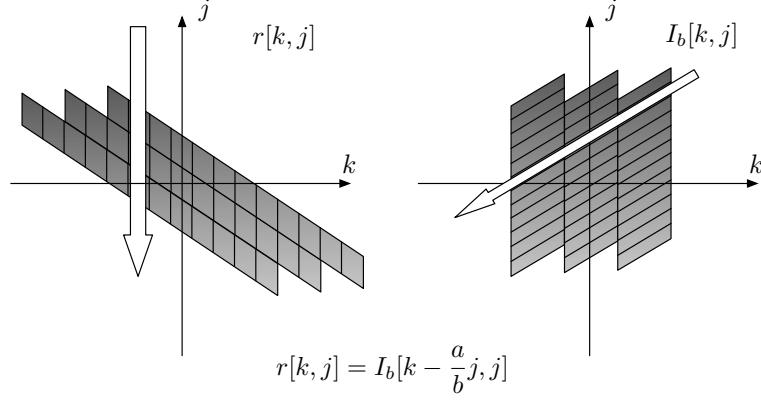


Figure 7: Summation of the radiance in the vertical (angular) direction at the main lens image is equivalent to summation at a slant of the radiance at the microlenses.

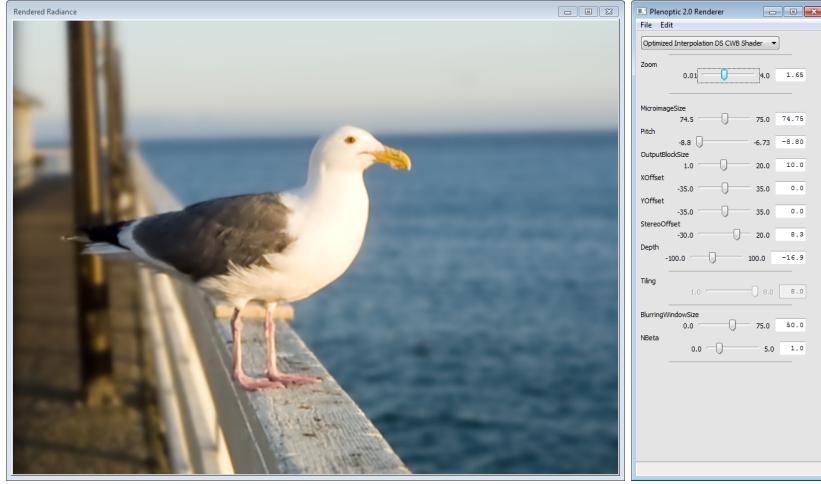


Figure 8: Screenshot of our OpenGL plenoptic rendering application. The sliders in the right-hand panel are used to interactively change the values of uniform variables shared between the control program and the shaders.

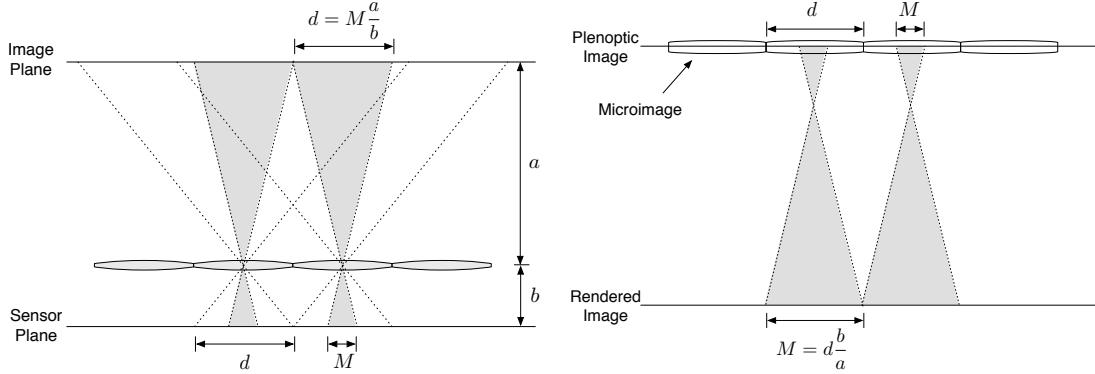
## 4. GPU IMPLEMENTATION OF RENDERING

### 4.1 Programming the GPU

A number of programming languages and tools have emerged for GPU programming: the OpenGL Shading Language (GLSL),<sup>19</sup> Cg,<sup>20</sup> CUDA,<sup>21</sup> and OpenCL.<sup>22</sup> Whereas GLSL and Cg are aimed at rendering and related tasks in mind, CUDA and OpenCL are aimed directly at general purpose programming tasks. Because lightfield rendering is similar to traditional rendering, and because a key part of interactively working with lightfields is to display them, GLSL is well-suited to our needs.

Our GLSL implementations of lightfield rendering are carried out completely in OpenGL fragment shaders. Although our examples were implemented in Python (via the PyOpenGL library<sup>23</sup>), interfaces to OpenGL are similar across languages. Other than providing support for creating and installing the shader, the main functionality provided by OpenGL were the following:

1. Read in the radiance data (from a stored image),
2. Serialize the radiance data to a format suitable for OpenGL,
3. Create a 2D OpenGL texture object for the data,



(a) Image capture geometry. For optimal sensor coverage, the diameter of a microimage circle  $d$  needs to be equal to the distance between microlenses, the microlens pitch.

(b) Rendering a single view from the radeance captured by the focused plenoptic camera is the inverse of capture. A single view spans multiple pixels in each captured microimage.

Figure 9: Image capture and single view rendering for the focused plenoptic camera.

4. Create an OpenGL scene geometry of a single quad to which to render the texture, and
5. Update shared (“uniform”) variables used by the shader.

Rendering the plenoptic image data is then accomplished by rendering the installed texture, using our custom shader. The OpenGL support code is omitted here for space reasons. A screenshot of our OpenGL plenoptic rendering application is shown in Figure 8.

## 4.2 Basic Focused Plenoptic Rendering

Our basic focused plenoptic rendering algorithm renders only one angular sample per pixel, depending on a user specified magnification factor  $d/M$  (see Figure 9), which shows how the focused plenoptic camera captures the image plane on its sensor. Here  $d$  is the size of each microlens image i.e., the diameter of the microimage circle.  $M$  is the size of a square patch that we pick from it. In other words, we use only a square patch of  $M^2$  pixels from each microimage, as opposed to using all pixels in the microimage circle of diameter  $d$ .

We remark that one way of interpreting this rendering process is that we take regions of size  $M$  from each microlens image in the flat and then magnify (by  $a/b$ , or equivalently  $d/M$ ) and flip them, then piece them together to make an output image as if captured directly at the main lens focal plane. This process, illustrated in Figure 9b, is the inverse of the optical process in the microlenses during plenoptic image capture.

This process is interpreted in phase space in Figure 10, where  $M$  is selected to be such that there is no overlap of pixels with same  $q$  values from different microlenses. For example, all samples from one microlens are inside the grey box, i.e.  $M = 4$ . We can pick any other patch of  $M^2$  pixels inside the microlens, which would correspond to a different grey box moved higher in  $p$  and rendering a different view.

## 4.3 Rendering Directly from the Plenoptic Image

Although the plenoptic function is four dimensional, a physical plenoptic camera must necessarily capture this 4D function as a 2D array of 2D microimages. For rendering, one could first convert this flat (the flattened plenoptic function) into a 4D array. However, this conversion process may not be as straightforward as simple index remapping because the microimage size may not be an exact integer number of pixels or the microimage array axes may not be precisely aligned with the CCD pixel axes. In this case, the plenoptic data would need to be interpolated to the 4D array, a process that is computationally expensive, and that can introduce noise into the data. Moreover, as described below, we access the plenoptic data as a texture on the GPU card, a process universally supported in 2D, but not supported in 4D. However, the data are the same,

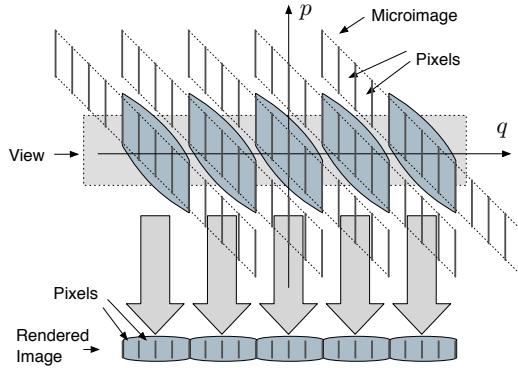


Figure 10: Phase space interpretation of rendering a single view from focused plenoptic data.

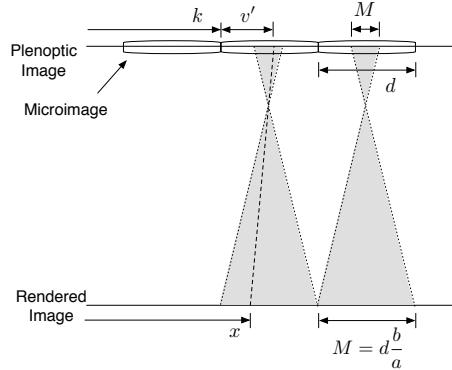


Figure 11: Basic rendering directly from the plenoptic image (without blending). Note the magnification  $a/b$ .

regardless of the dimensionality of representation. Therefore, as briefly discussed above (§ 3.4) and described more fully below, our approach using the GPU is to render directly from the flat.

Let  $d$  be the size of one microlens, measured in pixels. For a given point  $x$  in the output image, we need to find the corresponding sampling point on the flat. To accomplish this process, we need to perform two computations. First, given  $x$ , we need to determine from which microlens we will render  $x$ . Second, we need to determine where in the region of size  $M$  the point  $x$  lies.

To compute which microlens corresponds to  $x$ , we simply take the integer part of  $x$  divided by  $d$ . Let us denote the microlens number by  $k$ , given by

$$k = \lfloor \frac{x}{d} \rfloor. \quad (6)$$

The pixel location of the beginning of that microlens in the flat is then given by multiplying the microlens number by the size of one microlens image, i.e.,  $kd$ .

Next, we need to compute the offset within the region of size  $M$  corresponding to  $x$ . To do this, we compute the difference between  $x$  and the start of microlens ( $kd$ ). This will give the offset in the rendered image, but we need the offset in the flat. Since we are scaling a region of size  $M$  in the flat to a region of size  $d$  in the final image, the offset needs to be scaled by  $\frac{M}{d}$ , which is the flat image step size corresponding to a 1 pixel step in the rendered image. That is, the offset  $v$  is given by

$$v = \left( x - \lfloor \frac{x}{d} \rfloor d \right) \frac{M}{d} = \left( \frac{x}{d} - k \right) M. \quad (7)$$

Finally, we need to make one more adjustment. We want the center of the  $M \times M$  region of the flat to render to the center of the corresponding region of the final image. The formulas above will map the left edge of the microlens image to

```

uniform sampler2DRect flat;           // Plenoptic image input
uniform float M, d;                  // Shared variables passed by control program
uniform vec2 offset;                // d is microlens size, M is patch size

void main()
{
    vec2 x_d = gl_TexCoord[0].st/d;   // x/d (Note: gl_TexCoord[0].st is x)
    vec2 k   = floor(x_d);          // k = ⌊x/d⌋
    vec2 v   = (x_d - k) * M;        // (x/d - k)M
    vec2 vp  = v + 0.5*(d - M);     // v' = v + (d - M)/2

    vec2 fx = k * d + vp + offset;  // f(x) = kd + v'

    gl_FragColor = texture2DRect(flat, fx); // Lookup pixel value
}

```

Figure 12: GLSL fragment shader code for basic focused plenoptic rendering.

the left edge of the corresponding region in the rendered image. To accomplish this centering, we add an offset of  $\frac{d-M}{2}$  to  $v$ :

$$v' = v + \frac{d - M}{2} = \left(\frac{x}{d} - k\right)M + \frac{d - M}{2}. \quad (8)$$

Combining (6) and (8), the corresponding point in the flat for a given point  $x$  in the output image is given by  $f(x)$  where

$$f(x) = kd + v'. \quad (9)$$

The GLSL fragment shader code to carry out this algorithm is obtained in a straightforward fashion from this formula and is shown in Figure 12. The plenoptic image as well as the values for  $d$  and  $M$  are provided by the user program via **uniform** variables. The shader program computes  $v$ ,  $v'$ , and  $f(x)$  as  $v$ ,  $vp$ , and  $fx$ , respectively. Novel view generation is enabled by adding a user-specified *offset* vector (equal to  $(0, 0)$  by default) is added to the coordinates  $fx$ , resulting in a shift in the viewpoint of the rendered image. Finally, we look up the value of the pixel in the flat and assign that value to the requested fragment color.

#### 4.4 Rendering with Blending

To realize the integration over  $p$  in equation (1), we must average together (“blend”) the same spatial point across microlens images. For microlenses spaced  $d$  apart and a patch size of  $M$ , the pixels that need to be averaged together to a given pixel in the rendered image will be distance  $(d - M)$  apart. That is, we average all pixels at position  $f_i(x)$  where

$$f_i(x) = k_i + v'_i \quad (10)$$

where

$$k_i = \lfloor \frac{x}{d} \rfloor + i(d - M) \quad (11)$$

$$v'_i = v + \frac{d - M}{2} - iM \quad (12)$$

for  $i = \dots, -2, -1, 0, 1, 2, \dots$ . Since  $d$  is constant this means that for images generated with a given sampling pitch  $M$  there is a fixed upper bound for the number of microimages that can be sampled to contribute to a given  $x$ . This upper bound,  $lim$ , is given by

$$lim = (\frac{\lfloor \frac{d}{M} \rfloor - 1}{2}). \quad (13)$$

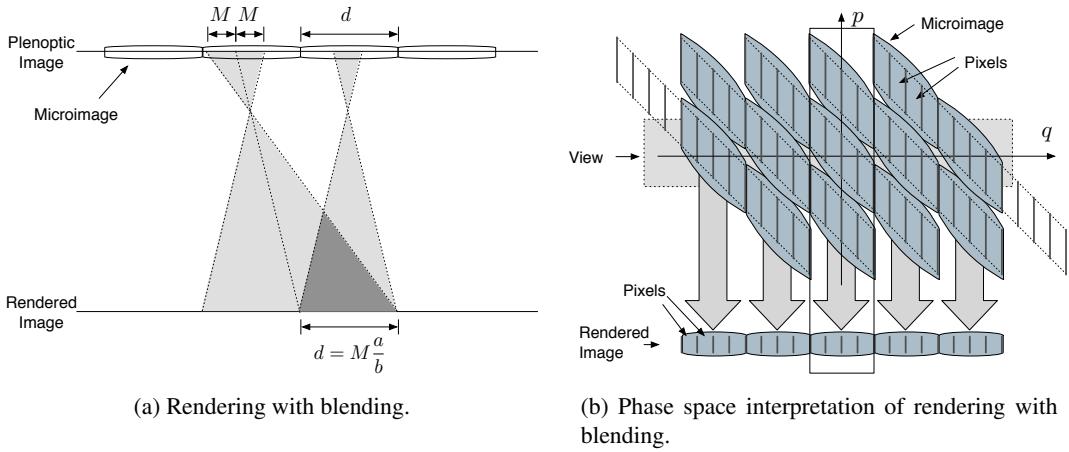


Figure 13: Optical and phase-space interpretations of rendering with blending.

```

uniform sampler2DRect flat, weightm; // Plenoptic image input and weight mask
uniform float M, d; // Shared variables passed by control program
uniform int lim; // d is microlens size, M is patch size
// lim is number of neighbors to blend together

void main()
{
    vec2 x_d = gl_TexCoord[0].st/d; // x/d (Note: gl_TexCoord[0].st is x)
    vec2 k = floor(x_d); // k = ⌊x/d⌋
    vec2 v = (x_d - k) * M; // (x/d - k)M
    vec2 vp = v + 0.5*(d - M); // v' = v + (d - M)/2

    vec4 colXY = vec4(0.0);
    float total_weight = 0.0;

    for (int i = -lim; i <= lim; ++i) { // For each microimage in
        for (int j = -lim; j <= lim; ++j) { // lim × lim neighborhood
            vec2 ij = vec2(float(i), float(j));
            vec2 dv = vp - ij * M;

            vec2 vPosXY = (k + ij)*d + dv; // Compute pixel location

            float weight = texture2DRect(weightm, dv); // Look up weight value
            colXY += texture2DRect(flat, vPosXY) * weight; // Look up pixel value and scale by weight
            total_weight += weight;
        }
    }
    gl_FragColor = colXY / total_weight ;
}

```

Figure 14: GLSL fragment shader code for focused plenoptic rendering with weight mask for blending.

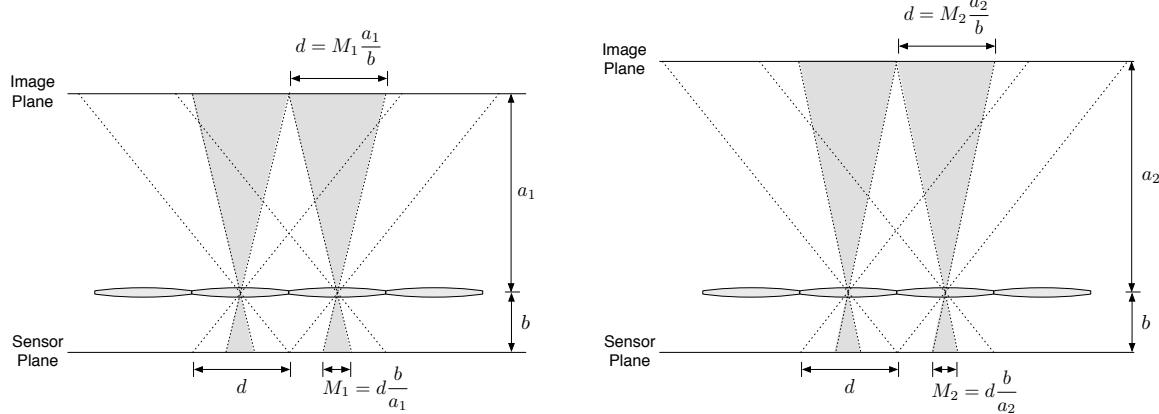


Figure 15: The focused plenoptic focusing principle. Image planes at different depths are rendered through appropriate choice of  $M$ .

In the focused plenoptic camera, it is likely that the pixels in the center of the microlens image are of higher quality than the pixels close to the edges. Moreover, since  $lim$  and  $M$  may be specified by the user, it is possible that we may sample out of bounds pixel values. Accordingly, we may want to weight the pixels differently in our summation process.

To accomplish weighted blending within GLSL, we use another texture (single-component in this case) as a lookup table to specify the weight of each pixel in the microlens as a function of position. If fragment coordinates fall outside the weight mask then the texture wrapping mode would determine what happens with the lookup. This situation occurs when the weight mask is smaller than one microlens image, or when the chosen  $lim$  value is larger than the one obtained from equation (13). We suggest the use of a  $d \times d$  Gaussian mask with GL\_CLAMP set as the wrapping mode. The GLSL implementation is shown in Figure 14.

#### 4.5 Focusing and Refocusing

An important characteristic of the focused plenoptic camera—the “focused plenoptic focusing principle”—can be seen in Figure 15. The focused plenoptic focusing principle states that within the depth of field of the microcameras in the Lippmann sensor, the image plane that is in focus is determined by the choice of  $M$ . To see this, consider the left and right illustrations in Figure 15, which show two different image planes focused at the sensor. We assume that  $d$  and  $b$  are the same in both cases and that the microlens aperture  $d$  is small enough that the microimages in both cases are in focus.

Since  $d$  is constant, the distance to the focal plane determines the value of  $M$  that needs to be used to properly render that plane. The focused plenoptic focusing principle directly follows as the converse: The choice of  $M$  used to render the output image determines the plane in the image that is in focus. Images captured with the focused plenoptic camera can therefore refocus rendered images by choosing different values of  $M$ . Thus, the GLSL shaders shown in Figures 14 and 12 are already refocusable. All that is required is to vary the value of the uniform variable  $M$ , something that can be done interactively via the control program associated with shader.

Typical (interesting) scenes are not at a uniform depth throughout the scene. Choosing a single value of  $M$  will render in focus those parts of the scene that correspond to depth  $a = b \frac{d}{M}$ . Other parts of the scene will not be rendered in focus. In the case of single-view rendering, out of focus portions of the scene will be rendered with visible artifacts due to misalignment of patch boundaries.<sup>7</sup> In the case of rendering with blending, if there are a sufficient number of views to blend, blending the misaligned patch boundaries results in the expected focus blur (see Figure 17). If there are not sufficient number of views to suppress the artifacts, other techniques such as depth-based rendering must be used.<sup>24</sup>

## 5. EXPERIMENTAL RESULTS

### 5.1 Experimental Setup

Rendering tests were performed on a DELL Precision T7400 with two Intel Xeon processors at 3.40GHz, 32.0GB RAM. The GPU was a Quadro FX 5600, with eight 16-core stream multi-processors clocked at 1.35GHz and connected to a

6x64bit 800MHz (effective 1600MHz) video memory bus. The operating system used was 64-bit Vista. We used OpenGL 3.0.0 drivers from Nvidia, release 186.18. The shader programs were run using a simple wrapper program written in Python (2.5).

Plenoptic image data was acquired with a medium format camera with an 80-mm lens and a 39-megapixel P45 digital back from Phase One. The lens is mounted on the camera with a 13-mm extension tube, which provides the needed spacing  $a$ . The microlens array is custom made by Leister Axetris. The microlenses have focal length of 1.5 mm so that they can be placed directly on the cover glass of the sensor. We are able to provide additional spacing of up to 0.5 mm to enable fine tuning of the microlens focus. The pitch of the microlenses is 500  $\mu\text{m}$  with precision 1  $\mu\text{m}$ . The sensor pixels are 6.8  $\mu\text{m}$ . The value of  $b \approx 1.6$  mm was estimated with precision 0.1 mm from known sensor parameters and independently from the microlens images at different F/numbers.

A crop from the  $7308 \times 5494$  plenoptic is shown in Figure 16. Notice the use of square main lens aperture (for efficiency), resulting in square microimages. Figure 17a shows a crop from image rendered using the basic rendering algorithm (and the basic rendering shader). Notice the blocky artifacts of areas that are not in focus, i.e. for which  $a/b$  is not right. Figure 17b shows a crop of the same region as Figure 17a. Notice the artifacts are now suppressed and that the portions of the image for which  $a/b$  is not right appear out-of-focus (blurred). The renderings are done interactively at 1600 and 500 frames per second, respectively.

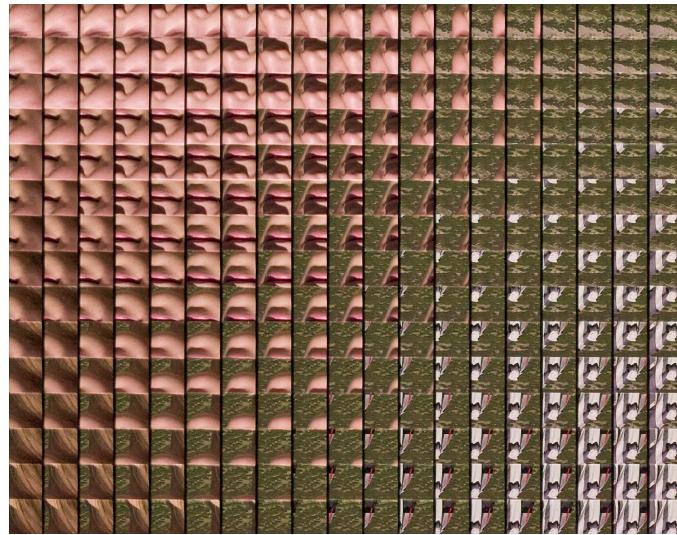


Figure 16: A small crop of about 5% from a 39 megapixel raw image (flat) captured with our Plenoptic 2.0 camera.

## 5.2 Performance

With our GPU based approach, we effect plenoptic rendering by rendering a texture to a single OpenGL quad. The size of that quad is specified by the user, i.e., it is the size of rendered image to be displayed to the screen (or saved to a file). Consequently, the amount of computation to render the final image is proportional to the size of the rendered image, not the size of the plenoptic data.

Modern GPU architectures are quite sophisticated and although our shaders are quite straightforward, modeling their computational performance would be quite complicated. However, a simple lower bound on performance can be estimated by assuming that rendering will be primarily memory-bound. The GPU card we used for our experiments, and Nvidia Quadro FX 5600, has eight 16-core stream multi-processors clocked at 1.35GHz and connected to a 6x64bit 800MHz(effective 1600MHz) video memory bus. With these specifications, the expected video memory bandwidth is  $6*(8 \text{ Bytes})*1600\text{MHz}$ , or 76.8GB/sec (71.5GiB/sec), which can deliver 19.2 billion IEEE 32FP color components to the stream processors. There is some additional benefit due to caching (which is not as significant a benefit in GPUs as it is for CPUs). We expect the gain from caching to be 1.35/0.8, which is the ratio between the GPU memory clock and the



(a) Image rendered using basic algorithm. Note the presence of artifacts at the mountains at infinity due to non-matching patches at that depth.

(b) Image rendered using weighted blending algorithm. Note that the artifacts at infinity are now suppressed and that the out-of-focus regions appear properly blurred.

Figure 17: Comparison of basic (a) and weighted blending (b) rendering.

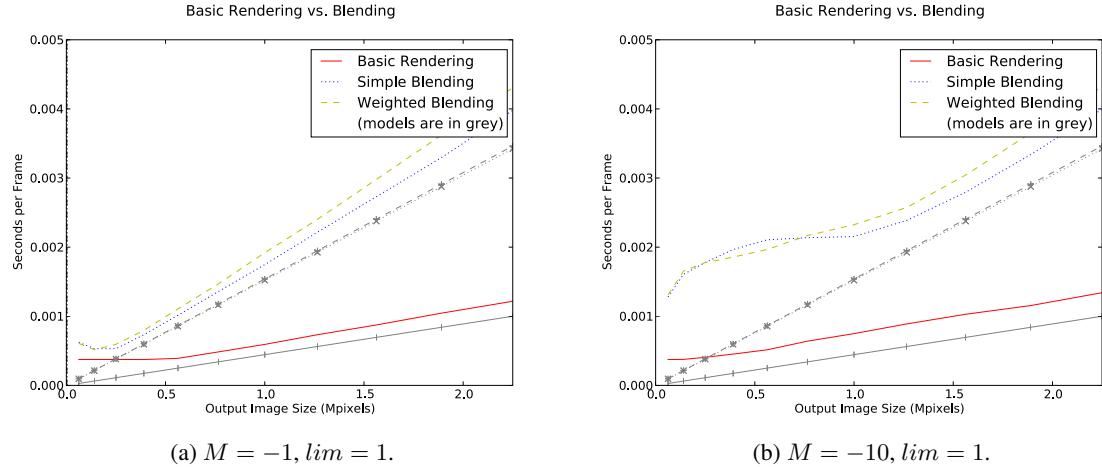


Figure 18: Performance comparisons of the shaders presented in this paper showing time to render an output image from plenoptic data as a function of output image size. Idealized performance is given in grey, and additionally marked with '—', 'x', and '+' for the basic, simple blending, and weighted blending shaders, respectively.

shader clock. In the case of uncached fetches, one 4-component (128bit) pixel requires  $\frac{16\text{Bytes}}{71.5\text{GiB/s}}$  seconds to complete. Accordingly, the time to render a single frame for a memory-bound shader can be expressed as

$$SPF = \frac{((nF - nCF) + nCF \times (0.8/1.35)) \times 16 \times OR}{71.5} \quad (14)$$

where,  $nF$  is the number of fetches per shader call;  $nCF$  is the number of cached fetches per shader call; and  $OR$  is the output resolution, measured in gigapixels. For the basic shader, we have  $nF = 1$  and assume  $nCF = 0$ . For the simple and weighted blending shaders we have  $nF = (2lim + 1)^2$  and assume  $nCF = 3lim^2 + 2lim$ . There are additional memory  $(2lim + 1)^2$  lookup costs for the weighted blending shader due to the weights, which would all be cached.

For our experimental performance measurements, we measured the time taken to render output images of varying sizes,

given the same plenoptic image source, and compare the results to our idealized estimates. The plenoptic image data was a  $7308 \times 5494$  color image, and the assigned weight mask was a  $75 \times 75$  grayscale squared Gaussian. Our performance results are shown Figure 18, which plot time per frame as a function of output image size. The graphs show results for basic, simple blending, and weighted blending shaders, with  $lim = 1$  and  $M = -1$  on the left graph  $M = -10$  on the right graph. Also plotted are the times predicted by our memory-bound model.

Our memory-bound model slightly overestimates the performance of the basic shader and of the blending shaders for  $lim = 1$ , because we have not taken into account latencies due to raster operations and because we neglect computational latencies, which contribute to the larger deviation from the model at smaller image sizes in the basic and weighted blending shaders.

## 6. CONCLUSION

In this paper we have presented an approach to full-resolution lightfield rendering that uses commercially available GPU hardware to render images at interactive rates. Although GPU hardware provides a significant amount of computing power, for rendering plenoptic image data we found that video memory bandwidth was also an important factor for achieving interactive performance. Using commercially available (and economical) hardware to interactively render plenoptic image data into final images with high resolution is another step towards making plenoptic cameras practical.

## REFERENCES

- [1] Ives, F., “Patent US 725,567,” (1903).
- [2] Lippmann, G., “Epreuves reversibles. Photographies integrales,” *Academie des sciences* , 446–451 (March 1908).
- [3] Adelson, T. and Bergen, J., “The plenoptic function and the elements of early vision,” in [*Computational models of visual processing* ], MIT Press (1991).
- [4] Levoy, M. and Hanrahan, P., “Light field rendering,” *Proceedings of the 23rd annual conference on Computer Graphics and Interactive Techniques* (Jan 1996).
- [5] Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F., “The lumigraph,” *ACM Trans. Graph.* , 43–54 (1996).
- [6] Ng, R., Levoy, M., Bredif, M., Duval, G., Horowitz, M., et al., “Light field photography with a hand-held plenoptic camera,” *Computer Science Technical Report CSTR* (Jan 2005).
- [7] Lumsdaine, A. and Georgiev, T., “The focused plenoptic camera,” in [*International Conference on Computational Photography* ], (April 2009).
- [8] Antunez, E., Barth, A., Adams, A., Horowitz, M., and Levoy, M., “High performance imaging using large camera arrays,” *International Conference on Computer Graphics and Interactive Techniques* (Jan 2005).
- [9] Isaksen, A., McMillan, L., and Gortler, S. J., “Dynamically reparameterized light fields,” *ACM Trans. Graph.* , 297–306 (2000).
- [10] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., and Purcell, T., “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum* **26**(1), 80–113 (2007).
- [11] Todt, S., Rezk-Salama, C., Kolb, A., and Kuhnert, K.-D., “GPU-based spherical light field rendering with per-fragment depth correction,” *Computer Graphics Forum* **27**, 2081–2095 (Dec 2008).
- [12] Meng, J., Weikle, D. A. B., Humphreys, G., and Skadron, K., “An approach on hardware design for computational photography applications based on light field refocusing algorithm,” Tech. Rep. CS-2007-15, University of Virginia (2007).
- [13] Nvidia, “NVIDIA GeForce 800 GPU architecture overview,” Tech. Rep. TB-02787-001 v1.0 45 (2006).
- [14] Luke, J. P., Nava, F. P., Hernandez, J. G. M., Ramos, J. M. R., , and Rosa, F., “Near real time estimation of depth super-resolved and all-in-focus images from a plenoptic camera using graphic processing units (GPU),” *International Journal of Digital Multimedia Broadcasting* (2009).
- [15] Nicodemus, F. E., ed., [*Self-study manual on optical radiation measurements* ], National Bureau of Standards (1978).
- [16] Guillemin, V. and Sternberg, S., “Symplectic techniques in physics,” (1985).
- [17] Gerrard, A. and Burch, J. M., [*Introduction to Matrix Methods in Optics* ], Dover Publications (1994).
- [18] Wolf, K. B., [*Geometric Optics on Phase Space* ], Springer (2004).
- [19] Rost, R. J., [*OpenGL(R) Shading Language (2nd Edition)* ], Addison-Wesley Professional (January 2006).

- [20] Mark, W. R., Steven, R., Kurt, G., Mark, A., and Kilgard, J., “Cg: A system for programming graphics hardware in a C-like language,” *ACM Transactions on Graphics* **22**, 896–907 (2003).
- [21] Nickolls, J., Buck, I., Garland, M., and Skadron, K., “Scalable parallel programming with cuda,” *Queue* **6**(2), 40–53 (2008).
- [22] Khronos Group, “OpenCL - The open standard for parallel programming of heterogeneous systems.” <http://www.khronos.org/opencl/>.
- [23] “PyOpenGL.” <http://pyopengl.sourceforge.net/>.
- [24] Georgiev, T. and Lumsdaine, A., “Reducing plenoptic camera artifacts,” *Comput. Graph. Forum* **29**(6), 1955–1968 (2010).