LIGHT FIELD RENDERING

TNM089, IMAGING TECHNOLOGY

Linus Mossberg
linmo400@student.liu.se

Thursday 29th October, 2020

Abstract

This project-report presents a method and implementation to synthesize novel views from a collection of images using light field rendering. Two light field parameterizations from previous research is presented, the light slab parameterization and the dynamic reparameterization. The dynamic reparameterization is chosen for the implementation, but an approach to render light slab parameterized light fields using this method is also derived. The rendering method is extended to use a more realistic dynamic synthetic aperture based on the thin-lens camera model, and an efficient hardware-accelerated texture-mapping method utilizing this dynamic aperture is presented. An autofocus method is finally presented which utilizes template matching and projective geometry to instantaneously bring a selected focus point in the scene to focus.

1 Introduction

Before introducing light field rendering, it is best to start by briefly introducing the light field. The light field is a vector function that describes the amount of light that flows through every point in every direction in space. Using three spatial coordinates x, y, z and two angular coordinates θ, ϕ, this results in a 5-dimensional light field $L(x, y, z, \theta, \phi)$.

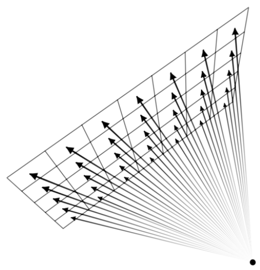


Figure 1: Light field sampled by pinhole camera.

To capture and represent the light field in practice, it is useful to consider a *pinhole camera*. As seen in figure 1, a pinhole camera only samples the light field at a single point in space located at the pinhole aperture.

It does however sample several summed ranges of directions from that single point, one for each pixel on the sensor. The entire range of directions given by the field of view of the camera can then be approximated by interpolating the resulting image. To capture larger light fields, more images may then be captured using several pinhole cameras located at different positions in space.

For simplicity, this report only considers arrays of cameras located on a camera plane as shown in figure 2. The normal vector of the camera plane is assumed to be parallel with the world z-axis, and the data cameras are assumed to be centered around the world origin.

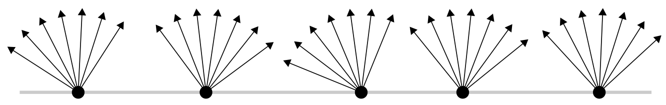


Figure 2: Pinhole camera-array

The number of sampled positions is also limited in practice, usually much more so than the number of sampled directions. Section 2.3 presents a method to similarly utilize a type of interpolation in the spatial domain to approximate the whole range of positions covered by the camera array.

Once a light field has been captured, the purpose of light field rendering is to utilize the light field to synthesize novel views. This is done by casting rays from the desired camera and querying the light field to retrieve the corresponding pixel intensities in the desired image. It is however not apparent how the 5-dimensional light field representation efficiently can be queried to retrieve the correct intensity for a given ray.

To do this, the light field is typically parameterized first to retrieve a representation that is easier to query. By considering that light moves in straight lines in free space, the light field can be parameterized between two surfaces. A ray may then be represented as a 2D surface-coordinate on each surface that represents the entrance and exit points of the ray. This is effectively a reduction to four dimensions, and querying a ray passing through a light field using this representation only requires finding two ray-surface intersections. The following two sections introduces two such parameterisations.

1.1 Light Slab Parameterization

The *light slab* parameterization was first presented by two independent papers released in 1996, *Light Field Rendering* [1] and *The Lumigraph* [2]. As the name suggests, the later called this parameterization the lumigraph. This parameterization utilizes two parallel planes to represent the light field.

In this method, images taken by the camera-array must be pre-processed with a process called *rectification*. This is done by locating common points in the scene that lie on a plane parallel with the camera plane in all images. The images are then transformed using a planar homography to align these points. This results in sheared camera projections which maps points on the chosen plane to common image coordinate in all the rectified images. The chosen plane effectively becomes the baked-in focal-plane of the light slab, whose 2D surface-coordinates has a direct correspondence with image coordinates in the rectified images.

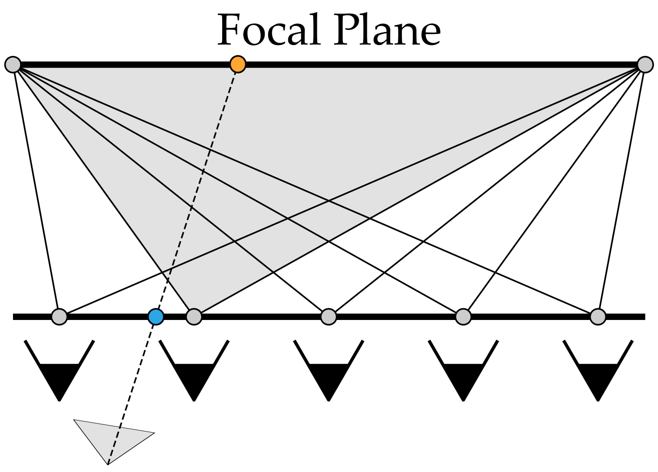


Figure 3: Light slab parameterization.
Blue point: Camera-plane intersection.
Orange point: Focal-plane intersection.

As a result, retrieving the intensity of a ray passing through the light slab can be done by simply finding the ray-plane intersections with the camera-plane and the focal-plane. The camera closest to the camera-plane intersection may then be chosen, and the surface coordinate of the focal-plane intersection can be used to locate the closest pixel from the corresponding rectified image.

The rendering methods proposed in [1] and [2] that takes advantage of this parameterization utilizes the baked-in focal plane as the focal plane of the desired camera. This is a big limitation since the focal plane of a dynamic camera is supposed to move with the camera. There are simple pixel-shift *"refocusing"* methods that can move the plane of focus back and forth, but using a fully dynamic focus plane that moves with the desired view requires other methods.

1.2 Dynamic Reparametrization

Dynamic light field reparameterization is a method that allows the focal plane to be moved around arbitrarily by adding one additional step. This parameterization was proposed in the master thesis of Aaron Isaksen [3] as well as the paper *Dynamically Reparameterized Light Fields* [4], both released in 2000.

The ray-plane intersections with the camera plane and focal plane is found as usual in this method, and the data-camera closest to the camera-plane intersection may be chosen again. The image coordinate of the corresponding image can however no longer be retrieved directly from the focal-plane intersection. Instead, the camera projection of the data-camera is used to project the intersection point on the focal plane down to the image plane of the data camera, where the closest pixel value can be retrieved directly.

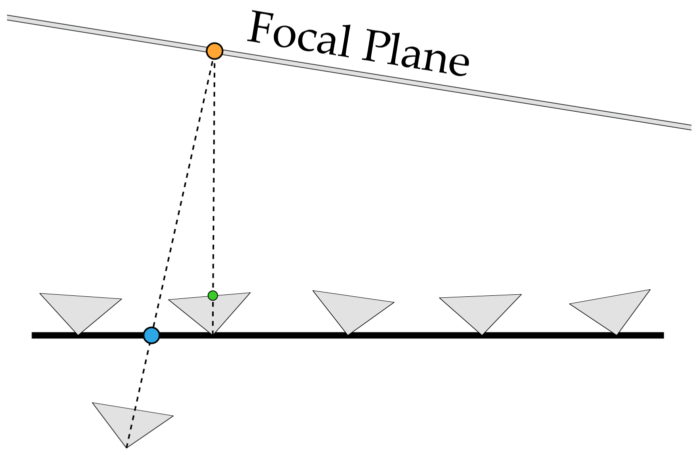


Figure 4: Dynamic reparametrization.
Blue point: Camera-plane intersection.
Orange point: Focal-plane intersection.
Green point: Point on focal plane projected to data camera.

Any camera type can be used with this method if the mapping from world coordinates to image coordinates is known. This means that the camera images do not have to be rectified like in the light slab parameterization, it is possible to use the normal perspective images taken by the camera array directly. It is however still possible to use rectified images with this approach by deriving the sheared projections. Camera projections for both normal perspective cameras and sheared cameras are presented in section 2.2.

This method is much more flexible than the light slab parameterization without any real drawbacks. This is therefore the chosen parameterization method used for the renderer.

2 Background

The following sections presents the chosen light field rendering method and describes the implementation in more detail.

2.1 Camera Model

It is useful to begin by describing the camera model used in the following sections, since this is referenced throughout the report. The camera model is the *thin-lens* camera model [5], which can be seen in figure 5.

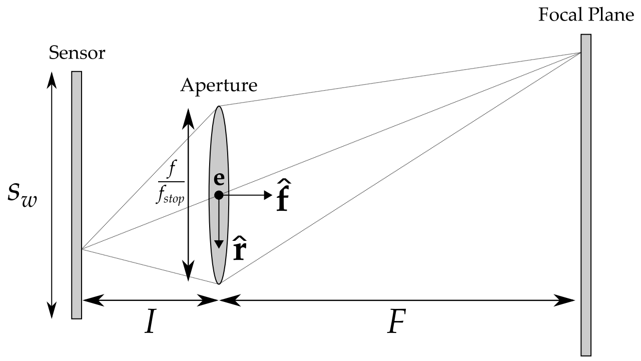


Figure 5: Top-down view of annotated thin-lens camera model.

The forward vector of the camera $\hat{\mathbf{f}}$ defines the direction from the center of the sensor plane to the center of the aperture. The vectors $\hat{\mathbf{r}}$ and $\hat{\mathbf{u}} = \hat{\mathbf{r}} \times \hat{\mathbf{f}}$ (not shown) are the right- and up-vectors respectively that span the sensor plane of the camera. These three vectors are orthogonal and constitute a left-handed orthonormal basis for the camera. The center of the aperture is the *eye*-coordinate of the camera, $\mathbf{e}$. The sensor plane is offset relative to $\mathbf{e}$ along -$\hat{\mathbf{f}}$ by the *image distance*, I. Similarly, the focal plane is offset relative to $\mathbf{e}$ along $\hat{\mathbf{f}}$ by the *focus distance*, F. The sensor width is given by s_w and the aperture diameter is given by the quotient of the focal length and the f-stop, $\frac{f}{f_{stop}}$.

Equation 1 shows how the image distance I can be derived from the focal length and the focus distance using the *thin-lens equation* [5].

$$\frac{1}{f} = \frac{1}{I} + \frac{1}{F} \Longleftrightarrow I = \frac{F \cdot f}{F - f} \tag{1}$$

Using this image distance naturally results in *focus breathing*[1] however, which can be mitigated by instead using the approximation $I = f$.

A special case of this model is the pinhole camera model, in which the f-stop is infinite and the aperture reduces to an infinitesimal point. As a result, only a single ray from each visible point in the scene can reach the sensor, regardless of distance. This model therefore produces images where the whole scene is in focus.

[1]Change of field of view as focus distance changes.

2.2 World to Image Space

This section describes how a point $\mathbf{p}$ can be transformed from world to image space using either a perspective camera or a light slab parameterized data camera. Both uses projective geometry and assumes pinhole apertures. The images are also assumed to be free of lens-distortion.

The resulting image-space coordinate $\mathbf{p}_{image}$ is transformed to the range $[(0,0)^T, (1,1)^T]$ for points visible by the camera, where $(0,0)^T$ maps to the lower-left edge of the image and $(1,1)^T$ to the upper-right edge.

2.2.1 Perspective Camera

Equation 2 shows how a world-coordinate $\mathbf{p}$ may be transformed to the 2D image-coordinate $\mathbf{p}_{image}$ of a perspective camera [6]:

$$\begin{aligned} \mathbf{p}_{clip} &= \mathrm{P} \cdot \mathrm{V} \cdot (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z, 1)^T \\ \mathbf{p}_{device} &= \frac{\mathbf{p}_{clip}}{\mathbf{p}_{clip,w}} \\ \mathbf{p}_{image} &= \frac{\mathbf{p}_{device,xy} + 1}{2} \end{aligned} \tag{2}$$

The *view*-matrix V is given by equation 3, while the *projection*-matrix P is given by equation 4.

$$\mathrm{V} = \begin{bmatrix} \hat{\mathbf{r}}_x & \hat{\mathbf{r}}_y & \hat{\mathbf{r}}_z & -\hat{\mathbf{r}} \cdot \mathbf{e} \\ \hat{\mathbf{u}}_x & \hat{\mathbf{u}}_y & \hat{\mathbf{u}}_z & -\hat{\mathbf{u}} \cdot \mathbf{e} \\ -\hat{\mathbf{f}}_x & -\hat{\mathbf{f}}_y & -\hat{\mathbf{f}}_z & \hat{\mathbf{f}} \cdot \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

$$\mathrm{P} = \begin{bmatrix} \frac{2 \cdot I}{s_w} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot I \cdot W}{s_w \cdot H} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4}$$

The new variables W and H in (4) are the pixel width and height respectively of the camera image. Note that this projection-matrix omits the near and far clipping planes.

2.2.2 Light Slab Parameterized Data Camera

The equivalent transformation for a sheared data camera from a light slab parameterized light field is given in equation 5. Note that this presupposes that the data cameras are centered at the origin, and that the camera plane normal is parallel with the z-axis.

$$\begin{aligned} \hat{\mathbf{v}} &= \frac{\mathbf{e}_d - \mathbf{p}}{\|\mathbf{e}_d - \mathbf{p}\|} \\ \mathbf{p}_{image} &= \frac{1}{2} + \frac{\mathbf{e}_{d,xy} + \hat{\mathbf{v}}_{xy} \cdot \frac{-F_{LS}}{\hat{\mathbf{v}}_z}}{\mathbf{S}_{LS}} \end{aligned} \tag{5}$$

F_{LS} is the distance between the camera plane and the light slab focal plane, $\mathbf{S}_{LS}$ is the 2D-dimensions of the light slab focal plane and $\mathbf{e}_d$ is the eye of the data camera. Note that $\mathbf{S}_{LS}$ has the same aspect ratio as the corresponding data image.

2.3 Aperture

Selecting the closest pixel from the closest camera when producing a pixel in the desired camera is analogous to nearest neighbor interpolation in 4D ray-space. This is also analogous to the pinhole camera model, where the whole synthesized view is in focus. Both the camera array and data camera images have limited resolution in practice, which results in unpleasant aliasing and flickering if this method is used. [1]

The first and simplest step to mitigate this is to use bilinear interpolation when retrieving pixel values from data cameras. [1][2] The second step is to use more than one data camera for each pixel by considering the aperture of the thin-lens camera model. This is visualized in figure 6.

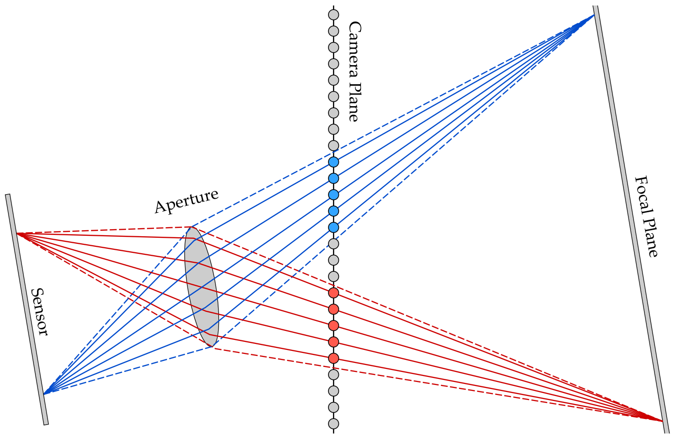


Figure 6: Synthetic aperture. The dots on the camera plane represents individual data cameras. The colored dots represents the data cameras that should be considered when producing the corresponding pixel value on the sensor.

This method is similar to the synthetic aperture filtering method presented in [4], but the aperture is now attached to the desired camera rather than being symmetrically centered at each data camera on the camera plane with a fixed size. This method should produce more realistic results where the depth of field changes dynamically with the desired camera and the chosen focus distance.

Using a naive per-pixel ray-tracing approach, this method would result in the following basic procedure to render the desired view: For each pixel in the desired camera, locate the focal plane intersection and all the data cameras that is contained within the cone that results from projecting the aperture to the focal plane intersection. Then, project the point on the focal plane to each of these data cameras to retrieve their image coordinates and corresponding bilinearly interpolated pixel values. Finally, produce an average of these values.

The texture-mapping method presented in [4] is however much more efficient, so we can do better by deriving the equivalent method using the new dynamic aperture. To do this, we consider one data camera at a time and wish to find the image region in the data camera that will contribute to an image region in the desired image.

This can be thought of as sweeping through all pixels on the sensor to find out for which pixels the given data camera is contained within the projected aperture cone. This is visualized in figure 7. The aperture is now represented as a polygon with vertices $\mathbf{a}_i$ attached to the desired camera.

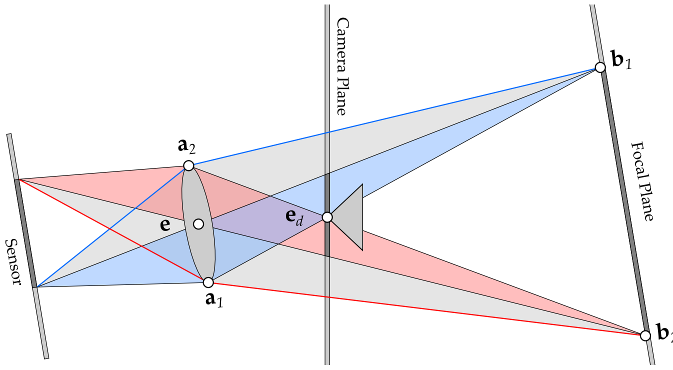


Figure 7: Visualization of the contribution of a single data camera to the desired image.

To find these image regions, we first find the vertices $\mathbf{b}_i$ at the focal plane intersection with the ray that starts at the aperture vertex $\mathbf{a}_i$ and passes through the data camera eye $\mathbf{e}_d$ as shown in equation 6.

$$\mathbf{b}_i = \mathbf{a}_i + \frac{(\mathbf{e}_d - \mathbf{a}_i) \cdot F}{(\mathbf{e}_d - \mathbf{a}_i) \cdot \hat{\mathbf{f}}} \tag{6}$$

Note that $\mathbf{b}_i$ ends up on the opposite side of $\mathbf{e}_d$ relative to the desired camera. Next, the polygon spanned by the vertices $\mathbf{b}_i$ is projected to the data camera to find the image region spanned by the polygon with vertices $\mathbf{o}_i$. Similarly, the polygon spanned by the vertices $\mathbf{b}_i$ is projected to the desired camera to find the image region spanned by the polygon with vertices $\mathbf{c}_i$ in the desired image. The image region in the data image is then simply mapped to the image region in the desired image. This is visualized in figure 8.

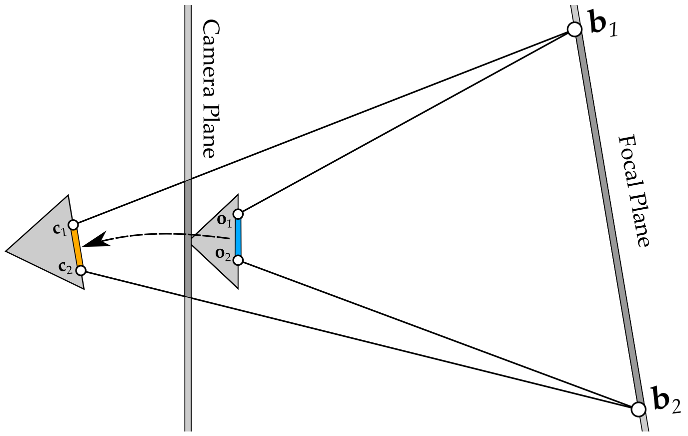


Figure 8: Focal plane polygon to image regions.

This procedure is then repeated for each data camera and their contributions are accumulated in the desired image.

Each pixel value is then divided by the total contribution for each pixel to produce an average. To simulate vignetting and to smooth out the transition between different images, each contribution may also be filtered using a radial aperture filter. [4] This filter weights the contribution of pixels at the center of the aperture more heavily than the ones at the edges.

2.3.1 Implementation

In practice, a circular aperture polygon that consists of $N+1$ vertices $\mathbf{a}_i$ paired with texture coordinates $\mathbf{t}_i$ is used. These are given by equation (7).

$$
\begin{aligned}
&i = \{0, ..., N\} \\
&\mathbf{t}_i = \begin{cases} \left(\frac{1}{2}, \frac{1}{2}\right)^T & \text{if } i = 0 \\ \frac{1}{2}+\frac{1}{2}\left(\cos\left(\frac{(i\text{-}1)\cdot 2\pi}{N}\right), \sin\left(\frac{(i\text{-}1)\cdot 2\pi}{N}\right)\right)^T & \text{otherwise} \end{cases} \\
&\mathbf{a}_i = \mathbf{e} + \frac{f}{f_{stop}} \cdot \left(\left(\mathbf{t}_{i,x} - \frac{1}{2}\right)\cdot \hat{\mathbf{r}} + \left(\mathbf{t}_{i,y} - \frac{1}{2}\right)\cdot \hat{\mathbf{u}}\right)
\end{aligned}
\tag{7}
$$

These vertices form N triangles T_j given by equation (8). The resulting aperture polygon is visualized in figure 9.

$$
\begin{aligned}
&j = \{1, ..., N\} \\
&T_j = \begin{cases} \triangle \mathbf{a}_0\mathbf{a}_N\mathbf{a}_1 & \text{if } j = N \\ \triangle \mathbf{a}_0\mathbf{a}_j\mathbf{a}_{(j+1)} & \text{otherwise} \end{cases}
\end{aligned}
\tag{8}
$$

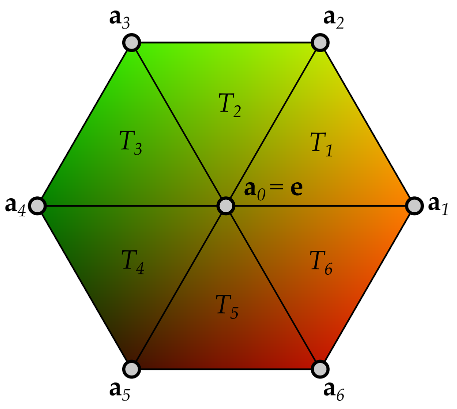


Figure 9: Aperture polygon for $N = 6$. The color corresponds to the interpolated texture coordinate $\mathbf{t}_i$, where red has been mapped to the x-component and green to the y-component.

Each aperture vertex is then processed individually in an OpenGL vertex shader program. The vertex is first projected to the focal plane using (6) to retrieve the focal plane vertex $\mathbf{b}_i$. This focal plane vertex is then projected to the data camera to retrieve the data image coordinate $\mathbf{o}_i$ using either (2) or (5) depending on the light field type. The data image coordinate $\mathbf{o}_i$ along with the aperture texture coordinate $\mathbf{t}_i$ are then set as output variables of the vertex shader program.

The focal plane vertex $\mathbf{b}_i$ is finally projected to the clip-space of the desired camera using the first equation in (2), which is set as the final output position of the vertex shader program.

OpenGL then automatically transforms this clip-space coordinate to the image space of the desired camera [6] to retrieve the image coordinate $\mathbf{c}_i$. Once this has been performed for all vertices, the resulting triangles are rasterized to pixels in the desired image [6]. In this rasterization process, the output variables $\mathbf{o}_i$ and $\mathbf{t}_i$ are interpolated for each pixel k using barycentric interpolation [6] across the screen triangles to form $\mathbf{o}_k$ and $\mathbf{t}_k$. The result is then passed along to the fragment shader program, where each pixel value $\mathbf{P}_k$ in the desired image is set individually using equation (9).

$$
\begin{aligned}
\mathbf{P}_{k,a} &= \left(1 - 2\cdot\left\|\mathbf{t}_k - \tfrac{1}{2}\right\|\right)^{\lambda} \\
\mathbf{P}_{k,rgb} &= \mathbf{D}(\mathbf{o}_k)\cdot \mathbf{p}_{k,a}
\end{aligned}
\tag{9}
$$

$\mathbf{D}(\mathbf{o})$ in (9) is the function to sample the bilinearly interpolated intensity of image coordinate $\mathbf{o}$ from the data image texture, while λ is a coefficient that controls the radial falloff of the aperture filter. $\mathbf{P}_{k,a}$ is the alpha-channel value of pixel k, which is used to store the aperture filter value.

This shader program is then used to iteratively render each data camera to the same framebuffer-object using additive blending [6]. The resulting framebuffer is then set as input texture to another shader program used to normalize the filter weights. This is simply done by dividing each pixel value by the accumulated aperture filter weight, i.e. the alpha-channel value. The result can then finally be displayed on screen.

2.4 Autofocus

The multiple perspectives of the data cameras can also be utilized to find the focal plane required to bring a point in the desired image to focus. This can be done by individually rendering two selected data cameras with the rendering method described previously. The aperture should however be ignored to project the views to the maximum area of the desired image. The rendered images are also converted from *sRGB* to luminance to reduce each pixel to a single value. This is visualized in figure 10.

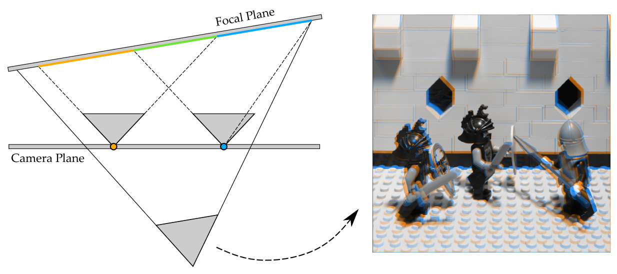


Figure 10: Disparity between two data cameras rendered to the desired view. The two images have been overlayed by mixing the color channels in the image to the right to visualize their disparity

In this example, the focal plane is located at the figure in the middle since the two images are aligned at this point. To align the images at a different focus point in the desired image, we can begin by finding the pixel disparity between corresponding points in the images at this point.

Template matching [5] is a simple method to find this disparity at a chosen focus point. This is performed by selecting one of the images as *template-image*, denoted $T(x_T, y_T)$, and the other as *search-image*, denoted $S(x_S, y_S)$. The template image is cropped to a smaller square centered at the desired focus point, while the search image can remain at its original size. To reduce the search space, the search-image may however also be cropped to a square centered at the focus point, provided that this square is large enough to encompass the disparity. The problem is then to find the position $\mathbf{p}$ in the search-image where the template-image differs the least. This is visualized in figure 11, with the hand of the figure to the right in figure 10 chosen as focus point.

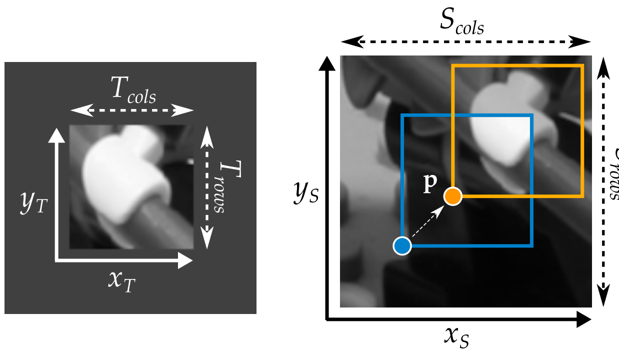


Figure 11: Template Matching.
Left: Template Image $T(x_T, y_T)$
Right: Search Image $S(x_S, y_S)$

The *sum of the squared differences* [5] measure was chosen for this purpose, denoted $R(x_S, y_S)$. The coordinate $\mathbf{p}$ may then be found as the coordinate that minimizes $R(x_S, y_S)$ over the search domain as shown in equation (10).

$$R(x_S, y_S) = \sum_{x_T}^{T_{cols}} \sum_{y_T}^{T_{rows}} (T(x_T, y_T) - S(x_S + x_T, y_S + y_T))^2$$
$$\mathbf{p} = \underset{(x_S, y_S)^T}{\arg\min}\, R(x_S, y_S) \tag{10}$$

This measure effectively works by placing the origin of the template image at the coordinate $(x_S, y_S)^T$ in the search image. The squared difference between each pixel in the template image and the underlying search image is then accumulated, and the resulting sum is the difference value for that coordinate. This is then repeated for all pixel-coordinates in the search image, and the coordinate $\mathbf{p}$ with the smallest difference represents the best match. This type of template matching is sensitive to scale and rotational changes [5], but this is not a big issue here since the images are mapped to a common view first.

Next, two pixel coordinates in the desired view may be derived from the template matching result using equation (11). These coordinates represent corresponding points in the two images.

$$\mathbf{p}_T := \text{Focus Point}$$
$$\mathbf{p}_S = \mathbf{p}_T + \mathbf{p} + \frac{(T_{cols} - S_{cols}, T_{rows} - S_{rows})^T}{2} \tag{11}$$

$\mathbf{p}_T$ in equation (11) belongs to the data camera chosen for the template image, and $\mathbf{p}_S$ to the data camera chosen for the search image. Getting these points to align in the desired view requires finding the focal plane where the data cameras sees their respective coordinate at a common point on the focal plane, $\mathbf{b}'$. This is visualized in figure 12.

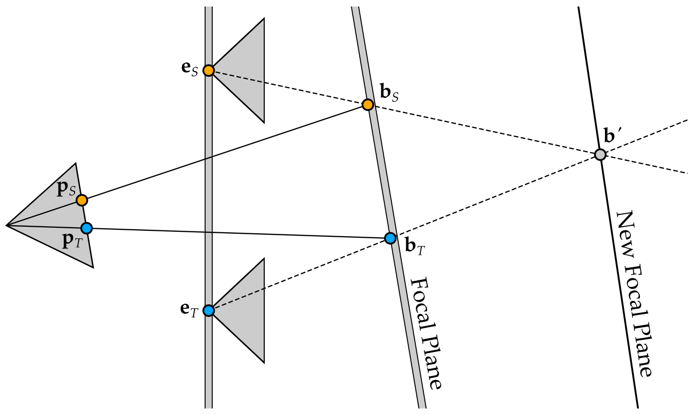


Figure 12: New focal plane from pixel disparity

This new focal plane can be found by first projecting these two pixel-coordinates back to the current focal plane using equation (12)

$$\mathbf{b}_T = \mathbf{e} + \frac{\mathbf{v}(\mathbf{p}_T) \cdot F}{\mathbf{v}(\mathbf{p}_T) \bullet \hat{\mathbf{f}}}$$
$$\mathbf{b}_S = \mathbf{e} + \frac{\mathbf{v}(\mathbf{p}_S) \cdot F}{\mathbf{v}(\mathbf{p}_S) \bullet \hat{\mathbf{f}}} \tag{12}$$

where the world-space direction-vector $\mathbf{v}(\mathbf{x})$ of a pixel-coordinate $\mathbf{x}$ in the desired camera is given by equation (13).

$$\mathbf{v}(\mathbf{x}) = \frac{s_w \cdot (\mathbf{x}_x - \frac{W}{2})}{W} \cdot \hat{\mathbf{r}} + \frac{s_w \cdot (\mathbf{x}_y - \frac{H}{2})}{W} \cdot \hat{\mathbf{u}} + I \cdot \hat{\mathbf{f}} \tag{13}$$

Note that $\mathbf{v}(\mathbf{x})$ is not normalized since it does not need to be in equation (12). The new focal plane is then located at the intersection between the rays $\vec{r}_T(t)$ and $\vec{r}_S(s)$ that are given by equation (14).

$$\vec{r}_T(t) = \mathbf{e}_T + \frac{\mathbf{b}_T - \mathbf{e}_T}{\|\mathbf{b}_T - \mathbf{e}_T\|} \cdot t = \mathbf{e}_T + \hat{\mathbf{d}}_T \cdot t$$
$$\vec{r}_S(s) = \mathbf{e}_S + \frac{\mathbf{b}_S - \mathbf{e}_S}{\|\mathbf{b}_S - \mathbf{e}_S\|} \cdot s = \mathbf{e}_S + \hat{\mathbf{d}}_S \cdot s \tag{14}$$

This intersection may however not exist since rays rarely intersect in three dimensions.

Instead, we can find the point on each ray that is closest to the other ray, and use the midpoint between these two points as an approximation of the intersection using equation (15) [7].

$$
\begin{aligned}
&k_0 = \hat{\mathbf{d}}_T \cdot \hat{\mathbf{d}}_T, \quad k_1 = \hat{\mathbf{d}}_T \cdot \hat{\mathbf{d}}_S, \quad k_2 = \hat{\mathbf{d}}_S \cdot \hat{\mathbf{d}}_S \\
&k_3 = \hat{\mathbf{d}}_T \cdot (\mathbf{e}_T - \mathbf{e}_S), \quad k_4 = \hat{\mathbf{d}}_S \cdot (\mathbf{e}_T - \mathbf{e}_S) \\
&\mathbf{b}' = \frac{1}{2} \cdot \left(\vec{r}_T \left(\frac{k_1 k_4 - k_2 k_3}{k_0 k_2 - k_1^2} \right) + \vec{r}_S \left(\frac{k_0 k_4 - k_1 k_3}{k_0 k_2 - k_1^2} \right) \right)
\end{aligned} \tag{15}
$$

The new focus distance F' that places the focal plane at this point is finally found using equation (16).

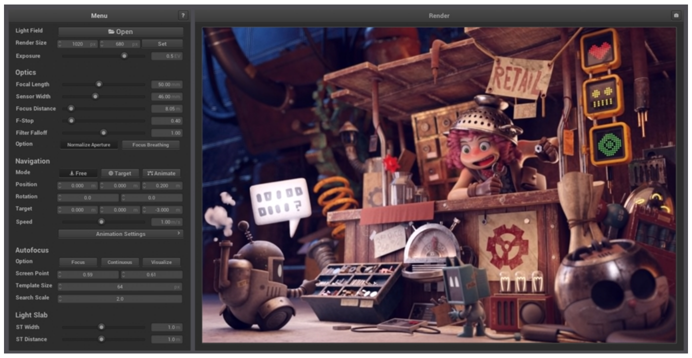
$$F' = (\mathbf{b}' - \mathbf{e}) \cdot \hat{\mathbf{f}} \tag{16}$$

3 Results

The source code for the renderer and animated results can be found on github:

github.com/linusmossberg/light-field-renderer

The renderer is implemented in C++ and uses OpenGL for hardware acceleration. It is possible to interactively navigate the scene by changing the position and rotation of the camera using the mouse and keyboard. The program has a graphical user interface that allows all aspects of the desired camera to be changed, such as focal length, focus distance, f-stop etc. The focus distance can be changed interactively using autofocus by clicking at a point in the scene. A screenshot of the renderer is shown in figure 13.



Light Field
Render Size 1020 px 680 px
Exposure 0.5 EV

Optics
Focal Length 50.00 mm
Sensor Width 46.00 mm
Focus Distance 8.05 m
F-Stop 0.40
Filter Falloff 1.00
Option

Navigation
Mode
Position 0.000 m 0.000 m 0.200 m
Rotation 0.0 0.0
Target 0.000 m 0.000 m -3.000 m
Speed 1.00 m/s

Autofocus
Option
Screen Point 0.59 0.61
Template Size 64 px
Search Scale 2.0

Light Slab
ST Width 1.0 m
ST Distance 1.0 m

Figure 13: Screenshot of the renderer.

The renderer is capable of rendering rectified light fields from the *Stanford Lego Gantry* [8]. These light fields consists of 17 × 17 images that were captured by a *Canon Digital Rebel XTi* camera. The rectified and cropped *Lego Knights* light field used here has a resolution of 1024 × 1024 pixels. The scale of these light fields are unknown, but the $\mathbf{S}_{LS}$ and L_{LS} variables in equation (5) can be changed interactively until the result seems reasonable. Figure 14 shows an example of the rendered Lego Knights light field using various focal plane positions in the scene.

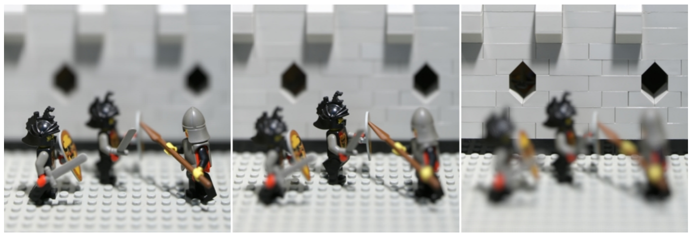


Figure 14: Variable focus distance with the Lego Knights light field. The focus distance was changed using autofocus by selecting the figure to the left, the figure in the middle and the back-wall. The focus distances from left to right are roughly: 1.2 meters, 1.4 meters and 1.6 meters.

The renderer is also able to render non-rectified light fields taken by normal perspective cameras. The *Junk Shop* [9] light field used here was captured in *Blender* [10] using a python script. The light field consists of 27 × 27 images with a resolution of 1440 × 960 pixels. The extent of this light field is 0.8 × 0.8 meters, and each data camera has a sensor width of 36 millimeters and a focal length of 30 millimeters. Figure 15 shows an example of this light field where the z-position and focal length of the desired camera is changed to get a *"dolly zoom"* effect. This is most apparent by looking at the background and the metallic sphere in the lower right in the foreground.

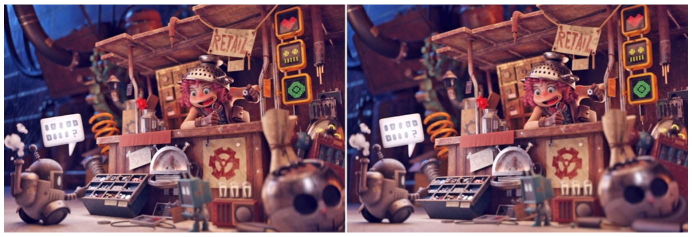


Figure 15: "Dolly zoom" using variable z-position and focal length with the Junk Shop light field. The intensity in the shadows of the image has been raised to easier see background. A negative z-position means that the desired camera is in front of the camera plane, while a positive value means that it is behind it.
Left: z-position: -0.9 meter, focal length: 43.09 millimeter.
Right: z-position: 1.1 meter, focal length: 55.9 millimeter.

This project is inherently interactive however which makes it difficult to convey most aspects in this format. I would therefore suggest compiling and running the program for more representative results.

3.1 Performance

All of the light fields presented can be rendered at interactive framerates on a system with a *NVIDIA GTX 1070* graphics card. The largest bottleneck appears to be the video memory, the Junk Shop light field require 27 × 27 × 1440 × 960 × 3 bytes ≈ 3 gigabytes for instance.

A resampled 960 $\times$ 640 pixel version of the Junk Shop light field can be rendered using a much more modest system using the integrated graphics of an *Intel Core m3-6Y30* processor. This system starts getting problems for larger apertures however since this reduces the number of pixels that can be discarded for each data camera.

4 Future Work

The most pressing thing that I have not had the time to try is using light fields with unstructured data cameras rather than confining them to a plane. Since the dynamic aperture in my method is no longer confined to the camera plane, this should already be possible with some minimal modifications to the program. This would allow the light fields to be captured directly by moving and taking pictures with a hand-held camera such as a smartphone.

Another thing I would like to try is to compare a synthesized view with a reference image captured using the same camera settings. This would be possible to do in a controlled way using Blender for example, and the images could be compared using quality metrics such as *SSIM* and *S-CIELAB*.

Lastly, I have not had the time to try using light fields with larger extents. This allows very large apertures to be simulated, which enables the possibility to effectively look through large objects.

5 Conclusion

The presented light field rendering method is able to efficiently synthesize novel views from a collection of images in a way that closely resembles physical cameras. This method may be used to for example experience a captured scene using a VR-headset, or to capture a image after the fact to more carefully compose it and select optical settings. The presented autofocus method can be used to focus at a point in the scene very quickly and accurately, and it appears to work flawlessly in most situations. This method could also easily be extended to use color information and more than two cameras for example.

References

[1] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 31–42, New York, NY, USA, 1996. Association for Computing Machinery.

[2] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 43–54, New York, NY, USA, 1996. Association for Computing Machinery.

[3] Aaron Isaksen. Dynamically reparameterized light fields, 2000.

[4] Aaron Isaksen, Leonard McMillan, and Steven J. Gortler. Dynamically reparameterized light fields. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 297–306, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[5] Richard Szeliski. *Computer Vision:Algorithms and Applications*. Springer, 2010. [September 3, 2010 draft].

[6] John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven Feiner, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley, Upper Saddle River, NJ, 3rd edition, 2013.

[7] Bit Barrel Media. 3d Math functions. `http://wiki.unity3d.com/index.php/3d_Math_functions`, 2018. [Online; accessed 2020-10-25].

[8] Vaibhav Vaish and Andrew Adams. The (new) stanford light field archive. `http://lightfield.stanford.edu/lfs.html`, 2008. [Online; accessed 2020-10-25].

[9] Alex Treviño and Anaïs Maamar. The junk shop blender scene. `https://cloud.blender.org/p/gallery/5dd6d7044441651fa3decb56`, 2019. [Online; accessed 2020-10-25].

[10] Blender Foundation. Blender. `https://www.blender.org/`, 2020. [Online; accessed 2020-10-25].