Interactive Light Field Rendering and Capture in a Commercial Game Engine

Adrian Biagioli
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213
abiagiol@andrew.cmu.edu

Abstract

Light field-based representations of images offer numerous advantages over traditional 2D approaches, allowing the user to shift the point of view or refocus the image in post-processing. This paper explores another promising application for light field-based methods: as a companion to real time renderers in video games. I present a production-ready system for efficient rendering of light fields in the Unity game engine, as well as a virtual plenoptic camera that can take light field photographs of in-game scenes. The system is also capable of importing images from existing plenoptic cameras. In this paper, I detail how the system represents, renders, and virtually captures light fields.

1. Introduction

A *light field* is a four-dimensional function $L(u, v, s, t)$ that recovers the radiance of all rays that pass through two planes (see Figure 1). Levoy & Hanrahan [1] establish a theoretical framework for representing and displaying Light Fields, and introduce methods of light field rendering and capture (both with a physical camera with careful calibration, and via software rendering). They were able to achieve real-time performance in their implementation (below 50 milliseconds per frame), albeit without running a rasterizer at the same time.

When considering the translation of their technique to the Unity game engine, significant implementation challenges remain. Firstly, there is the challenge of data storage and serialization. Of course, because lightfields are 4-dimensional, an $n \times n \times n \times n$ lightfield requires $O(n^4)$ video memory. Therefore, compression of these input textures is necessary for practical use in a game (which may already use gigabytes of video memory for other textures). However, existing widespread compression methods available in Unity (optimized for traditional 2D textures) are still inadequate for datasets as large as HD lightfields, and more creative methods will be necessary to render low-resolution lightfields in an aesthetically pleasing way.

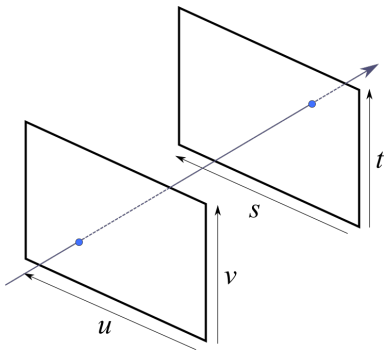


Figure 1. A light field $L(u, v, s, t)$ yields the radiance of a ray that passes through (u, v) in the front plane, and (s, t) in the rear plane. Note that the (u, v) and (s, t) planes may have different dimensions, and do not need to be parallel.

Second, for optimal performance, It is paramount that the renderer additionally takes advantage of the GPU as much as possible. As a result, most of the computational overhead in lightfield rendering is contained in shaders, written in the Cg language (which is native to Unity).

Finally, it is important to establish useful tooling that enables game artists to quickly import or generate light field data. For example, it is important to be able to import data from a traditional plenoptic camera as well as easily create lightfield renders in-engine.

2. Data Representation

The Unity game engine, like most other popular game engines, contains a variety of standard ways to store Texture data in RAM/VRAM. In addition to trivial uncompressed texture formats (for example, storing the texture as a standard 32 bit-per-pixel `ARGB` format) Unity also can compress textures in optimized formats such as `DXT1`, `DXT5`, and `BC7` [6]. Unlike formats such as JPEG and PNG, which are optimized for the highest possible compression ratio, these formats additionally aim to achieve efficient indexing and texture sampling on the GPU. This compression strategy is also

known as block compression [2].

In order to take advantage of this already-built-in functionality of the engine, it is important that we use existing texture objects that are already exposed by the Unity scripting API. For example, my initial attempts to implement light field rendering used a custom compute shader to index into an uncompressed 4-dimensional array of color data (which was precomputed on the CPU). This naive implementation (1) did not leverage the significant engineering effort done by Unity to facilitate fast texture lookup on different platforms; (2) would require even more effort to include advanced features such as trilinear filtering or mip-mapping; and (3) would require long load times as hundreds of megabytes of uncompressed texture data are streamed to the GPU, even for a modestly-sized light field.

Unfortunately, the off-the-shelf options provided by Unity are quite limited. The simplest choice is the `Texture2D` object, which is used for most textures in a traditional game. `Texture2D`s support almost all compression formats, and they are also the most feature-rich with respect to mip-mapping, texture compression, and anisotropic filtering on all platforms. 2D textures also have the best hardware support, even on decades-old platforms and mobile devices [6].

The obvious flaw with 2D textures lies in the fact that there are only two dimensions, so the lightfield data would have to be flattened from a 4-dimensional array into a 2-dimensional texture. This is not by itself a fundamental limitation. Indeed, many existing plenoptic camera systems and light field databases store their light field photographs by multiplexing (see Figure 2). However, this method is sub-optimal because it requires the use of massive textures on the order of 10000×10000 pixels for high-resolution textures. Unity has a maximum texture size of 8192×8192, so this method immediately puts a hard upper bound on light field dimension. Additionally, block compression techniques are lossy, especially within each block (in the case of `DXT5`, each block is 4×4 pixels wide [6]). If the texture were stored as in Figure 2, this would likely result in apparent smearing as the perspective is shifted due to lossy compression across the (u, v) domain.

There is also the `Texture3D` object, which is commonly used to represent 3D volumes such as clouds and MRI data. 3D textures support a form of bilinear filtering that extends to 3 dimensions. Unfortunately, this hardware filtering could not be taken advantage of to sample a 4D dataset, as it would be necessary to multiplex two dimensions from the original light field into one dimension in the 3D texture.

The solution that my implementation uses is a medium between a `Texture2D` and a `Texture3D`, called a `Texture2DArray`. The `Texture2DArray`, as the name implies, is a collection of `Texture2D`s that share the same resolution and compression method. Thus, a

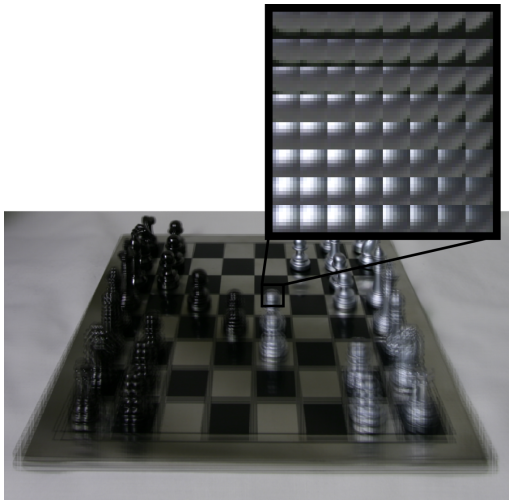
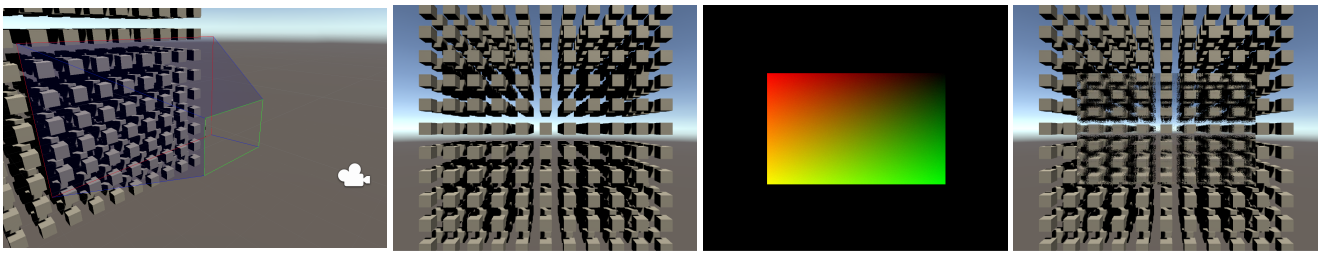


Figure 2. Multiplexing four-dimensional light field data into a two-dimensional image by fixing (s, t) over many blocks of $(\text{range}(u), \text{range}(v))$ pixels. In this instance, (s, t) light field slices have dimensions 400×700 and (u, v) slices have dimensions 8×8. Data courtesy of the Stanford Computer Graphics Laboratory [4]

`Texture2DArray` is fundamentally three-dimensional: one first indexes into the correct texture k, with $k \in \mathbb{N}$ and $k <$ the number of textures in the array. Then the texture k can be sampled by some coordinate $(i, j) \in \mathbb{R}^2$, with (i, j) within the image dimensions. Note that (i, j) need not be integers, and fractional coordinates will use bilinear sampling. Additionally the textures in the array can be compressed, and lossy compression does not introduce artifacts across adjacent textures in the array (but may introduce artifacts in the (i, j) domain, as in 2D textures).

The question now becomes: which two dimensions do we multiplex, so that a 4D light field can fit in a 3D data structure? This question is especially important in the context of texture arrays, as the multiplexed dimensions can not use bilinear filtering. One solution would be to break up a large texture from a plenoptic camera/renderer (such as the texture shown in Figure 2) into a different texture for each fixed (s, t) coordinate. For example, suppose we have a light field $L(u, v, s, t)$ with dimensions $l \times l \times w \times h$. Then for a texture array $T(i, j, k)$, we could index into it via $L(u, v, s, t) = T(u, v, s + th)$. The texture array dimensions would therefore be $l \times l \times (wh)$.

There are three problems with this approach. Firstly, as in a `Texture2D`-based approach, there would be compression artifacts across the uv plane which would result in smearing as the perspective changes. Second, light field datasets tend to provide significantly higher resolution in the (s, t) domain than in the (u, v) domain (*e.g.* Figure 2). Because of this, a local patch of rendered pixels (in screen-



(a) (b) (c) (d)

Figure 3. (a) In-engine controller of the UV (green, front) and ST (red, back) planes. The UV and ST plane dimensions, as well as their distance from each other, are user-defined. The in-game camera location is shown on the right, looking directly at the light field. (b) In-camera view of scene before light field is rendered. (c) Step one of light field rendering, which records the UV-plane coordinates to the red/green channels of a temporary render target. Because dithering is enabled, the gradient appears to be slightly noisy. (d) Final image after rendering a light field of the same scene, from a virtual plenoptic camera with a matching configuration. Light field dimensions are $16 \times 16 \times 320 \times 200$ for u, v, s, t respectively.

space) are far more likely to have similar (u, v) coordinates than similar (s, t) coordinates. Therefore the above solution would have poor locality characteristics as they would need to query different textures in the texture array[1]. Third, again due to low resolution in the (u, v) domain, bilinear sampling would create additional smearing as the perspective is shifted. It is much more useful to use these features in the high-resolution (s, t) domain. I discuss how to reconcile aliasing in the (u, v) domain later in this paper.

Therefore, supposing we begin with a light field of dimensions $l \times l \times w \times h$, my implementation creates a 2D texture array $T(i, j, k)$ with dimensions $w \times h \times l^2$, with $L(u, v, s, t) = T(s, t, u + vl)$. This means that the bilinear filtering can be used to sample within each (s, t) slice of the light field, and also compression artifacts do not appear in the (u, v) domain. I will refer to this 2D texture array as the *light field mosaic*, and it is illustrated in Figure 4.

Although my system natively stores light fields in the manner described above, it is possible to import or export data from a plenoptic texture (as in Figure 2). The conversion between a plenoptic texture and a so-called "Light Field Asset" is performed by a custom compute shader in the Unity editor. The light field asset is essentially a 2D Texture array serialized to a file. Users can also convert between a light field asset and folders containing each texture in the light field mosaic (*i.e.* a set of PNG files).

3. Display

The light field formulation of Levoy & Hanrahan [1] that my implementation uses was specifically optimized for ease of rendering in realtime with minimal overhead (importantly, only linear transformations are needed). Indeed, my implementation achieves light field rendering only by rendering two quads on screen, both of which contain a special shader. As a result, almost all of the time spent rendering the light field is contained in texture lookups.

Each frame, rendering of the light field begins after all opaque objects in the scene have been rendered. The light field rendering procedure proceeds as follows:

1. For each camera, a temporary render target is created with the same dimensions as the screen dimensions. A quad (*i.e.*, two triangles) is rendered with the same location and dimensions as the (u, v) plane, from the same perspective as the main camera. This quad is rendered with a simple shader that stores the u and v coordinates of each pixel in the red and green channels of the render target, respectively. The shader computes the appropriate coordinates using simple linear inter-

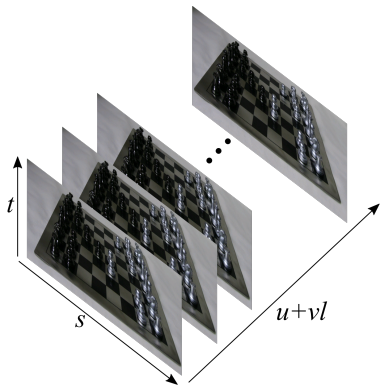


Figure 4. Multiplexing four-dimensional light field data into a 2D texture array, by fixing (u, v) for each texture in the array. This is also called a light field mosaic.

[1]It is true that the GPU does not have a cache, so locality is not nearly as important as on the CPU. However, close-together pixels tend to share the same execution units, which may be able to share texture lookups, depending on driver optimizations and hardware configuration.

polation via the vertex shader. In addition, the alpha channel is set to 1.0 when the (u, v) plane is visible and 0.0 when the (u, v) plane is invisible. An example of this temporary render target is shown in Figure 3(c).

2. The render target is switched back to the main camera. A second quad is rendered with the same location and dimensions as the (s, t) plane. This quad is rendered with a different shader. The inputs to this shader are: the light field mosaic (a 2D texture array), the render target created in the previous step (a 2D texture), and the usual Model/View/Projection matrices provided by Unity. This shader first recovers the appropriate (s, t) coordinates via vertex shader interpolation. Then, in the fragment shader, it queries the value of render target from step 1, at the same pixel as the current fragment. If the alpha value of this lookup is 0.0, then the (u, v) plane is not visible and the shader bails early. Otherwise, the shader continues with the appropriate (u, v) coordinates. With all four u, v, s, t coordinates recovered, the shader finally indexes into the light field and returns the result.

The above procedure is encoded in a Graphics Command Buffer [5], so there is minimal overhead with respect to CPU-GPU communication and engine callbacks. This method is very performant, as the majority of the render time is contained in the texture lookup step.

3.1. Minimizing Aliasing Artifacts via Dithering

The above approach to rendering is quite robust, and the limiting factor to the feasibility of light field rendering in a large game has shifted from performance to memory consumption. Compression is very effective at limiting this problem, however it is still useful to improve the rendering quality of low-resolution light fields if possible. As mentioned in Section 2, there is no bilinear filtering in the (u, v) domain, and the (u, v) domain tends to have low resolution. So the majority of visual artifacting will likely occur due to these limitations.

We would like to approximate the effect of having a higher resolution in the (u, v) domain without requiring additional texture lookups or significant computation time. The method that I implement in my system is the use of *dithering*. At a high level, dithering is a technique that simulates a high-resolution gradient by rapidly modulating between low-resolution values. Dithering takes advantage of the eye's natural ignorance of high-frequency detail in order to give the illusion of detail. For example, dithering was very common in the 8-bit and 16-bit eras of video game pixel art, to simulate true color with a limited palette.

The first step in executing dithering is to generate what is called a *threshold matrix* or *Bayer matrix*. The threshold matrix contains values in the range $[-1, 1]$. Supposing we have an $n \times n$ threshold matrix B, classical dithering can be computed as

$$I_{\text{dithered}}(x, y) = I_{\text{original}}(x, y) + B[x \bmod n, y \bmod n] \tag{1}$$

In short, the threshold matrix perturbs the output intensity, modulo n. Of course, the choice of threshold matrix is quite important. I use an algorithm proposed by Yliluoma [8] (see *Appendix 2: Threshold Matrix* in Yliluoma 2011) to build a Bayer matrix of arbitrary size. Other formulations may be used, including artistically-defined matrices, however the algorithmic approach proved sufficient for my purposes.

Recall, however, that we are not merely dithering intensities, as in Equation 1. Rather, we wish to perturb (u, v) *coordinates*. If, for example, we simply uniformly increased or decreased the average intensity of the UV render target (see Figure 3(c)), we would essentially be translating the (u, v) coordinates along the vector $(1, 1)$. Indeed, the *direction* of the transformation is important to achieve a uniform dithering effect.

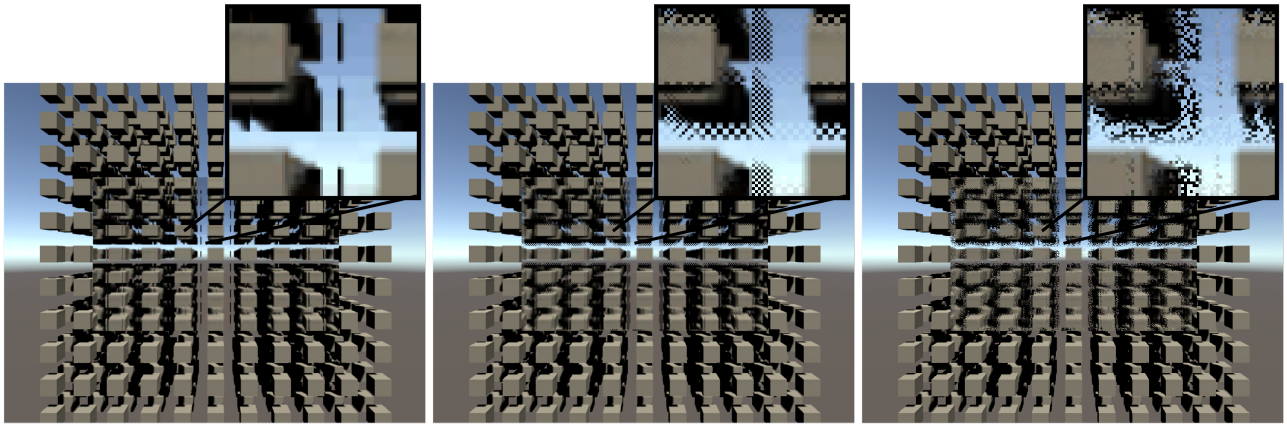
My solution is inspired by the gradient function in Ken Perlin's improved noise [3]. First, we define a hash function $h(x, y) \in \mathbb{N}^2 \to \mathbb{N}$. Next, in the UV render target shader, we use this hash to index into a library of direction vectors, using the location of the current pixel in screen space as the input. This library is arbitrary, but in my implementation I use the size-4 library of $(\pm 1, \pm 1)$. The final (u, v) value is then shifted by this vector, multiplied by the influence value from the Bayer matrix. Notice in Figure 3(c), the UV texture is very slightly noisy due to dithering. Figure 5 presents two different hash functions and the resultant renders.

4. Capture

My framework is additionally capable of capturing light field data from any Unity scene. To do this, it follows the general procedure presented by Levoy & Hanrahan [1]: A standard camera is swept across the (u, v) plane, and the frustum of that camera is skewed such that the camera's near-plane is exactly equal to the (s, t) plane in world-space. Thus, each snapshot of the skewed camera represents a set of (s, t) values for a fixed (u, v) value. In the terminology of Section 2, this is a single element of the light field mosaic. Figure 6 illustrates this skewing behavior.

In practice, this transformation is accomplished with a custom projection matrix for each camera on the (u, v) plane. Assume that the (u, v) and (s, t) planes are parallel[2]. Consider the vector space relative to the rendering camera; that is, the vector space with the camera located at the origin and the z-axis along the camera's forward vector. Let (l, u, n) be the top-left corner of the (s, t) plane in this coordinate frame, and let (r, b, n) be the bottom-right corner

[2]There is no fundamental limitation why the UV and ST planes must be parallel, however for ease of implementation my system makes this assumption.



(a) (b) (c)

Figure 5. Survey of various dithering strategies when rendering light fields. (a) No dithering, simply uses nearest-neighbor interpolation in the (u, v) domain. (b) Dithering with a hash function $h(x, y) = x$ xor y (c) Dithering with hash function inspired by Ken Perlin's improved noise [3], which indexes from a random permutation of numbers from 1 to 256. Light field resolution is $16 \times 16 \times 320 \times 200$.

of the (s, t) plane in this coordinate frame. Finally let f be the desired maximum rendering distance. Then the projection matrix can be computed as [7]:

$$
\begin{aligned}
x &= 2n/(r-l) \\
y &= 2n/(t-b) \\
a &= (r+l)/(r-l) \\
b &= (t+b)/(t-b) \\
c &= -(f+n)/(f-n) \\
d &= -(2fn)/(f-n)
\end{aligned}
\qquad
\begin{bmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
a & b & c & -1 \\
0 & 0 & d & 0
\end{bmatrix}
$$

This formulation was provided by the Unity manual page on projection matrices [7]. The complete light field mosaic is captured by moving the camera along the (u, v) plane in a uniform grid. For example, if the user would like to capture an $8 \times 8 \times 800 \times 600$ light field, then a total of 64 images are taken from different positions on the (u, v) plane, with each individual image at an 800×600 resolution. Importantly, this strategy integrates well with Unity's renderer, and no extra effort is required to support advanced rendering effects in the light field.

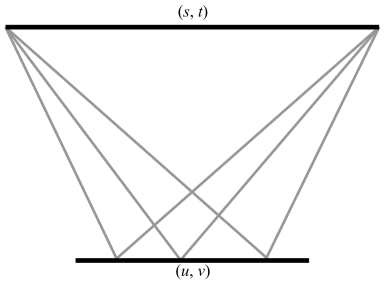


Figure 6. Visualization of the perspective skewing procedure performed during synthetic light field capture. Gray lines represent the camera frustum of 3 cameras on the UV plane, that will render to different frames of the light field mosaic.

References

[1] M. Levoy and P. 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] Microsoft. *Direct3D 10 Graphics Programming Guide: Block Compression*, May 2018.

[3] K. Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, page 681–682, New York, NY, USA, 2002. Association for Computing Machinery.

[4] Stanford Computer Graphics Laboratory. The (new) stanford light field archive. `lightfield.stanford.edu`.

[5] Unity Technologies. *Unity Manual: Graphics Command Buffers*, 2019.2 edition, 2019. `https://docs.unity3d.com/Manual/GraphicsCommandBuffers.html`.

[6] Unity Technologies. *Unity Manual: Texture compression formats for platform-specific overrides*, 2019.2 edition, April 2019. `https://docs.unity3d.com/Manual/class-TextureImporterOverride.html`.

[7] Unity Technologies. *Unity Scripting Reference: Camera.projectionMatrix*, 2019.2 edition, 2019. `https://docs.unity3d.com/ScriptReference/Camera-projectionMatrix.html`.

[8] J. Yliluoma. Joel Yliluoma's arbitrary-palette positional dithering algorithm, January 2011. `https://bisqwit.iki.fi/story/howto/dither/jy/`.