LightShop: Interactive Light Field Manipulation and Rendering

Daniel Reiter Horn* Stanford University

Billy Chen† Microsoft

Abstract

Light fields can be used to represent an object's appearance with a high degree of realism. However, unlike their geometric counterparts, these image-based representations lack user control for manipulating them. We present a system that allows a user to interactively manipulate, composite and render multiple light fields. LightShop is a modular system consisting of three parts: 1) a set of functions that allow a user to model a scene containing multiple light fields, 2) a ray-shading language that describes how an image should be constructed from a set of light fields, and 3) a real-time light field rendering system in OpenGL that can plug into existing 3D engines as a GLSL shader.

We show applications in digital photography and we demonstrate how to integrate light fields into a modern space-flight game using LightShop.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.3 [Computer Graphics]: Image Based Modeling—Systems

1 Introduction

A light field [Levoy and Hanrahan 1996; Gortler et al. 1996] is a four-dimensional function mapping rays in free space to radiance. Using light fields to represent scene information has become a popular technique for rendering photo-realistic images. However, most systems that incorporate light fields use a rigid architecture that is suitable for a single task, like view-interpolation, focusing, or deformation [Levoy and Hanrahan 1996; Vaish et al. 2004; Chen et al. 2005]. A general system that accomplishes all these tasks allows a user to creatively manipulate light fields by combining such tasks together. In this paper, we present such a system, LightShop, that allows a user to interactively manipulate and composite 4D light fields in a single, unified framework.

This unified framework is an important advantage of LightShop. The system is designed to manipulate and render light fields independent of parameterization and acquisition method. In our results, we show composite scenes of light fields captured from a camera array [Wilburn et al. 2005], a hand-held light field camera [Ng et al. 2005], a gantry-arm [Levoy 2004] and a ray-tracer. This mix of light fields can manipulated in a number of ways, deformed, composited, or refracted, to name a few. This enables a programmer to quickly prototype new algorithms for manipulating light fields, independent of their representation. Another use is as an education tool for light fields.

In addition the LightShop interface is designed to be simple and amenable to graphics hardware, with the goal that game developers may adopt light fields as an additional "augmented" texture source. In fact, we demonstrate the use of LightShop by integrating a light field into a full open source video game. This paper summarizes the design decisions necessary for building a practical system for manipulating light fields.

LightShop consists of three parts: 1) an API that constitutes a modeling interface for defining a scene containing light fields, 2) a ray-shading language that defines how a 2D image is rendered from this scene and 3) a rendering system that generates an image from a specified scene and a ray-shading program.

The modeling interface is an API, similar to Renderman [Upstill 1992] or OpenGL [Board et al. 2005] that allows the programmer to define virtual cameras and insert light fields into the graphics environment.

Once a scene has been defined, the user writes a ray-shading program that describes how to generate an image from the graphics environment. This program executes on every pixel location of the output 2D image. In the program, a "view-ray" is formed based on the pixel location and the camera parameters. The program describes how the view-ray will interact with the scene of light fields to determine the radiance (i.e. color) in its direction. The color of the output pixel is set to the color of the view-ray.

One of the key features of LightShop is that manipulating light fields is described as an operation on view rays. For example, to deform a light field the user writes a program that systematically warps the view-rays. To assist the user, LightShop provides several useful functions for manipulation: sampling from a light field, warping, and compositing. Such functions can be combined simply by writing the appropriate ray-shading program. In the results section, we illustrate how multiple functions that operate on individual light fields can be combined with each other.

After defining a ray-shading program, LightShop's rendering system executes the program for every pixel of the final 2D image, forming the output.

The rest of the paper is organized as follows. First, we describe related work in the area of image-based editing. Second, we review the light field construction and present the LightShop specification. We demonstrate how to use this system with a simple example. Third, we present an OpenGL/GLSL implementation that enables interactive editing and manipulation. Finally, we demonstrate our implementation for digital photography and image-based modeling as well as its integration as a plugin into a modern 3D space game.

2 Related work

Since most light field datasets are represented by multiple images, light field manipulation is one type of image-based

*email: danielrh@graphics.stanford.edu

†email: bill.chen@microsoft.com

editing. Other types of image-based editing can be categorized by their dimensionality. 2D image-based editing is the most mature and includes industry-tested tools like Adobe Photoshop [Adobe 2000]. See Gonzalez and Woods [2002] for a review of common 2D image operations.

3D image-based editing refers to editing video or manipulating images with depth information. In the former, tools like Adobe Premiere offer a variety of operations from cutting to merging or blending videos. In the latter, researchers have developed tools for augmenting single images with depth information. Oh et al. [2001] built a system that allows the user to manually specify depths in an image. Subsequent editing operations, like cloning and filtering, can be performed while respecting scene depth. Barsky [2004] applies focusing operators to a similar image representation to simulate how a scene would be observed through a human optical system.

Seitz and Kutulakos [1998] use a voxel-representation for depth information. They introduce editing operations such as scissoring and morphing on this voxel representation. Meneveaux and Fournier [2002] use a similar representation and extract normals and reflectance properties for a reshading operation.

4D image-based editing refers to light field manipulation. With the exception of a few works, this area has remained unexplored. Shum and Sun [2004] extend 2D compositing [Porter and Duff 1984] to light fields. They introduce a technique called *coherence matting* for seamless compositing of light fields. Two other systems allow for warping light fields [Zhang et al. 2002; Chen et al. 2005]. In the former, two input light fields are combined to produce a morphed version. In the latter, an animator can interactively deform an object represented by a light field. Our system enables operations such as these to be combined and executed on a mix of light fields.

Image-based editing, regardless of dimension, needs to be interactive. In LightShop, we achieve interactive editing by utilizing graphics hardware acceleration. Previous systems have also used graphics hardware but only for light field rendering. Chen et al. [2002] introduce a real-time rendering algorithm for surface light fields that approximates the data with lower-dimensional functions over elementary surface primitives. Vlasic et al. [2003] extend this with opacity hulls for view-dependent shape as well as appearance.

3 The 4D Light Field

Before describing LightShop, we review the "light field" image-based representation. LightShop takes light fields as input, composites and manipulates them, and outputs a novel view of the scene.

A light field is a continuous 4D function mapping rays in free space to radiance. The input, a ray in free space, takes 4 coordinates to represent [Levoy and Hanrahan 1996]. The output, radiance, is usually approximated by a three-component vector RGB. Similar to the light field compositing work by Shum and Sun [2004], we augment the RGB vector with an alpha channel [Porter and Duff 1984] that describes the opacity along that ray.

To perform computation on a light field, we represent it with discrete samples. A *discrete light field* is simply a 4D texture map, and henceforth will be referred to simply as a "light field". In practice, acquired light fields are represented as a set of camera images. In this case, the light field takes four coordinates that select camera and pixel, and returns the RGBA color of that camera pixel. The alpha can be computed by using matting techniques [Smith and Blinn 1996] or manually [Shum and Sun 2004]. These light fields are the input to LightShop.

4 LightShop: A System for Manipulating Light Fields

To manipulate light fields, we design LightShop to use the same conceptual model as OpenGL for manipulating polygonal objects [Board et al. 2005]. By doing so, a programmer familiar with OpenGL can easily adapt to using LightShop. We characterize OpenGL's conceptual model as one that *models* a scene containing multiple objects, *manipulates* these objects, and *renders* an output image based on the modified scene. To model and manipulate a scene the programmer uses the OpenGL API. To render an image, the programmer writes a vertex and a fragment program.

LightShop is organized in a similar way to OpenGL, with an API for modeling the scene, a ray-shading language for manipulating it, and a rendering system. The modeling API exports a set of functions that are used to define a scene containing light fields. The ray-shading language is used to describe how that scene should be rendered to a 2D image, e.g. how a view-ray is shaded, given multiple light fields in the scene. LightShop's renderer then executes the user-defined ray-shading program at each pixel of the output image. Each execution of the program shades a single pixel until the entire image is rendered.

To use the interface, a programmer makes a series of procedure calls to setup a scene with light fields. These include positioning light fields and defining viewing cameras. Then the programmer uses the ray-shading language to describe how a view ray from a selected camera is shaded as it interacts with the light fields.

First we describe LightShop's modeling interface. Then, we describe a ray-shading language that enables the programmer to manipulate the scene and to specify how a 2D image should be rendered.

4.1 LightShop's Modeling Interface

In LightShop, a scene is modeled with two primitive types: cameras with a single lens and light fields. The programmer calls on an API to insert these primitives into an internal representation of the scene. Each primitive has related attributes. For example, the camera primitive has a simple lens model so it has attributes that describe an image plane, a lens plane, and a focal plane.

Light fields have attributes associated with texture sampling. These include the `sampling` (i.e. nearest-neighbor, or quadralinear) and `wrapping` (i.e. repeat or clamp to a value) behaviors. Nearest-neighbor sampling simply extracts the color of the ray "nearest" to the input ray. Quadralinear sampling [Levoy and Hanrahan 1996] is the 4D equivalent to bilinear interpolation in 2D. In addition to these attributes, a special "transform" attribute is used to specify the position and orientation of each light field.

These function calls, either to insert primitives or to modify their attributes, update the internal representation, which we call the *graphics environment*. The graphics environment is a set of cameras and light fields shared between the modeling API and the ray-shading language.

4.2 LightShop's Ray-shading Language

After using the modeling interface to define a scene containing light fields, the programmer writes a ray-shading program that effectively manipulates the scene and precisely defines how this scene should be rendered to a 2D image. An image is created by associating a ray to each output pixel, and deciding on how to shade this "view-ray". As the view-ray travels through the scene, its color (RGBA) or direction may change due to interaction with light fields. This is similar to ray-tracing in a scene where objects are represented by light fields.

The ray-shading language executes in a manner similar to the Pixel Stream Editor [Perlin 1985]. It takes as input the xy location of a pixel of the 2D image, executes the ray-shading program at this pixel, and outputs a RGBA color. At any one pixel, the program has access to the graphics environment. It uses this environment to form a ray from the given pixel position and to shade the color of this ray. LightShop's renderer executes the same ray-shading program at each pixel location to determine the color for each pixel in framebuffer. Combinations of editing operations can be executed by simply calling on multiple editing functions.

First we briefly describe the language features. Then we introduce our set of high-level functions available for interactive light field editing.

4.2.1 Language Features

The ray-shading language is built upon the GLSL specification. The program execution begins in the `main` procedure for each pixel. The `main` procedure accesses the graphics environment to decide how to form a ray and shade it, and then it returns a RGBA color.

In order to make this computation easier to program, we have implemented several useful functions. The first function, `LiGetRay`, takes as input a 2D pixel location, a 2D position on the lens aperture and a camera specification. It computes the ray that shoots from the pixel location through the sample position on the lens, which refracts out from the camera. The amount of refraction is based on the simple lens model. For simplicity, we represent rays with a 3D point and a 3D direction vector.

Once a ray has been formed, it can be passed as input to several high-level functions. These functions are defined in LightShop because they are common in many light field operations.

4.2.2 4D Light field Sampling

The sampling procedure takes as input a ray and a light field and returns the color in that ray direction. Because light fields in LightShop are represented in a sampled form, any given ray direction may not have a color in the light field. Hence, the procedure utilizes the `sampling` light field attribute to determine how it should return a color for any given ray. The sampling procedure is most commonly used for novel view synthesis from light fields.

```
LtColor LiSampleLF(LtInt lightfieldID, LtRay ray)
```

4.2.3 Compositing

Using the proper compositing operators and ordering allows a programmer to render an image of a scene containing multiple light fields. Recall that a color contains RGBA channels. RGB approximates the radiance along a ray direction in the light field. Alpha, or A, represents the ray's opacity and coverage. Once the color along a ray has been sampled from a light field, it can be composited with other colors (along rays) sampled from other light fields.

We implement Porter and Duff's [1984] compositing operations. For example, the function call for the over operator is shown below:

```
LtColor LiOver(LtColor A, LtColor B)
```

4.2.4 Warping

Ray warping is commonly used to simulate deformation of a light field [Chen et al. 2005], or refractive effects [Heidrich et al. 1999; Yu et al. 2005]. We implement warping by a set of functions that take a ray as input and return a new ray.

LightShop provides two types of warps to the programmer: procedural warps and 4D table lookup. Procedural warps are linear transformations on 3D vectors. Since we represent a ray as a point and direction, warping it involves multiplying both components by a matrix, which is fast on graphics hardware.

To enable more flexible warping, LightShop offers a 4D lookup table option. With a lookup table the ray warping function is approximated by many samples. The table itself is loaded as a light field[1], except that the values are not color, but ray coordinates. One can think of the light field as a *ray*-valued one as opposed to a *color*-valued one. The LightShop procedure that samples from the ray-valued light field is shown below.

```
LtRay LiWarpRayLookup(LtInt lightFieldID, LtRay ray)
```

5 A Simple Example

We now describe an example that illustrates the expressive power of the LightShop system. We delay the discussion of the implementation to Section 6. The final result that LightShop renders is shown in Figure 3d.

First, we describe the input to LightShop. The scene consists of 4 light fields. Two of them represent a Buddha and a flower. The third light field represents a table lookup for a ray warp that simulates the refraction of rays through a glass ball[2]. The fourth light field represents the specular highlight of that ball.

The procedure calls that model the scene are shown in Figure 1. Referring to Figure 1, the programmer first inserts a camera into the scene. Specifying a lens with a width and height of zero describes a pinhole camera. The next set of procedure calls insert light fields into the scene.

Once the scene has been modeled, the programmer writes a ray-shading program that defines precisely how a 2D image is rendered from this scene. This is done by writing a program that executes per output pixel of the image to determine the color of each pixel, given the scene of light fields. Figure 2 shows the ray-shading program.

In lines 5–13 we convert the current pixel location into a ray and use this ray to sample from the Buddha light field. We then set the background color to be black. Figure 3a shows the image after executing these lines of code.

Next, in line 17 we use the same ray to sample from the flower light field and composite that color over the Buddha sample, which produces Figure 3b.

[1] For simplicity, we use two light fields to store the 6 coordinates defining points and directions of rays. A 4-coordinate ray-representation could also be used.

[2] The lookup table is computed by ray-tracing through a sphere with glass material properties.

```
// insert the camera
LtInt camera0 = LiCamera();
LiAttributeCam(camera0, "lower left", 4.14, 4, 7.92);
LiAttributeCam(camera0, "up", up, 0, -8, 0);
LiAttributeCam(camera0, "right", -7.99, 0, 0.14);
LiAttributeCam(camera0, "lens width", 0);
LiAttributeCam(camera0, "lens height", 0);

// insert the light fields
LtInt buddha = LiLightField("buddha");
LtInt flower = LiLightField("flower");
LtInt glassBall = LiLightField("glass ball");
LtInt highlight = LiLightField("highlight");

// set light field attributes
LiAttributeLF(buddha, "transform", {4.0,0, 0,0,...});
LiAttributeLF(flower, "transform", {0.6,0, 0,0,...});
...
```

Figure 1: LightShop procedure calls that model the toy scene shown in Figure 3d.

```
00 LtColor main(LtVec2 currentPixel) {
       // the output color for this pixel
       LtColor col;

       // form a ray from the current pixel
05     LtRay ray=LiGetRay(camera0,currentPixel,LtVec2(0,0));

       // set the background color to be black
       LtColor background = LtVec4(0,0,0,1);
       col = background;
10
       // sample from the Buddha light field
       // and composite over a black background
       col = LiOver(LiSampleLF(buddha, ray), col);

15     // sample from the flower light field and
       // composite it over the buddha one
       col = LiOver(SampleLF(flower, ray), col);

       // warp view ray to simulate the refraction effect
20     LtRay warpedRay = LiWarpRayLookup(glassBall, ray);

       if(warpedRay.dir != 0) {
          LtColor refractedBuddha = LiSampleLF(buddha, warpedRay);
          LtColor refractedFlower = LiSampleLF(flower, warpedRay);
25        LtColor refraction = LiOver(refractedFlower,
                       LiOver(refractedBuddha, background));

          // tint the refracted ray color
          LtColor tint = LtVec4(1, .5, .5, 1);
30        refraction = tint * refraction;

          // composite refracted color to output pixel color
          col = LiOver(refraction, col);
       }
40     // obtain the specular highlight of
       // the glass ball and add it to the scene
       LtColor highlightColor = LiSampleLF(highlight, ray);
45     col = col + highlightColor;
       return col;
   }
```

Figure 2: An example ray-shading program. It renders the image shown in Figure 3d. For clarity, the light field identifiers for `LiSampleLF` have been written as the name of the light field, instead of an integer.

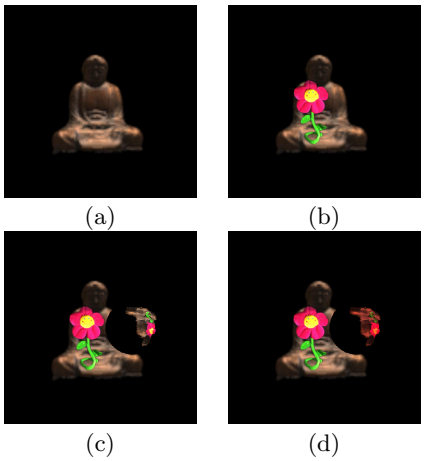


Figure 3: (a) The image after sampling from the Buddha light field. (b) The image after compositing the flower RGBA sample over the Buddha one. (c) The image after compositing the refracted Buddha and flower light fields. (d) The image after creating a red tint in the glass ball refraction.

In lines 20–26 we simulate the spherical refraction effect by warping the view ray as if it had gone through the glass ball. Recall that the glass ball light field is a table lookup mapping an input ray to a warped ray. We use the `LiWarpRayLookup` procedure to acquire the warped ray. This warped ray is then used to sample from the Buddha and the flower light field to produce a refracted color. Figure 3c shows the current image.

In addition to computing a refracted image of the Buddha and flower, we can also adjust the tint of the glass ball simply by introducing a multiplicative factor before compositing the refracted images as in lines 29–30.

Finally, in lines 44-46 we add a specular highlight to the scene by sampling from the light field containing the ball's specular highlight and adding this color to the final color. This produces the final image, as shown in Figure 3d. Although we show the final result as a single image, LightShop is capable of rendering different viewpoints by changing the camera position through the modeling API. Doing so would illustrate view dependent effects in the rendered image.

6 Implementation

Our system supports the two plane [Levoy and Hanrahan 1996] and sphere-plane parameterizations. The two plane parameterization is a natural parameterization for datasets acquired from an array of cameras. The UV-plane is defined as the plane on which the cameras lie. The ST-plane is the plane on which all camera images are rectified. Unfortunately, a single two plane parameterization cannot represent rays parallel to the planes. This makes it difficult to capture objects with inward-looking light fields. Therefore, for capturing objects, we use a sphere-plane parameterization.

The sphere-plane parameterization is defined by a sphere defining the manifold containing all camera positions and a family of planes defining each camera's image plane. Sampling a ray involves first intersecting the ray with the sphere, then finding the nearest cameras to that point, and then sampling from the associated image planes.

To represent the light field on disk, we take images from either parameterization and compress them into a file using

4:1 ratio S3 texture compression [Iourcha et al. 1999].

LightShop's modeling interface is a class hierarchy in C++ and the ray-shading language comprises a set of helper utilities within GLSL. The computational framework of writing a single program that executes per output pixel is exactly the same as the framework for a fragment shader. Additionally GLSL is designed for real-time rendering. This enables LightShop to be used as an interactive editing tool.

To make the interaction more practical, when LightShop launches, it brings up a text editor allowing the user to interactively change the ray-shading code displaying the scene. Additionally LightShop's ray-shading language has methods to query the keyboard, so a user can write a shader that reacts to input.

To implement the graphics environment, we exploit OpenGL's graphics state. When the programmer uses the modeling interface to define a camera, LightShop defines specially named program constants so that GLSL may access them. When a procedure is called to insert a light field into the scene, our LightShop implementation loads compressed datafiles from disk to graphics memory in the form of a 3D texture. This allows the light fields to be accessible by the fragment shader in GLSL, which is our implementation of the ray-shading language.

As the programmer writes a ray-shading program, he may access the camera parameters via reserved LightShop uniform variables. Light fields are sampled using the `LiSampleLF` procedure discussed in Section 4.2.2.

After writing the ray-shading program (which is essentially a fragment program with extra LightShop state), the program is run through a preprocessor that converts it into valid GLSL code. This code is compiled by the graphics driver and linked into the rendering program for execution.

The LightShop renderer that executes the ray-shading program per output pixel is simply the OpenGL fragment renderer. To force the fragment shader to execute over every pixel in the output, we define a single OpenGL quad that fills the display screen.

7 Results

We show results demonstrating how LightShop can be used in digital photography and video games. The first result shows creating a scene with multiple light fields. The light fields have been deformed and relit to create a believable composite. The second result shows how novel focusing techniques can be explored by manipulating how we sum view-rays through a camera lens. The final result demonstrates how light fields can be easily integrated into a complete video game using LightShop. To render the results, we use a P4 2.6 GHZ with 1 GB RAM, using a Nvidia 6600 GT with 128 MB onboard.

7.1 Compositing a Wedding Scene

We use LightShop to create a composite light field of a wedding scene. Figure 4a shows one image from a wedding light field captured using a hand-held light field camera [Ng et al. 2005] and three images from light fields of actors. Each actor is captured in front of a green screen using the Stanford Multi-Camera Array [Wilburn et al. 2005]. The green screen is for matte extraction [Smith and Blinn 1996]. Additionally, we acquire a light field of each person under two lighting conditions: left light on and right light on. In LightShop, we can simulate coarse relighting by taking different linear combinations of colors in these light fields.

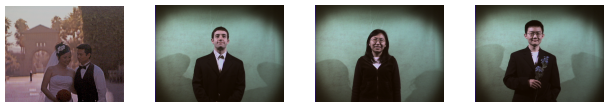


Figure 4: Image from a light field of a wedding couple and images from three light fields of three individuals in front of a green screen. The wedding light field, captured by Ren Ng, is 10 MB. Each actor dataset, for a given lighting condition, is 15 MB.

(a) (b)

Figure 5: (a) An image from the composite light field. To approximate the illumination conditions in the wedding light field, we take linear combinations of the light fields of a given individual under different lighting conditions. The illumination would match better if more lighting conditions were captured. (b) An image from the composite light field where we have turned two individuals' heads.

In Figure 5a, we use LightShop to composite the actors into the wedding scene to produce a composite light field. Figure 5b shows an image after applying a ray deformation to turn two of the actors' heads. The ray-shading program is similar to the one shown for the example in Section 5. This image is difficult to create using conventional 2D editing tools. The pixels that form this image are selected from more than 436 images. Also, moving LightShop's virtual camera would exhibit the proper parallax between light fields. The image is rendered at 40 frames per second (FPS).

7.2 Manipulating Focus in Sports Photography

This result demonstrates how we can explore novel focusing techniques simply by modifying how view-rays through the lens are summed, for each pixel. For example, focusing on a single depth in a scene can be accomplished by summing the colors of rays passing through the lens aperture. In the ray-shading program, this is accomplished by refracting multiple rays through the lens aperture and summing the resulting colors along each ray:

```
LtColor col = LtVec4(0,0,0,0);
for(LiInt i = -1; i < 1; i += stepSizeX)
    for(LiInt j = -1; j < 1; j += stepSizeY)
        col += LiSampleLF(0, LiGetRay(0, currentPixel, LtVec2(i,j)));
```

`stepSizeX` and `stepSizeY` refer to user-specified step sizes on the lens aperture. Figure 6a illustrates this simple focusing technique on the swimmers light field.

However, since LightShop provides full control over how colors of rays should be summed, we can modify the above code segment to produce non-photorealistic effects like having multiple-depths of focus. This may be useful in sports photographs, when sometimes areas of interest occur in multiple depths. A sports photographer may want to focus on these depths. For example, the photographer may want to

focus on the front and back swimmers as shown in Figure 6b. Alternatively, the photographer may want to focus on the mid, and back swimmers, but create a sharp focus transition to the front swimmer, as shown in Figure 6c.

This non-photorealistic form of imaging is accomplished by first segmenting the light field into four layers, three for the swimmers, and one for the crowd. A "layer" is a light field containing rays that are incident to objects at a particular depth in the light field (i.e. the front swimmer light field layer). We segment the swimmers light field by incrementally applying Bayesian matte extraction [Chuang et al. 2001] for each layer, in a front to back order [Chen 2006].

Second, the user inserts each layer (e.g. light field) into the scene. Then the user inserts four cameras into the scene, one for each light field. The cameras have the same attributes (e.g. image plane, lens aperture, lens position, etc.) except for the focal distance. Each camera is used to create an image with shallow depth of field from one of the four light fields. In each image, the corresponding layer may be in focus or not. These four images are then composited together with the `LiOver` operator to form the multi-focal plane image. Figures 6b and c illustrate this focusing effect. We sample the lens 256 times, so the system runs at 0.5 FPS.

7.3 Incorporating LightShop into a Game

Because LightShop is implemented in OpenGL and GLSL, it can be utilized as a plugin in interactive games. Games that utilize fragment shaders may use LightShop's functions to access light fields like any other texture. To properly draw a light field, the vertex shader needs only to define a quad spanning the bounding box of the object that the light field represents so that the fragment shader may execute LightShop's ray-shading program to color each pixel occupied by the object. From the game programmer's point of view, LightShop provides an interface for a "3D billboard"[3] [McReynolds et al. 1998; Akenine-Möller and Haines 2002].

To demonstrate LightShop's use in interactive games, we integrated a light field into *Vega Strike*, an open-source OpenGL-based space game [Horn et al. 2006] in active development since 2001.

The light field that we wish to insert into Vega Strike represents a red toy ship. The acquired light field is uniformly sampled [Camahort et al. 1998] around the toy ship and is approximately 125 MB.

In Vega Strike, each model has multiple meshes defining its geometry. Each mesh in turn has one associated 2D texture map. In the game loop, when a mesh is scheduled to be drawn at a particular location, the appropriate `MODELWVIEW` matrix is loaded into OpenGL and the associated texture is made active. The mesh vertices are then passed to OpenGL, along with the associated texture coordinates.

To integrate LightShop into Vega Strike, we define a Texture4D class to load in light field data. Texture4D is a subclass of the standard 2D texture other meshes use. The mesh for a light field object is simply a unit quadrilateral. The vertex shader bound to the mesh takes this quadrilateral and maps it to the correct screen coordinates, depending on the location of the view camera and the light field, to cover the screen-space bounds of the object. When the Texture4D class is activated, the appropriate `MODELVIEW` matrix is pulled from OpenGL and the game camera parameters are fed into the LightShop camera model. A scene with just one item is loaded into LightShop, and the resulting fragment shader (a ray-shading program) is activated when the light field is ready to be drawn. Figure 7 shows the light field of the toy ship integrated into the game.

The LightShop-rendered light field acts as any other active unit in Vega Strike, complete with AI and physics models. Collisions are modeled using a coarse geometric proxy for the red spaceship. The effect of having a light field object in view on the Vega Strike framerate is minimal and the game still achieves greater than 30 FPS. This is because the quadrilateral billboard comprises exactly two triangles and when they are drawn, there is no overdraw. Likewise the shader just samples the light field four times using the builtin bilinear interpolation to complete the quadralinear interpolation per pixel.

Thus LightShop can be plugged into any OpenGL application with minimal intervention and hence, light fields can be integrated into the standard graphics pipeline. This makes it practical to acquire real-world objects and place them into games or other graphics applications replete with polygonal items. This demonstrates the first integration of an acquired light field into a game.

8 Conclusion

We have presented a system, LightShop, that allows a user to manipulate and composite light fields at interactive rates. The system provides a unifying framework in which many of these operations can enhance each other by being combined. Furthermore, the light fields do not need to be the same size, or parameterized or acquired in the same way. We have shown that LightShop can manipulate light fields of people, wedding scenes, swimmers, and fluffy toys – all of which are nearly impossible to model with polygons. This system has applications in creating novel imaging effects, digital photography, and interactive games.

The system has its limitations. Since the input to LightShop is a 4D light field where illumination is fixed, compositing different light fields can look incorrect. This is the same limitation that other editing tools, like Adobe Photoshop, have. In LightShop, this can be addressed by capturing a light field of an object under multiple illumination conditions and writing a ray-shading program that takes linear combinations from each light field to mimic the target illumination. However this results in an order of magnitude data explosion for even the simplest of diffuse lighting conditions. However, there is hope that compression techniques could contribute to addressing this problem.

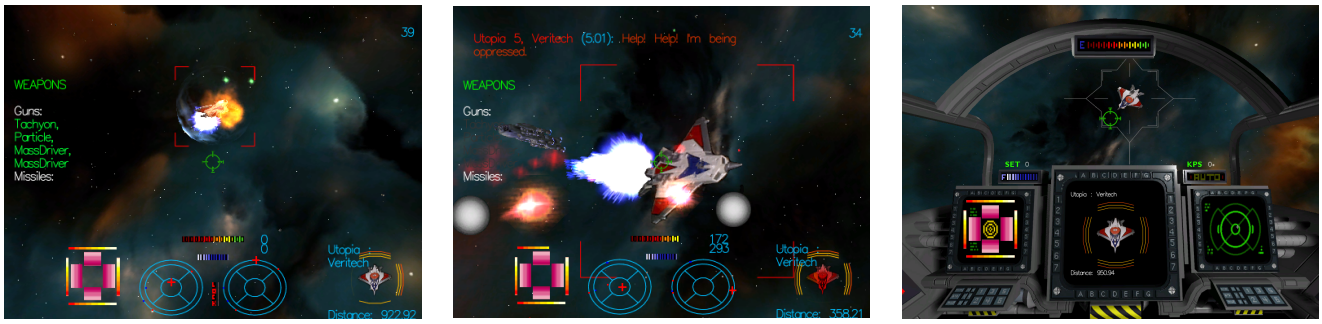
Another drawback of LightShop is the programmer-centric interface for interactively compositing and manipulating scenes. Due to the wide variety of scenes and techniques we wished to support, we decided a full programming language was necessary. The problem of visual programming of shaders is being addressed by a number of commercial engines, for example the Unreal engine [Sweeney and Epic 2006]. Thus instead of having a UI to composite and place light fields we mandate the interactive creation of a shading program to specify how to compose the light fields. We utilized this interactive method as well as interactive binding of keystrokes to quickly produce all scenes in this paper.

One more assumption is that the programmer writes a ray-shading program that is view-dependent. When compositing light fields, the programmer must specify a compositing order. If there is a visibility change, then this order may change. An advanced ray-shading program can handle visibility changes by implementing a per-pixel sort of the light fields. In our Vega Strike implementation, this hap-

[3]The billboard appears 3D since a 3D object appears to be inside it, but in fact the light field representation is in general 4D.

(a) (b) (c)

Figure 6: (a) Conventional focusing in a light field (10 MB in size). The front swimmer lies on the focal plane. (b) A multi-focal plane image where the front and back swimmers are brought into focus for emphasis. The middle swimmer and the crowd are defocused. (c) The front swimmer is defocused, but a large depth of field exists over the depths of the middle and back swimmer. There is a sharp transition in focus between the front and mid swimmers. Light fields captured by Ren Ng.



(a) (b) (c)

Figure 7: Screenshots from Vega Strike with a LightShop plugin. The red spaceship is an acquired light field of a physical toy, and all other objects in the screenshots are 3D polygonal models or sprites. Notice that the light field is integrated seamlessly into the surrounding 3D graphics.

pens implicitly by activating the Z-buffer when drawing the light field individually on their separate billboards.

There are a number of interesting avenues of research open for development in LightShop. For instance, LightShop currently does not support the animated or skinned light fields necessary for inserting humans into games. One option would be to utilize Flowed reflection fields introduced by Einarsson et al.[2006] for human locomotion. Another option would be to capture a light field of a motionless human and associate it with a bone structure in order to apply skinning techniques to the light field. Both ideas would fit within the LightShop API and would allow games to capture actors using cameras instead of time consuming modeling.

Another interesting context for future work could be medical imaging. For example our warping operator could be used to describe the aberrometry data of a human eye, produced from a Shack-Hartmann device [Platt and Shack 1971; Barsky et al. 2002]. The warped rays would sample from an acquired light field, presenting photorealistic interactive scenes as seen through a human optical system.

LightShop provides a powerful mechanism for manipulating and rendering light fields. The GLSL implementation runs in real-time and provides a complete interface for a game programmer to access light fields in the fragment shader. Future additions to LightShop might make use of more of the OpenGL rendering state to manipulate light fields (i.e. using light information to shade light fields that have normals information). Such additions will help to integrate light fields into the graphics rendering pipeline.

9 Acknowledgements

We would like to thank Marc Levoy for his countless insights and guidance. Hendrik Lensch and Gernot Ziegler also provided invaluable advice during the initial stages of this project. We would also like to thank the reviewers for their advice for revising the paper. Additionally we would like to thank Ren Ng for his light field action photography. This work is supported by the US Department of Energy (contract B554874-2) and the 2006 ATI Fellowship Program.

References

Adobe. 2000. Adobe photoshop version 6.0 user guide.

Akenine-Möller, T., and Haines, E. 2002. *Real-time Rendering*. A K Peters.

Barsky, B. A., Bargteil, A. W., Garcia, D. D., and Klein, S. 2002. Introducing vision-realistic rendering. In *Eurographics Rendering Workshop (Poster)*.

Barsky, B. A. 2004. Vision-realistic rendering: simulation of the scanned foveal image from wavefront data of human subjects. In *Proceedings of Symposium on Applied Perception in Graphics and Visualization (APGV)*, ACM Press, New York, NY, USA, 73–81.

Board, O. A. R., Shreiner, D., Woo, M., Neider, J., and Davis, T. 2005. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley.

Camahort, E., Lerios, A., and Fussell, D. 1998. Uniformly sampled light fields. In *Proceedings of Eurographics Rendering Workshop*, 117–130.

Chen, W.-C., Bouguet, J.-Y., Chu, M. H., and Grzeszczuk, R. 2002. Light field mapping: Efficient representation and hardware rendering of surface light fields. In *Proceedings of SIGGRAPH*.

Chen, B., Ofek, E., Shum, H.-Y., and Levoy, M. 2005. Interactive deformation of light fields. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 139–146.

Chen, B. 2006. *Novel Methods for Manipulating and Combining Light Fields*. PhD thesis, Stanford University.

Chuang, Y.-Y., Curless, B., Salesin, D. H., and Szeliski, R. 2001. A bayesian approach to digital matting. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, vol. 2, 264–271.

Einarsson, P., Chabert, C.-F., Jones, A., Ma, W.-C., Lamond, B., Hawkins, T., Bolas, B., Sylwan, S., and Debevec, P. 2006. Relighting human locomotion with flowed reflectance fields. In *Proceedings of Eurographics Symposium on Rendering*.

Gonzalez, R. C., and Woods, R. E. 2002. *Digital Image Processing (2nd Edition)*. Prentice Hall.

Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F. 1996. The lumigraph. In *Proceedings of SIGGRAPH*, 43–54.

Heidrich, W., Lensch, H., Cohen, M. F., and Seidel, H.-P. 1999. Light field techniques for reflections and refractions. In *Proceedings of Eurographics Rendering Workshop*, 187–196.

Horn, D., Sampson, J., Horn, P., Fierre, C., Aleksandrow, D., Borri, M., Lloyd, B., and Griffin, P. 2006. Vega strike open source space simulator engine. *http://vegastrike.sourceforge.net*.

Iourcha, K., Nayak, K., and Hong, Z. 1999. System and method for fixed-rate block-based image compression with inferred pixel values. *US Patent 5,956,431*.

Levoy, M., and Hanrahan, P. 1996. Light field rendering. In *Proceedings of SIGGRAPH*, 31–42.

Levoy, M. 2004. The stanford large statue scanner. *http://graphics.stanford.edu/projects/mich/*.

McReynolds, T., Blythe, D., Grantham, B., and Nelson, S. 1998. Programming with opengl: Advanced techniques. In *Course 17 notes at SIGGRAPH 98*.

Meneveaux, D., Subrenat, G., and Fournier, A. 2002. Reshading lightfields. Tech. rep., IRCOM/SIC, March. No 2002-01.

Ng, R., Levoy, M., Brédif, M., Duval, G., Horowitz, M., and Hanrahan, P. 2005. Light field photography with a hand-held plenoptic camera. Tech. rep., Stanford University.

Oh, B. M., Chen, M., Dorsey, J., and Durand, F. 2001. Image-based modeling and photo editing. In *Proceedings of SIGGRAPH*.

Perlin, K. 1985. An image synthesizer. In *Proceedings of SIGGRAPH*, ACM Press / ACM SIGGRAPH, 287–296.

Platt, B., and Shack, R. 1971. Lenticular hartmann-screen. In *Optical Science Center*.

Porter, T., and Duff, T. 1984. Compositing digital images. In *Proceedings of Computer Graphics*, 253–259.

Seitz, S., and Kutulakos, K. N. 1998. Plenoptic image editing. In *Proceedings of International Conference on Computer Vision (ICCV)*, 17–24.

Shum, H.-Y., and Sun, J. 2004. Pop-up light field: An interactive image-based modeling and rendering system. In *Proceedings of Transactions on Graphics*, 143–162.

Smith, A. R., and Blinn, J. F. 1996. Blue screen matting. In *Proceedings of Computer Graphics and Interactive Techniques*, 259–268.

Sweeney, T., and Epic, 2006. Unreal engine 3.

Upstill, S. 1992. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley.

Vaish, V., Wilburn, B., Joshi, N., and Levoy, M. 2004. Using plane + parallax for calibrating dense camera arrays. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, 2–9.

Vlasic, D., Pfister, H., Molinov, S., Grzeszczuk, R., and Matusik, W. 2003. Opacity light fields: interactive rendering of surface light fields with view-dependent opacity. In *Proceedings of Symposium on Interactive 3D Graphics (I3D)*, ACM Press, New York, NY, USA, 65–74.

Wilburn, B., Joshi, N., Vaish, V., Talvala, E.-V., Antunez, E., Barth, A., Adams, A., Horowitz, M., and Levoy, M. 2005. High performance imaging using large camera arrays. In *Proceedings of Transactions on Graphics (SIGGRAPH)*, 765–776.

Yu, J., Yang, J., and McMillan, L. 2005. Real-time reflection mapping with parallax. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 133–138.

Zhang, Z., Wang, L., Guo, B., and Shum, H.-Y. 2002. Feature-based light field morphing. In *Proceedings of Transactions on Graphics (SIGGRAPH)*, 457–464.