A Generalized Light-Field API and Management System

Joan Blasco
Dto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera s/n
46022 Valencia, Spain
joablaib@inf.upv.es

Miguel Escrivà
Dto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera s/n
46022 Valencia, Spain
mescriva@dsic.upv.es

Francisco Abad
Dto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera s/n
46022 Valencia, Spain
fjabad@dsic.upv.es

Ricardo Quirós
Dto. Lenguajes y Sistemas Informáticos
Universidad Jaime I
Avenida Sos Banyat s/n
28071 Castellón, Spain
quiros@lsi.uji.es

Emilio Camahort
Dto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera s/n
46022 Valencia, Spain
camahort@dsic.upv.es

Roberto Vivó
Dto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera s/n
46022 Valencia, Spain
rvivo@dsic.upv.es

ABSTRACT

Light fields are a computer graphics modeling technique that represents objects using radiance samples instead of geometry. Radiance samples may be stored as sets of images or 4D arrays of pixel values. Light fields have various advantages: their rendering complexity only depends on the output image's, they can represent sophisticated illumination effects, and they are well-suited for display using autostereoscopic devices.

To study different light-field representations, as well as their capture, generation, resampling, storage, composition, rendering and display, we have developed a light-field based API and graphics system. The system supports models based on different light-field parameterizations, as well as different generation and capture methods, rendering methods, data caching algorithms and output displays. The API will also support hybrid light-field and geometry based models, as well as light-field resampling and composition.

Keywords: Image-based models, the light field, graphics APIs, modeling and rendering systems.

1 INTRODUCTION

Advances in imaging and graphics hardware in the past ten years have motivated the appearance of novel rendering techniques based on image data. Image-based rendering methods build up a 3D representation of an object using radiance data obtained from 2D image samples of the object. These methods can synthesize non-existent image samples from the information of the existent ones, granting the observer a perspective-correct view from either sampled or unsampled viewpoints.

Image-based rendering techniques offer simple acquisition capabilities coupled with realistic representation of complex lighting conditions. Furthermore, in contrast to geometry based techniques, purely image-based models provide two additional advantages. First, rendering complexity of image-based models depends only on the complexity of the output images. Second, most compression and simplification algorithms perform better on image data than geometric data. Additionally, both geometry-based models and image-based models can be merged to produce hybrid models using information originating from both geometry and images.

Light fields are a Computer Graphics modelling technique that relies on radiance or image data to represent complex geometric objects. These objects are inefficient to store and render using geometry, but radiance data can be organized in different arrangements of images or 4D pixel arrays to represent and display the objects [Lev06].nswered 58 12/19/2007 Roberto V

We introduce a light-field modelling system to efficiently build, store, combine, resample, retrieve and render light fields based on different representations. Our goal is to compare the representations, combine them, and display them on autostereoscopic displays [DEA$^+$06].

We want to achieve interactive rendering framerates together with an intuitive API for light-field capture and generation. This paper describes the design of our light-field representations, our API and the architecture of the underlying graphics system. We also report work in progress in multiple light-field rendering and light-field models augmented with geometry.

Light fields can be applied to many computer graphics related areas. These include rendering, scientific visualization, advertisement and virtual reality applications. Our main interest is driving autostereoscopic displays and other volumetric displays. These displays typically output 3D or 4D light-field data. They are the future of graphics devices because they provide multiple users with correct perspectives for both eyes without using intrusive devices, like goggles.

Our paper starts with a review of previous work in light-field modelling and rendering systems. Then, we formalize light-field representations and review light-field parameterizations. We introduce then the functionality of our system and describe thoroughly its goals. Section 4 describes our API, Section 5 our implementation, and Section 5.6 our results. Finally, we present some conclusions and directions for future work.

2 BACKGROUND

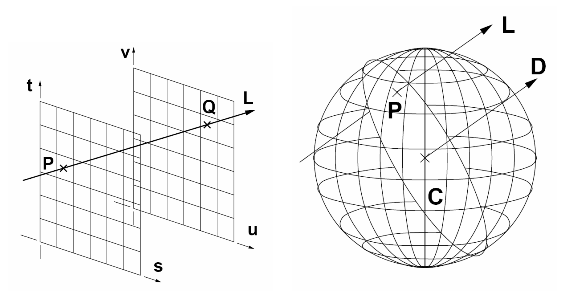
Among the different image-based rendering techniques, light fields are one of the newest and most complex ones. Light fields were introduced in the papers by Levoy and Hanrahan [LH96] and Gortler et al. [GGSC96], where they were called lumigraphs.

Formally, the light field or plenoptic function is defined as a 5D function that describes the radiance flowing through all the points in a scene in all possible directions [AB91]. A static light field can be represented as a 5D scalar function that can be simplified to 4D in the absence of occluders, as shown in [GGSC96, LH96].

In order to render a light field, the radiance function is sampled at points positioned on a sampling pattern, determined by the kind of parameterization used in the model. There are planar anisotropic [LH96] and spherical quasi-isotropic [CLF98] parameterizations. They result in different parameter arrangements and light-field configurations. Additionally, unstructured light-fiel models can also be built [Tak05].

The most important examples of planar parameterizations are Levoy and Hanrahan's "light slab" fields [LH96] and the Lumigraph [GGSC96]. Both use a 2 plane parameterization. They select a planar grid of camera positions and take an image for each camera. They require multiple pairs of planes, called slabs, to represent entire objects.

Schirmacher [SHS99] uses the same representation, but adds a simple geometric model (called proxy) similar to the lumigraph's. The entire light-field model is then built using an adaptive sampling method starts with a simple set of views, then attempts to add a new image from a different view. In order to determine which point of view to use for the new image, several candidates are considered. Candidate views are then prioritized using a cost function that takes into account disocclusion artifacts and a radiance error measure.



(a) Two-plane parameterizations. (b) Direction-and-point parameterizations.

Figure 1: Two related light-field parameterizations.

In the case of spherical arrangements, cameras are positioned throughout a sphere's surface, pointing towards its center. Images are thus sampled from the sphere's surface as described in [CLF98]. This way the sphere acts as a simple approximation to the convex hull of an object centered inside the sphere.

Using this approach, camera positions are determined by the sampling directions needed, and sampling direction distribution is determined by the specific set of directions selected. Examples of these arrangements are those used in [CLF98, IPL97], where the chosen distribution of the samples follows quasi-uniform triangulations of the sphere's surface [EDA$^+$06].

In unstructured light fields, however, camera positions need not to be positioned following any specific arrangement. Among these methods, it is possible to make a difference between those in which correct renderings are based on camera positions but also on a priori knowledge about geometry information from the scene, and those that use other approaches.

For example in [Tak05], where an algorithm for light-field rendering that does not make use of any geometry information is presented. The technique is based on a 3-step algorithm. In this algorithm a series of depth layers are created and pixels from the captured images are positioned on those layers according to an estimation of their depth based on an *in-focus* measurement.

In the case of geometric information support, the use of geometric proxies to enhance object representations has been researched since the birth of light-field modeling [GGSC96, PPS, CBCG02, WAA$^+$00, BBM$^+$01] and greatly resembles view-dependent texture mapping[DTM96] in the use of transformed captured images as textures for the geometric proxies.

For instance, in [BBM$^+$01] the authors use approximate geometry models to build a map containing the weight of the images obtained from each position for every pixel. A subsequent sampling of the image plane uses these weights to render images from cameras with non-zero weights at every sample point used in the representation. Each sample is rendered m times, being m

the number of cameras. These samples are also taken based on geometry information, and are determined by projecting the geometric approximation of the scene on the image plane.

We want to support these different types of light-field representations: planar, spherical and unstructured. Our work thus builds on the results of [LH96], [GGSC96], [?], [SHS99] and [Tak05] among others. But our focus is on the systems and implementation issues of building, managing and rendering multiple types of light-field models. In this paper we focus on a management system for general light fields and the associated API to access the functionality (features) of our system.

3 FUNCTIONALITY

Providing support for light-field modelling and rendering requires a fairly sophisticated hardware and software system. We may need to support different models and camera arrangements, different robot configurations for image capture, several storage strategies, and multiple rendering algorithms and display devices. Additionally, we may want to compare different sampling, resampling, composition and compression algorithms. In this Section we describe which of these features we want to support in our light-field system.

3.1 Light-Field Representations

We want a system capable of drawing and capturing different light-field representations. With this in mind, we have designed software that can be easily configured and extended to support different types of light-field representations. The flexibility of our API also supports models that include geometric information like, for example, depth information. Our architecture supports adding new representations by configuring light-field plugins in a simple way.

We have developed a plugin based on spherical light fields. It is based on the direction-and-point parameterization [EDA$^+$06]. A planar light-field plugin is also being currently developed. This added functionality, is useful to combine and resample different types of light fields.

3.2 Generation and Capture

Light-field models can not exist without generating or capturing them. Generation produces models of synthetic object, while capture samples radiance data from real-world objects. Light-field generation is performed using methods that are specific to each type of representation. Typically, the methods are camera iterators that produce the camera parameters needed to properly obtain the images of the light-field model. To generate synthetic models, we support rendering using OpenGL, Blender, POV or YafRay.

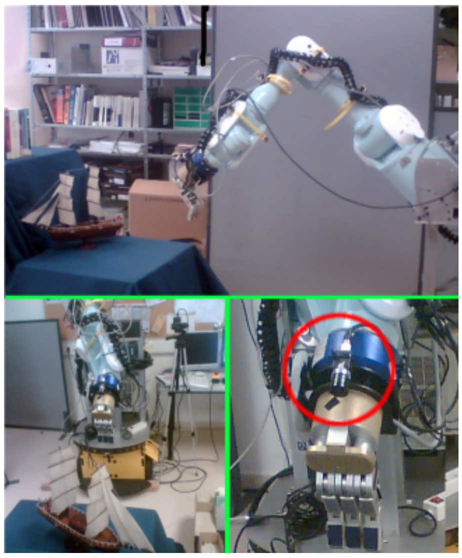


Figure 2: The robot arm is controlled by a PC. The arm is moved to different locations and takes snapshots of the object to capture.

Our software also allows the capture of light fields from real-world objects providing a list of camera parameters for each of the images needed. This camera parameters are also produced by iterators and they allow us to create snapshots from the real world objects. This capture of real-object light-fields is performed using a camera mounted on a robotic arm(see figure 2).

3.3 Spherical Light-Field Plugin

Light fields have a major drawback. Obtaining high-quality renderings needs a huge amount of images, up to 20000, which uncompressed result in approximately 3.7 GBytes of storage. Use of fewer images produces artifacts due to discretization errors. These errors produce seams visible across triangle boundaries when rendering is done without interpolation (this kind of artifact can be noticed in figure 11). Use of interpolation however turns this seams into ghosting (see figure 3(b)).

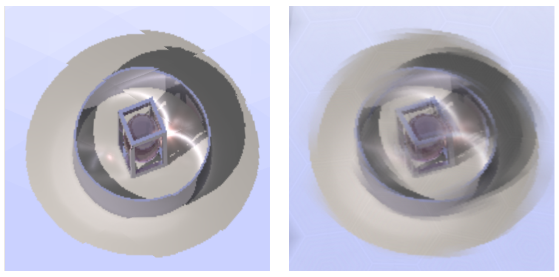

(a) With constant kernel. (b) With quadralinear kernel.

Figure 3: A light-field model with caustics. The model was rendered using two algorithms with different interpolation kernels. .

An alternative to higher-resolution light fields is the use of depth information and lower directional resolutions. Gortler et al. implemented depth-corrected light-field rendering using a representation extended with a low-resolution geometric proxy of the target object [GGSC96]. Shade et al. introduced layered depth images, images extended with a depth value associated to each pixel [SGHS98].

Using depth information rendering can be improved to compute accurate texture coordinates. With this technique the amount of storage required by a light field's radiance data can be reduced by a factor of $1/4$ or $1/16$.

3.4 Light-field Composition

Another of our goals is the composition of different light-field models and addition to external geometric information such as labels, tags and regular objects into a single scene. This goal can be accomplished with the use of depth maps, multiple images per directional sample, and improved rendering algorithms.

Handling of multiple light fields is managed by storing multiple images per directional sample. Images are sorted according to their position in 3D space along each direction and then multiple images for each visible triangle are reprojected. Drawing of these images in proper back-to-front order must be ensured for obtaining of correct renderings.

Geometry and light-field rendering can also be accomplished using depth information associated to the light-field data. Again, texture-mapped triangles are generated for each directional sample and each light-field model. They are drawn in 3D space at roughly the same location as the surface of the light fields' original geometric objects. Geometric objects are drawn subsequently.

4 API DESIGN

Light-field capture and generation have been simplified to provide an intuitive programmatical interface in our light-field software. Acquisition can be accomplished by following four simple steps. First, the light-field parameters are initialized. Second, a list of the camera positions, which may vary depending on the parameterization plugin used, is provided. Images from the different positions are then obtained sequentially, and once all of them have been processed, the light field is finally stored. The whole process is depicted in figure 4.

The rendering process has been simplified in a similar way. In this case, after the initialization of the light-field plugin of choice, the user needs to select and load the algorithm responsible for its rendering, and assign its concerning properties in case they exist. After that, the renderer and the image cache need to be created before starting the actual rendering.

Our software uses multiple threads, so that rendering and image caching can run separately (see figure 5).

Control of the image retrieval thread is done by the rendering thread, which decides which images to transfer depending on the time available for each frame. The

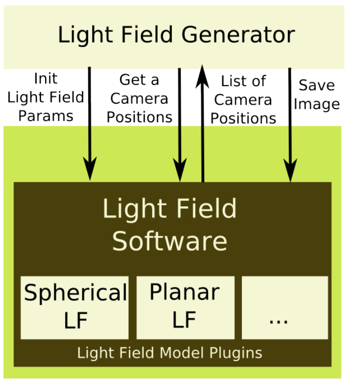


Figure 4: Capture system architecture.

rendering thread also decides the size of the working set based on the number of light-field images that the GPU can display in 1/25 seconds.

The whole set of light-field data is difficult to fit into main memory concurrently, so we use out-of-core storage techniques. Our system uses a texture least-recently-used (LRU) cache that manages the light-field images so that only new images need to be loaded from disk at any time.

Using a two-level cache architecture we support rendering of light-field models requiring up to several gigabytes of storage. The architecture transfers image data from secondary storage to main memory, and from main memory to GPU memory. The size of a light-field model requires storing it in a hard drive, but loading the image data from a hard drive produces really poor performance, so we need a faster way of loading the images. The fastest way stores the images in GPU memory and loads them directly when needed for a visible triangle. However, GPU memory is not enough to hold all the image data, so we have added an intermediate cache system in main memory to provide a better capacity/response time ratio as seen in figure 8)

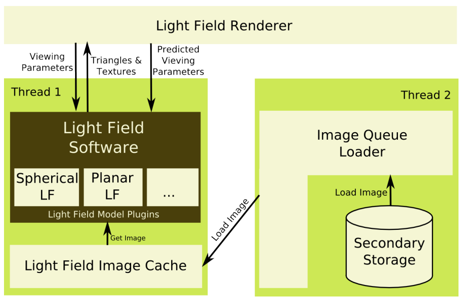


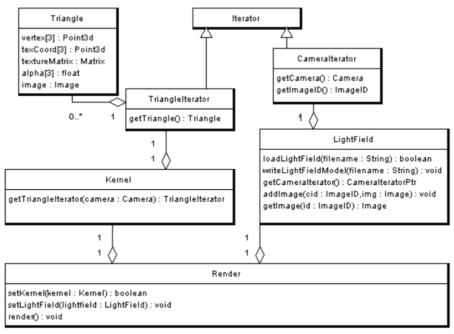
Figure 5: Render system architecture.

5 IMPLEMENTATION

In this section, we discuss the implementation of the above features. For the implementation we use C++, because it provides the expressiveness and performance needed for both the image processing and the rendering of the light-field data. We start with an overview of our class architecture.

5.1 Light-Field Interface

A light-field class represents an abstract interface to all of the different light-field representations. It is in charge of managing the light-field model and also possesses knowledge about the organization of the images stored in secondary storage.



Triangle
vertex[3] : Point3d
texCoord[3] : Point3d
textureMatrix : Matrix
alpha[3] : float
image : Image
Iterator
CameraIterator
getCamera() : Camera
getImageID() : ImageID
TriangleIterator
getTriangle() : Triangle
0..*
1
1
1
1
LightField
loadLightField(filename : String) : boolean
writeLightFieldModel(filename : String) : void
getCameraIterator() : CameraIteratorPtr
addImage(cid : ImageID,img : Image) : void
getImage(id : ImageID) : Image
Kernel
getTriangleIterator(camera : Camera) : TriangleIterator
1
1
1
1
Render
setKernel(kernel : Kernel) : boolean
setLightField(lightfield : LightField) : void
render() : void

Figure 6: Light–Field and Kernel interface diagram

The light-field class knows how to load, save and generate its light-field representation, but knows nothing about the rendering of the light-field (see figure 6). The class responsible for the task of rendering the light field is the *Kernel* class.

5.2 Kernel Interface

A light-field parameterization can be rendered using different algorithms providing different results (see figure 15 for an example). Control of the different algorithms used in the rendering stage has been modelled with the use of *Kernels*. In this system, a *Kernel* is the class in charge of the rendering of a light-field type (see figure 6).

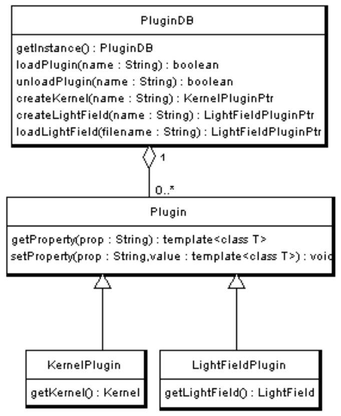
A light-field representation is able to use different kernels to render the light field. As an example of this fact, rendering of a direction-and-point parameterization (DPP) based light-field can be performed by using various kernels. At the present moment these consist of a constant reconstruction kernel, a quadratic reconstruction kernel and a kernel using depth-maps for geometric information supported rendering.

Examples of the use of such kernels with the DPP light-field plugin can be observed in figures 3(a), 3(b) and 15. The different results obtained with the use of the various kernels are clearly visible.

5.3 Plugin Architecture

Our system uses a plugin architecture in order to support multiple light-field parameterizations. In this way, different light-field parameterizations can be implemented in plugins that the *base* library can load on demand.

We support two kinds of plugins: LightFields and Kernel modules (see figure 7). Our plugin system allows us to check which plugins are registered in the system and inspect which kind of plugin the loaded plugins represent, as well as which properties the plugins support. All of this processing can be performed in execution time.



PluginDB
getInstance() : PluginDB
loadPlugin(name : String) : boolean
unloadPlugin(name : String) : boolean
createKernel(name : String) : KernelPluginPtr
createLightField(name : String) : LightFieldPluginPtr
loadLightField(filename : String) : LightFieldPluginPtr
1
0..*
Plugin
getProperty(prop : String) : template<class T>
setProperty(prop : String,value : template<class T>) : voi
KernelPlugin
getKernel() : Kernel
LightFieldPlugin
getLightField() : LightField

Figure 7: Classes that implement the plugin functionality.

Each LightField or Kernel may have different properties to configure. For example, the DPP LightField implementation has a property related to the level of the icosahedron subdivision, but PDP LightField has a property related to the number of horizontal and vertical plane subdivisions. This properties can be accessed directly by their name, and they are fully configurable and independent of the rest of the architecture.

5.4 Cache Architecture

As it was said in section 4, our system uses a two-level cache architecture, that allows us to render light-field models requiring a huge storage capacity.

As new images are needed by the renderer, the system caches the corresponding images, first obtaining them from disk and fitting them into main memory (when they are not directly available in there) and then from main memory into GPU memory (see figure 8).

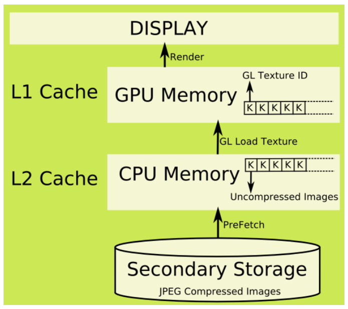


Figure 8: Two-level cache architecture: images are addressed using unique triangle identifiers (CPU memory and secondary storage) and texture identifiers (GPU memory).

The caching algorithm has also been designed to steer image retrieval following camera movement. During rendering, the most probable new camera position and orientation is computed using the last camera movement. This prediction provides an estimate of the next frame's viewing frustum. The frustum is then used to determine the set of images that are most likely to be needed soon.

5.5 Implementing a new Light-field type

In order to add a new light field, a series of steps are to be performed. First of all, a new class inheriting from the *LightField* interface and implementing its methods must be created. Class *DPP_LightField* is an example implementation, as figure 9 shows.

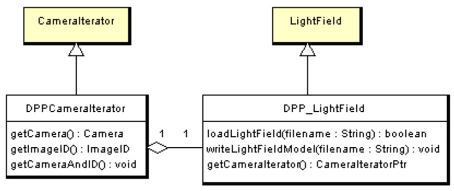

CameraIterator
LightField
DPPCameraIterator
getCamera() : Camera
getImageID() : ImageID
getCameraAndID() : void
1
1
DPP_LightField
loadLightField(filename : String) : boolean
writeLightFieldModel(filename : String) : void
getCameraIterator() : CameraIteratorPtr

Figure 9: DPP light-field implementation diagram

The most important methods needed to implement in this class are *loadLightField*, *writeLightFieldModel* and *getCameraIterator*. The *loadLightField* method is responsible for the setup of the light-field information, while the *writeLightFieldModel* method will be in charge of the information dump to a massive storage device at the end of the acquisition process. Finally, *getCameraIterator* is the most important method regarding the obtaining of the camera position arrangements of the plugin in use (see figure 9).

Once the light-field class for the new model has been implemented, the user must adapt, if needed, a new *CameraIterator* for it. This iterator will provide the new LightField class with the desired camera arrangement. At this stage, the system is already capable of acquiring light fields based on the new model, see figure 9.

In order to render the acquired light fields, the user must also implement the rendering algorithm to do so. This is accomplished by implementing one or more adapted classes inheriting the *Kernel* interface. This class will be the placeholder of the rendering algorithm, and will work jointly with a corresponding triangle iterator, that should be implemented too if needed. An example of this class architecture is shown in figure 10.

5.6 Results

The DPP light-field plugin for our system has been tested on several datasets. Figures 11, 12 and 13 show renderings from different models at different levels of detail. The models used were large polygon meshes, with polygon counts ranging from 226k to 478k polygons. In figure 14, complex lighting effects like caustics

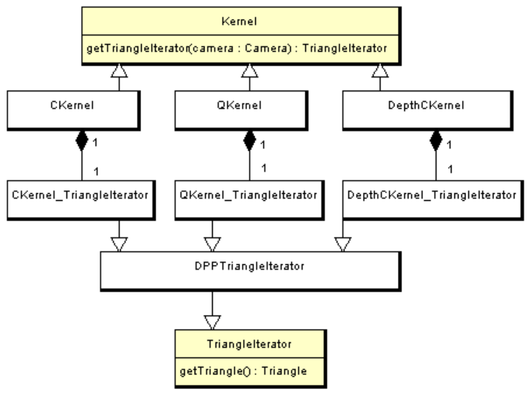


Figure 10: Kernel module architecture example

were also added to the scene. All of these light fields were obtained using the Blender and Yafray renderers.

Rendering of the original geometric models took 60 seconds for Models 1 and 2, 90 seconds for Model 3 and roughly 3.5 minutes for Model 4. The light-field models, however, were rendered at 50–60 frames per second using the constant and quadralinear kernels, and two seconds per frame using depth correction. This greatly improves on the rendering times of the geometric models.

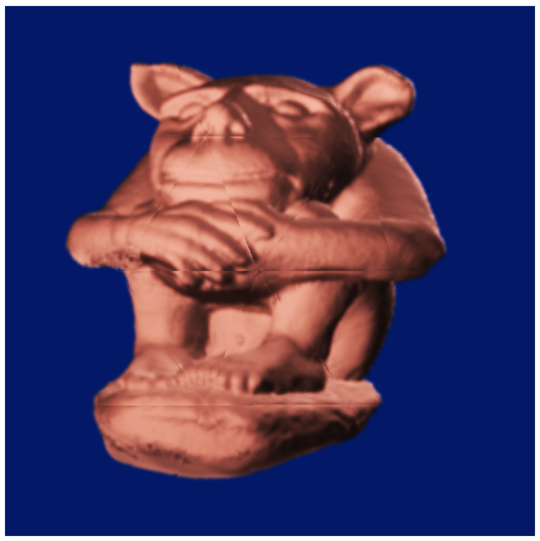


Figure 11: Model 1: The UMA model was scanned with our 3D scanner. We captured the light field using Blender. The image shows the render of model at level 4.

In figure 15 two different renderings of the Stanford dragon model can be observed, showing the differences in image quality obtained when depth information is used for the rendering of the light fields. In this case the light field was obtained using OpenGL.

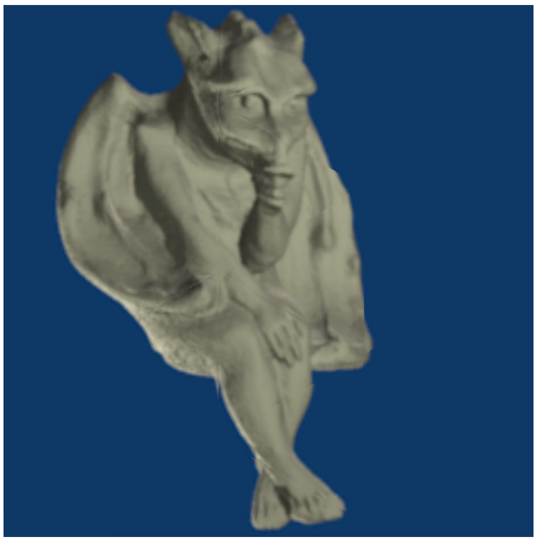


Figure 12: Model 2: A gargoyle model rendered al level 4. The light field was captured using Blender and the model geometry was acquired with our 3D scanner.

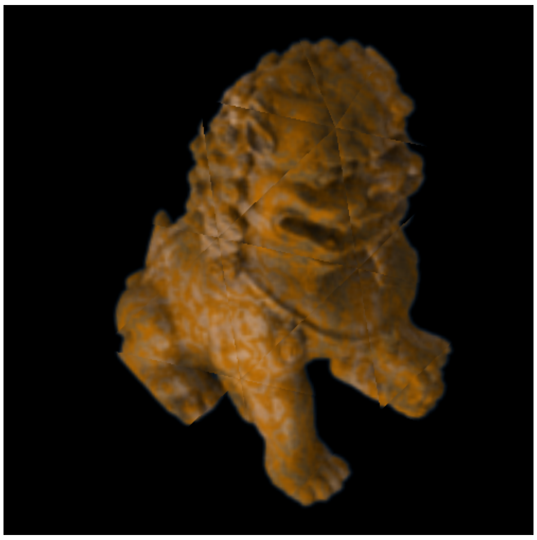


Figure 13: Model 3: Chinese lion light field rendered at level 3. Model is provided courtesy of INRIA by the AIM@SHAPE Shape Repository.

6 CONCLUSIONS AND FUTURE WORK

In this paper we address the issue of light-field modeling and rendering in a generalized way, through the creation of a system which allows abstraction from the different features that different light-field parameterizations have.

We presented a system that creates an abstract framework for light-field modelling and rendering, allowing a quicker development of novel techniques of this kind, which may use different parameterizations, or be supported by information coming from varied sources like radiance, depth or others. The system has been implemented in standard C++, also granting its flexibility and portability.

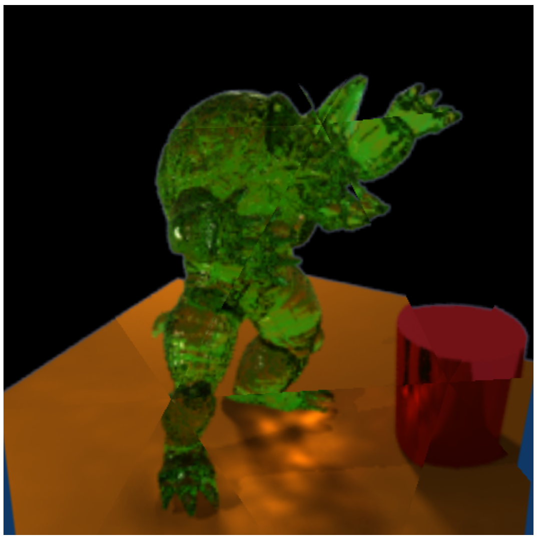


Figure 14: Model 4: The Stanford armadillo model with a green glass material. The light field was captured with YafRay raytracer to obtain high quality images with effects like caustics and reflections. Rendered at level 2.

Current work includes creation of a spherical light field plugin for the system, while implementation of additional plugins like a planar light field plugin are still under development.

The reader is referred to the webpage `http://www.sig.upv.es/ALF/papers/wscg2008/` for some videos produced with our system.

ACKNOWLEDGEMENTS

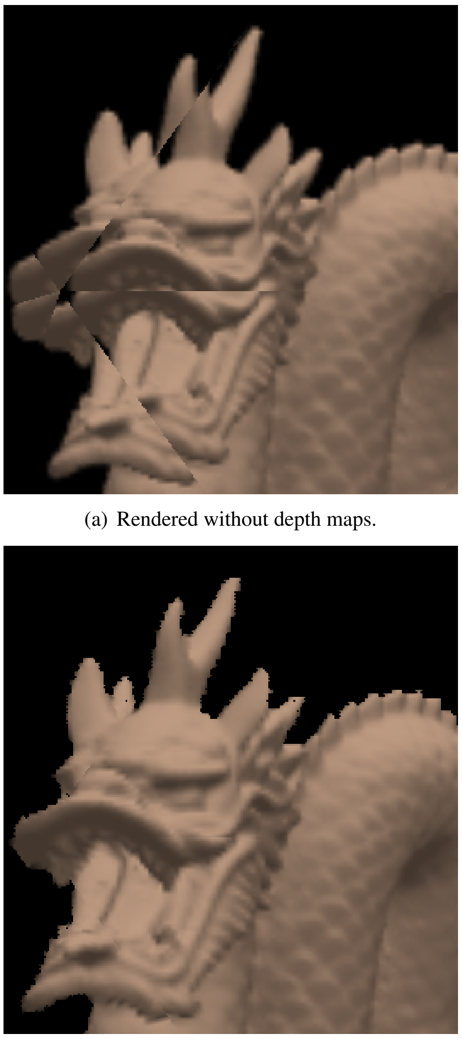
This work was partially supported by grants TIN2004-203303-E and TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science and by the GAME-TOOLS STREP project IST-004363 of the 6th Framework Program of the European Union.

REFERENCES

[AB91] E. H. Adelson and J. R. Bergen. The plenoptic function and the elements of early vision. In *In Computational Models of Visual Processing*, page Chapter 1, Cambridge, MA, 1991. MIT Press.

[BBM+01] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 425–432, New York, NY, USA, 2001. ACM Press.

[CBCG02] Wei-Chao Chen, Jean-Yves Bouguet, Michael H. Chu, and Radek Grzeszczuk.


(a) Rendered without depth maps.

(b) Rendered using the depth maps.

Figure 15: The Stanford dragon model. This figure shows how the use of depth maps improves the output image quality.

Light field mapping: efficient representation and hardware rendering of surface light fields. *ACM Trans. Graph.*, 21(3):447–456, 2002.

[CLF98] Emilio Camahort, Apostolos Lerios, and Donald Fussell. Uniformly sampled light fields. In *Proc. Eurographics Rendering Workshop '98)*, pages 117–130, 1998.

[DEA$^+$06] A. Domingo, M. Escrivá, F.J. Abad, R.Vivó, and E.Camahort. Introducing extended and augmented light fields for autostereoscopic displays. In *In Proceedings of 3rd Ibero-American Symposium in Computer Graphics*, pages 64–67, 2006.

[DTM96] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 11–20, New York, NY, USA, 1996. ACM Press.

[EDA$^+$06] M. Escrivá, A. Domingo, F. Abad, R. Vivó, and E. Camahort. Modeling and rendering of dpp-based light fields. In *In Proceedings of GMAI'2006*, London, UK, 2006. IEEE Computer Society Press.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proc. SIGGRAPH '96*, pages 43–54, 1996.

[IPL97] Insung Ihm, Sanghoon Park, and Rae Kyoung Lee. Rendering of spherical light fields. In *PG '97: Proceedings of the 5th Pacific Conference on Computer Graphics and Applications*, page 59, Washington, DC, USA, 1997. IEEE Computer Society.

[Lev06] Marc Levoy. Light fields and computational imaging. *Computer*, 39(8):46–55, 2006.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proc. SIGGRAPH '96*, pages 31–42, 1996.

[PPS] Steven J. Gortler Peter-Pike Sloan, Michael F. Cohen. Time critical lumigraph rendering. In *In Proc. 1997 Symposium on Interactive 3D Graphics*.

[SGHS98] Jonathan W. Shade, Steven J. Gortler, Li-Wei He, and Richard Szeliski. Layered depth images. In *Proceedings of ACM SIGGRAPH'98*, pages 231–242, July 1998.

[SHS99] H. Schirmacher, W. Heidrich, and H.-P. Seidel. Adaptive acquisition of lumigraphs from synthetic scenes. In *In EUROGRAPHICS'99*, pages 151–159, 1999.

[Tak05] T. Takahashi, K.; Naemura. Unstructured light field rendering using on-the-fly focus measurement. *IEEE International Conference on Multimedia and Expo*, 2005.

[WAA$^+$00] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface light fields for 3D photography. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 287–296. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.