# A Flexible Rendering Architecture for General Linear Cameras and other Multi-perspective Images

Augusto Román (aroman@stanford.edu)

CS448 Final Project

Spring 2004

## 1 Objective

The goal of this project is to implement a flexible rendering architecture that allows rendering any of the General Linear Cameras described in Yu & McMillan's ECCV 2004 [1] paper with the same title. The implementation should not be dependent upon specific input or output formats, favoring flexibility and extensibility over speed and efficiency.

## 2 Introduction

Multi-perspective images have many uses for visualizing large datasets, for example full city street blocks or even an entire city at once. However, there is not yet a standard paradigm for specifying such multi-perspective images that corresponds to the single-point perspective image framework. As such, rendering such images becomes a specialized task suited only to a single application at a time.

Such a standard framework is suggested and a basic implementation is presented in this project. As a demonstration of its ease and flexibility, the General Linear Camera framework described by Yu & McMillan is implemented and several examples shown. Synthetic aperture interpolation is also included to demonstrate the independent modular nature of the framework.

## 3 Images

An image can be considered simply a collection of rays in space that are displayed on a 2D grid. Generally the set of rays in an image have a structure to them such that the displayed result is meaningful to people. For example, an image from a typical camera can be described as having a perspective structure. This means that the set of rays that are displayed all pass through a single point, the center of projection. Furthermore, a subset of these rays is sampled for the final displayed output. Typically, this selection is defined by the intersection of a plane (the image plane) with the rays.

The General Linear Cameras described in [1] represent a framework for compactly defining the arrangement of rays in space. Specifically, their framework can describe any linear 2D subspace of the 4D ray space.

We can decompose these images into two relevant components. First, the *ray directions* describe the arrangement of rays in space. Second, the *image surface* defines the sampling of those rays for the final displayed output.

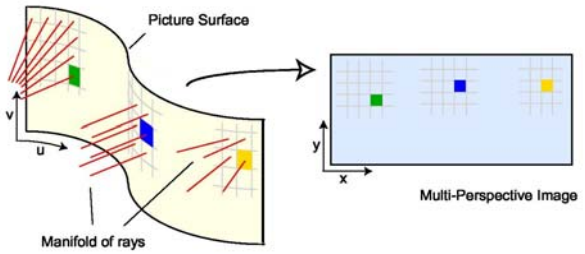


**Figure 1:** The final multi-perspective output image $I(x,y)$ can be represented by a combination of a picture surface $S(u,v)$ that samples the arrangement (or manifold) of rays $D(\Re^3)$.

Thus, an image can be defined with the following components as shown in Figure 1:

- $I(x,y) \in RGB$ with
  $x \in [0, \text{width}], y \in [0, \text{height}]$
  The final displayed output image, with $x$ and $y$ denoting the integer pixel coordinates. Typically the image displays RGB colors, but it can also describe opacity (alpha), radiance, wavelength, etc.
- $D(\Re^3) \in \Re^2$
  The ray directions vector field defines a direction for every point in space.
- $S(u,v) \in \Re^3$ with $u,v \in [0,1]$
  The image surface defines the spatial manifold over which the ray directions are sampled. The normalized coordinates $u$ and $v$ define the position in the final output image as well as the extent in $\Re^3$.

## 4 Core Architecture

The components of the software mimic the mathematical components of an image described above:

1. *Region*: Associates the entire image surface with a region of the final displayed output.
2. *Image Surface*: Defines the image surface, both the manifold and the sampling along it.
3. *Ray Selector*: Defines the ray directions in space.
4. *Ray Sampler*: Determines the color of a particular ray. This object encapsulates the dataset and any interpolation algorithms.
5. *Renderer*: The overall rendering engine that uses the above modules to render the image.

Described in more detail below, the middle three components are pure virtual abstract C++ base classes – that is, they define only the interface, allowing any implementation. Several basic implementations have also been provided as described in Section 5 below.

### 4.1 Region Description

The region description is an intermediate structure that simply associates the image surface with a region of the final output image. This allows multiple, independent views to be rendered onto a single output image, for example the *students* animation where the left side shows real image data while the right shows synthetic data. Thus, the region description object defines the set of pixels to render in the final output image and takes care of converting them to normalized *uv* coordinates.

### 4.2 Image Surface

This interface defines a single function, COMPUTEPOINTAT(), which converts the given normalized 2D *uv* coordinates to a 3D point.

### 4.3 Ray Selector

This interface defines a single function, GETRAYAT(), which converts the given position into a ray object that specifies both the starting point and direction. In addition to using the 3D point, nonlinear direction vector fields can be specified by providing the normalized *uv* image surface coordinates as well as the final output pixel location. However, this functionality is not utilized for GLCs in this project.

### 4.4 Ray Sampler

This interface defines a single function, SAMPLERAY(), which computes the pixel color for the specified ray. The sampler encapsulates the scene description whether that is real image data or synthetic scenes. It also encapsulates any interpolation schemes used.

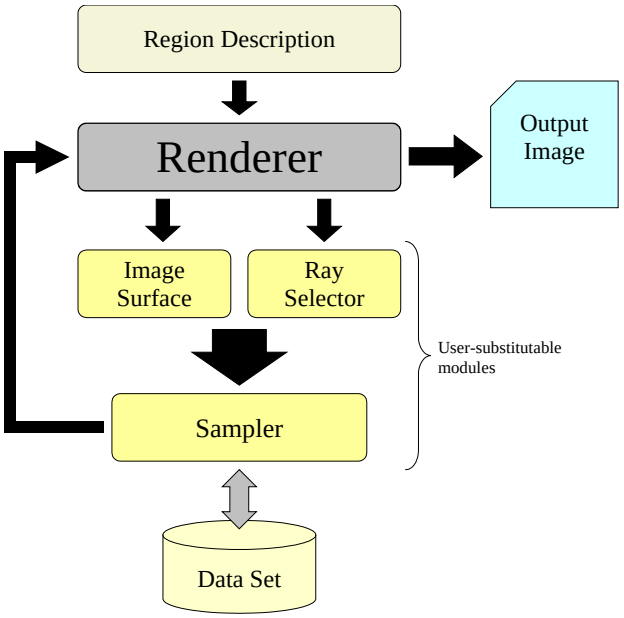


**Figure 2:** The modular software architecture. The Renderer is provided and independent of other modules. The Region Description determines which pixels of the output image are affected. The Image Surface and Ray Selector both determine which rays are queried from the Sampler. The Sampler is responsible for interfacing with the Data Set (whether real or virtual) and computing the color of the desired ray.

### 4.5 Renderer

The renderer object combines all of the above to render into the output image. Because of the above abstractions, the code for this is simply a loop over all pixels in the region with the inner loop contents containing only:

```
uv = region.PixelToUV(pixel);
surfacept = surface->ComputePointAt(uv);
ray = raySelector->GetRayAt(pixel, uv,
    surfacept);
sampler->SampleRay(ray, pPixel);
```

## 5 Provided implementations and Extensions

In order to make the above framework useable, reference implementations are provided that allow rendering any of the GLCs from either synthetic data (via OpenGL) or real image data from the Stanford Light Field Camera Array (LFCA) with metric calibration.

### 5.1 Planar Image class

This defines a simple plane in space given the origin, width, and height vectors. Additional utility functions include computing a rotated plane.

### 5.2 GLC Ray Selector class

This class defines the rays in space for any General Linear Camera by specifying the ray directions at the three corners of the image. These ray directions are then linearly interpolated across the image surface.

### 5.3 OpenGL Sampler class

This abstract class encapsulates the functionality necessary to rapidly select rays from a synthetic OpenGL scene. The scene itself is not defined and left to specialized subclasses to implement the rendering.

### 5.4 MySimpleGLScene class

This class draws the synthetic scene used to demonstrate the perspectives during the *students* animation.

### 5.5 Camera Frame Ray Sampler class

The algorithm for determining the color of a sampled ray is independent given the dataset of camera images and associated camera poses. Thus, this functionality is encapsulated in this abstract class. It is left to specialized subclasses to load the data into the appropriate format, such as the LFCA Sampler class below.

Three interpolation algorithms for determining the ray color are provided. First is a simple nearest-neighbor scheme that selects the closest camera and the pixel therein corresponding to the ray source. Second, a synthetic aperture interpolation scheme is used that averages the pixels that correspond to the ray source from all cameras. Finally, an adjustable synthetic aperture interpolation scheme is used that combines the pixels from only cameras from a cylinder of a specifiable radius around the ray with a linear falloff weighting.

Other frame-based sampler classes can be implemented that utilize geometric proxies for improving the estimated ray sampling.

### 5.6 LFCA Sampler class

This class interprets the metric calibration files available for the LFCA and loads them and the associated images files.

## 6 Results

In order to demonstrate the possible GLCs, a synthetic scene rendered via the OpenGL sampler was created. Figure 4 shows examples of several rendered GLCs. The accompanying video *students.avi* includes corresponding real-image data for each of the GLC views. Only the ray directions are changed throughout these examples; they all use the sample image plane. Thus, it is not surprising that for several of the cameras (most notably the oblique in (C) and the bilinear in (I)) the steeply angled rays cause the useful image area to skew off the image plane. In Yu & McMillan [1], they apply a homography to the image plane in order to rectify the images so they appear more natural. This corresponds to either skewing the image plane itself or specifically altering the sampling along the plane to achieve the same effect.

**Figure 3:** A synthetic aperture image

Figure 3 shows an example of a synthetic aperture image. The synthetic aperture was computed by expanding rays to cylinders and combing the contributions of all cameras within the cylinder. The accompanying video *students_sap.avi* shows an animation of the image plane tilting along with adjusting synthetic aperture effects.

Despite the generality of the architecture and lack of hardware acceleration in the provided implementations, the run times for real image data are reasonable. Nearest-neighbor sampling renders a 500x500 pixel image from 90 input cameras in about 1.44 seconds on a 2.8 GHz Pentium4. Full synthetic aperture images are rendered in about 2.5 seconds. This is almost acceptable for interactive interfaces with current generation hardware. It is expected that more careful optimization utilizing hardware acceleration could potentially provide the necessary speedups.

## 7 Discussion

The hope is that this architecture is (or can be) general enough to allow almost any type of image-based rendering. By separating out individual components of the rendering process, new algorithms can replace old modules and quickly and easily be compared to existing modules.

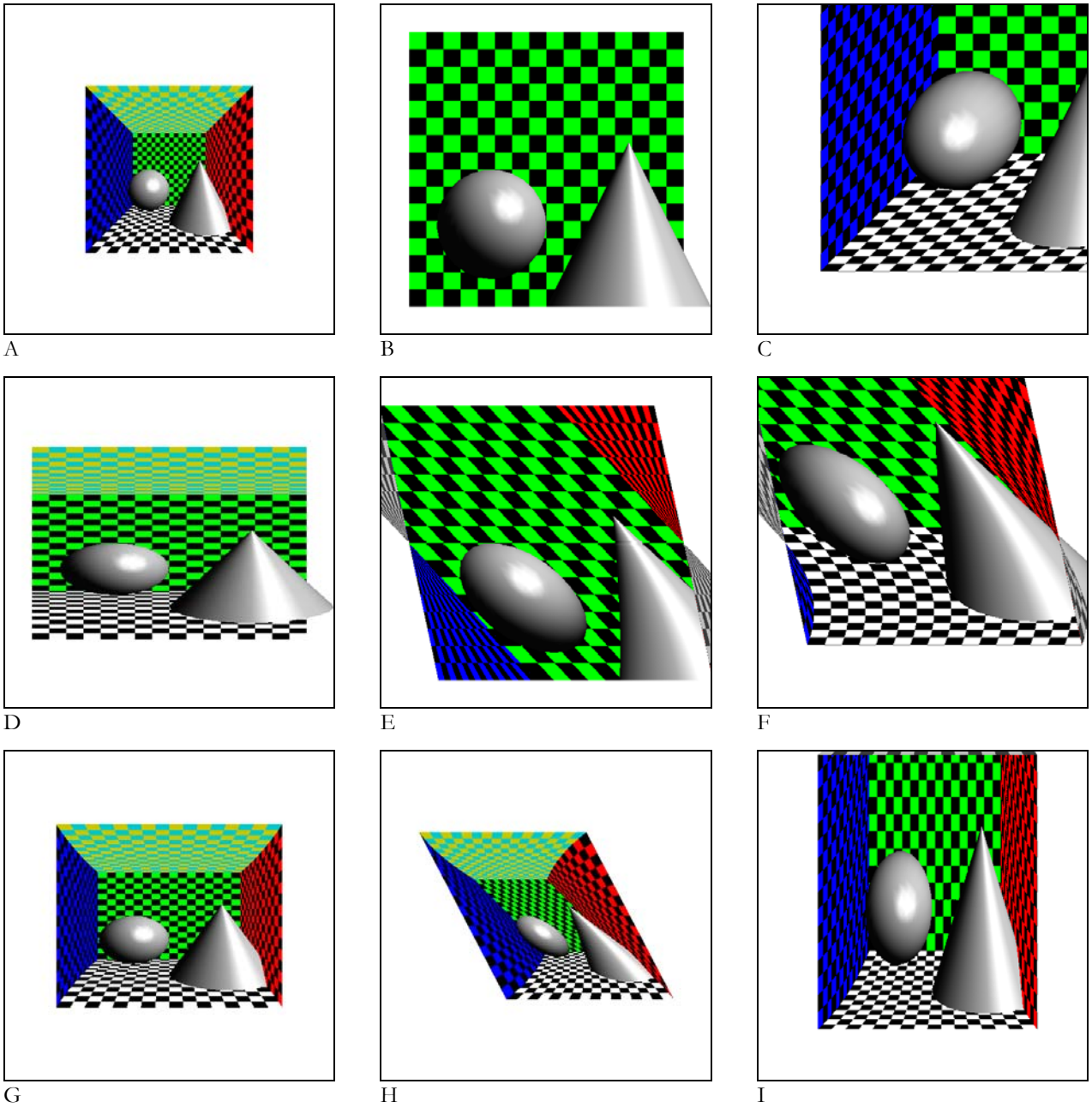


**Figure 4:** Examples of General Linear Cameras. From left to right, top to bottom: (A) perspective, (B) orthographic, (C) orthographic at an angle (oblique), (D) pushbroom, (E) twisted orthographic, (F) twisted orthographic at a downward angle, (G) cross-slits (or glide), (H) pencil (equivalent to rolling shutter plus horizontal motion), (I) bilinear.

This is especially true of the Sampler class. More sophisticated view-interpolation techniques can be substituted for the provided reference implementation and immediately compared. At the moment, it represents a large segment of image-based rendering research.

In addition, an interactive interface for specifying the parameters of these components as well as visualizing the dataset could be very useful. It would be important for a standard software interface to be defined for interacting with the dataset such that future modules can immediately be interfaced with any interactive viewer.

## 8 References

[1] YU, J., AND MCMILLAN, L. 2004. General linear cameras. In *Proceedings of the eighth European conference on Computer Vision (to appear)*, Springer-Verlag New York, Inc.