

# CS590G: Assignment #2 – Image-based Reconstruction

**Out:** October 5, 2004

**Due:** October 28, 2004

## Objective

This objective of this assignment is to use your calibrated camera system to reconstruct novel views of an object of your choosing. There are multiple ways to obtain novel views of the object. In this assignment, you will implement one particular approach. The next assignment (or mini-project) will allow you to extend the system in one of many ways or even to create a new system entirely.

In class, we have seen image morphing, view morphing, view-dependent texture-mapping, image warping, and lightfield/lumigraph rendering. In this assignment, we will use a combination of image morphing and view-dependent texture mapping to create novel views of an object. This assignment will require you to use the two previous assignments. The whole assignment is not that complicated but it is beneficial for you to think ahead about mini-projects and to think about what software infrastructure is best for you. As always, I do recommend **STARTING EARLY!**

## Detailed Description

For this assignment, you will capture various images from viewpoints surrounding the object placed on the calibration pad and create a novel view by interpolating between captured images. The viewpoints should be distributed on the surface of an imaginary plane or sphere (see Figure 1). The viewpoints do not need to span the entire area in front of the object, but the more the better. A denser set of views will produce better quality reconstructions while covering a wider area will give the observer more freedom to view the object.

### Step 0 – Capture

To begin, capture a set of images from in front of the object while keeping the object more or less centered in the image and spanning as much of the field-of-view as possible (but remember, you will also need to calibrate each image – i.e., compute the camera's position using the calibration pad). You need to capture *at least* a 4x4 grid of images. You can fully calibrate each image separately but better results will be obtained by using features in all images to create a single set of internal camera parameters and then compute the external parameters for each image (e.g., position and orientation of each captured image).

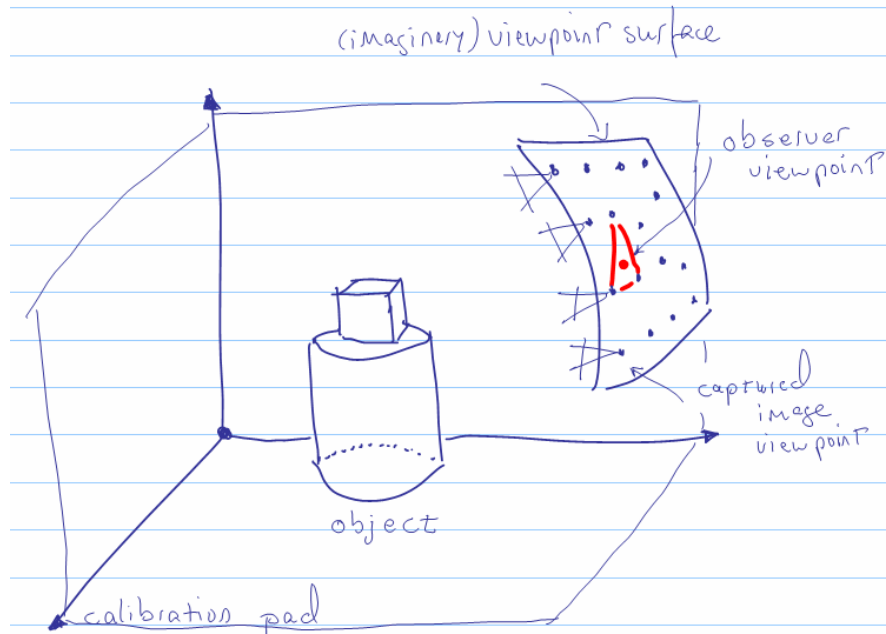


Figure 1. Diagram of capture and render setup.

MINOR EXTRA CREDIT: you may capture more images but this will increase the amount of tedious manual work you must do unless you do even fancier extra credit work.

### Step 1 – Feature Correspondence

Choose object feature points that are generally visible from all images. Using assignment #0, select the projections of the feature points on all the captured images thus forming a corresponded feature set. Label the features 1 to  $f$ . Not all features will be in all images, but the more images a feature is in, the better. The number, coverage, and density of feature points will also affect image quality. The number of feature points you need also depends on the object you are reconstructing.

### Step 2 – Calculating Weights

The viewpoints of the images you captured should lie approximately on the surface of a sphere (centered on the object) or a plane (“in front of” the object). For this assignment, it is sufficient if you assume the observer viewpoint is always on this surface (let’s call it the “viewpoint surface”, as opposed to the “object surface”). Hence, the observer has 2 degrees freedom for translation. By triangulating the image viewpoints, you can easily select the image triple that surrounds the observer viewpoint. You may automatically triangulate the viewpoints, e.g. use Delaunay triangulation, or manually create a triangulation. If you project the observer’s viewpoint onto the triangular patches of the viewpoint surface, you can compute the barycentric coordinates of that observer viewpoint and obtain three weights ( $u, v, w$ ) that tell you how much of each captured image to use.

EXTRA CREDIT: instead of restricting the novel viewpoint to a lie on the viewpoint surface, let the observer viewpoint freely roam around 3D space. This implies that you will need to zoom in/out to *simulate* getting closer or farther from the object. You will only be “simulating” moving forward or backwards unless you do much more sophisticated operations like 3D image warping. You can compute the scale factors by, for instance, computing the bounding box of a proxy of the object and using this to “scale” the image to the appropriate size based on the distance to it.

### Step 3 – Interpolation and Rendering

The next step is to interpolate between the corresponded feature points and the images themselves. Using the aforementioned weights, you can compute an interpolated 2D screen position for a set of features common to the three captured images (e.g., when  $(u,v,w)=(1,0,0)$ , the feature position is that of image ‘u’; when  $(u,v,w)=(0,1,0)$ , the feature position is that of image ‘v’; when  $(u,v,w)=(0.5,0.5,0)$ , the feature position is halfway between that of image ‘u’ and image ‘v’; etc. Similarly, like in view-dependent texture mapping, the weights  $(u,v,w)$  can be used to blend between three images. To accomplish this, triangulate the feature points in image space (again, using Delaunay triangulation or manually, for instance). For each triangle, you will have three textures. Use texture-blending to render all three textures with weights  $(u,v,w)$ .

Remember, you will be interpolating between the 2D feature positions of the three relevant captured images and you will be blending between the three images, texture-mapped onto a triangulation of the feature points.

You may choose on your own what to do with the background “surrounding” the object. You can choose to either create additional features to interpolate it, or you can segment out the background and only render the foreground object you are reconstructing – just be careful not to clip parts of the object you are trying to show.

### Step 4 – User Interface

For this assignment, you will at least need to create a mouse-based or keyboard-based user interface that allows the user to *interactively* change the observer’s viewpoint. The observer’s viewpoint can translate anywhere along the surface constructed from the captured viewpoints and it can rotate about the screen normal as well (e.g., the reconstructed image can be rotated in the screen plane – as if you were tilting your head from left to right). Thus your user interface should support at least 2 degrees of translation freedom and 1 degree of rotation freedom. Your user interface must also support a “reset” button (e.g., ‘r’ key) which brings the view back to a nicely centered view of the object.

EXTRA CREDIT 1: you can create visualization tools, including a graphical user-interface, help the observer understand the rendering process. For instance, an option to hit a key and see which are the current features; options to toggle feature interpolation

and texture blending; options to select how many/which captured images to use; visualizations that pictorially show what is happening and where the observer's viewpoint is on the viewpoint surface (e.g., in the corner of the window draw a small subwindow; inside the subwindow show a 3D rendering of the geometrical configuration of the calibration pad, a proxy for the object, the viewpoints of the captured images, and a depiction of the observer's viewpoint – for example, a live version of figure 1).

EXTRA CREDIT 2: another item for extra credit is to support more degrees of rotational freedom. To accomplish this, you will have to think much more in “3D” – think about it. This can be a very cool feature! Come see me if you are interested in more details...

EXTRA CREDIT 3: use the camera and the simple camera grabbing interface to grab images on the fly and reconstruct the object right in front you. For instance, ask the user to “grab” several images. Then calibrate each image by manually selecting the features and their correspondences. Finally, display the reconstructed the object.

### **Grading and Deliverable**

Your grade will be influenced by how well your particular object is reconstructed and how well you completed the assignment requirements. Moving the observer's viewpoint far from the captured viewpoints will create a badly distorted image – that is ok. Moving the observer's viewpoint on top of one of the captured images should produce a perfectly reconstructed image. Moving in between captured viewpoints, will construct novel views where the amount of “ghosting” and “blurriness” depends on your viewpoint sampling density, feature selection, and object geometry. If you choose an excessively complicated object, you might need more images or features than someone else's object. If you choose a solid-color box as your object and only need a couple of features, you did not necessarily do better than somebody who chose a more complicated object and has reconstruction artifacts.

As with assignment #1, I will schedule a brief demo with each of you and you can demo your system to me. A zipped file containing source, executable, and images must be mailed to me before the due date.

---

Additional details:

I have placed in <http://www.cs.purdue.edu/~aliaga/cs590g/software/triangulation.zip> a copy of a Delaunay triangulation package. This software constructs the Voronoi region of a set of 2D points, computes the dual of that (called a “Delaunay triangulation”) and returns a triangulation where all triangles are roughly the same size. The original makefile compiles the code using GNUmake (I have not included it). You'll have to write a new makefile or just incorporate the code into your project – it's pretty straightforward.

Just compile everything with the environment variable LIBRARY defined. If you want to use another triangulation package, go ahead. To use this code, just include “fortunedelaunay.h” and call the function delaunay\_triangulation(float \*points, int n, int \*\*tris, int \*ntris). The points array contains a list of ‘n’ 2D vertices and the tris array contains ‘ntris’ triples of vertex indices that form the computed triangulation. This code is an adaptation of freeware written by a well known researcher, Steve Fortune.

Barycentric coordinates are a way of representing a 2D point inside a triangle in terms of the surrounding three vertices (barycentric coordinates are actually more general and apply to N-dimensional spaces, but for this assignment, let’s just worry about N=2). Given the three vertices of the triangle (p1,p2,p3) and a point q, here’s a function to compute the barycentric coordinates (u,v,w) of the point q:

```

/*****
Given three points and a point q, compute the barycentric coordinates
of the point q.
*****/
static void
ComputeBarycentricCoords(float q[2], float p1[2], float p2[2], float p3[2],
                        float *u, float *v, float *w)
{
    // 3x3 determinant of the denominator
    float d = p1[0]*p2[1] - p2[0]*p1[1] - p1[0]*p3[1] +
              p3[0]*p1[1] + p2[0]*p3[1] - p3[0]*p2[1];

    // each numerator divided by common denominator
    *u = ( q[0]*p2[1] - p2[0]* q[1] - q[0]*p3[1] +
           p3[0]* q[1] + p2[0]*p3[1] - p3[0]*p2[1]) / d;

    *v = (p1[0]* q[1] - q[0]*p1[1] - p1[0]*p3[1] +
           p3[0]*p1[1] + q[0]*p3[1] - p3[0]* q[1]) / d;

    *w = (p1[0]*p2[1] - p2[0]*p1[1] - p1[0]* q[1] +
           q[0]*p1[1] + p2[0]* q[1] - q[0]*p2[1]) / d;
}

```

To perform image capture, you may do the same as you did for Assignment #1 or you may use the real-time capture library provided with the Flea camera. Using real-time capture, will allow you to do the extra credit item of building the model on the fly. The capture library is very easy to use. Basically, you setup the camera, grab an image, and convert it to RGB and then you have an array of pixels just like you read in from a PPM file. A very short example program to interface the camera is in “C:\Program Files\Point Grey Research\PGR FlyCapture\src\pgrflycapturetest”. Use this as a guide.