SIGHTATIONS

G

Simulating Calibrated Cameras in OpenGL

Edit: The article below is out of date and incomplete. You can find a revised, expanded, and (hopefully) clearer version at my new blog (http://ksimek.github.io/2013/06/03/calibrated_cameras_in_opengl/). — Kyle, April 3, 2013

If you've ever tried to simulate a calibrated camera in opengl, you've probably realized it isn't as straightforward as you might like. The `glFrustum` and `gluProjection` functions can get you most of the way there, but the can pose problems. First, there isn't an intuitive mapping to camera parameters like principal point and focal length. Second, there isn't any way to represent camera axis skew and non-square pixels (exhibited by CCD arrays in cheap digital cameras). A more straightforward approach is to build the projection matrix directly, which allows you to represent all of the intrinsic camera parameters: focal length, pixel aspect ratio, principal point (x,y), and axis skew.

The glFrustum Matrix

Lets start by looking at the matrix generated (http://msdn.microsoft.com/en-us/library/dd373537(v=VS.85).aspx) by an OpenGL call to `glFrustum(left, right, top, bottom, near, far)`:

$$\text{glFrustum} = \begin{pmatrix} \frac{2\ near}{right-left} & 0 & A & 0 \\ 0 & \frac{2\ near}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where

$A = \frac{right+left}{right-left}$

$B = \frac{top+bottom}{top-bottom}$

$C = -\frac{far+near}{far-near}$

$D = -\frac{2\ far\ near}{far-near}$

Not everything in this matrix is obvious at first glance, but you should at least start to see similarities between this matrix and the intrinsic camera parameter matrix. Note that opengl's "projection" preserves the z-depth, which explains the third row. Removing the third row, this has the same structure as an intrinsic camera matrix (a.k.a. "K" in Hartley and Zisserman (http://www.amazon.com/Multiple-View-Geometry-Computer-Vision/dp/0521540518/ref=sr_1_1?ie=UTF8&s=books&qid=1280966294&sr=8-1)) with zero axis skew:

$$K = \begin{pmatrix} \alpha & s & x_0 & 0 \\ 0 & \beta & y_0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where
$\alpha,\ \beta$ are the focal length in x and y units (a.k.a. "scaling factor" in x, y directions)
s is the axis skew.
x_0, y_0 are the principal point offset.

However, you can't just replace elements (0,0) and (1,1) of OpenGL's projection matrix with your camera's focal length. The reason is that this matrix actually does two things at once: (1) it performs a perspective projection, and (2) it rescales the coordinates to Normalized Device Coordinates (http://medialab.di.unipi.it/web/IUM/Waterloo/node15.html)s Part (1) is exactly analogous to multiplying by the intrinsic camera matrix, while part (2) has arguably nothing to do with cameras and is just required by the OpenGL an architecture.

Decomposing the glFrustum Matrix

So how can we separate these two operations to get at the real camera matrix? Lets start by looking at projection that doesn't add any perspective: `glOrtho()`. The matrix generated by `glOrtho` (http://msdn.microsoft.com/en-us/library/dd373965(v=VS.85).aspx) only performs part (2) above, namely, conversion into Normalized Device Coordinates. Lets looks at its matrix:

$$\text{glOrtho} = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & -\frac{2}{far-near} & t_x \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where
$t_x = -\frac{right+left}{right-left}$

$t_y = -\frac{top+bottom}{top-bottom}$

$t_z = -\frac{far+near}{far-near}$

We see a number of similarities between the elements of `glOrtho` and the elements of `glFrustum()`. In fact, we can factor out `glOrtho` from the `glFrustum` matrix to get:

$$\text{glFrustum} = \text{glOrtho} * \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & X & Y \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where

$$X = near + far$$

$$Y = near\ far$$

We can ignore X and Y, as they don't pertain to the calibrated camera matrix; they're just used to map z-depths in OpenGL. Notice that the second matrix now looks strikingly like the intrinsic camera matrix, K. Lets spend a moment to interpret this result. A 3D coordinate passing through this matrix is first multiplied by our intrinsic camera matrix, which does a perspective transformation. Then it passes through `glOrtho()`, which simply scales and translates the point into Normalized Device Coordinates. Alternatively, we can think of the perspective transformation as converting a trapezoidal-prism-shaped viewing volume into a rectangular-prism-shaped viewing volume, which `glOrtho()` scales and translates into the 2x2x2 cube in Normalized Device Coordinates.

Relation to Calibrated Camera

Now that we've decomposed the glFrustum matrix we can draw a direct comparison to our calibrated camera. Namely, glFrustrum is setting up a camera with:

- Focal length of "near"
- A pixel aspect ratio of (near / near) = 1
- Zero skew
- A principal point at the exact center (0,0)*

* I should note that `glFrustum` *can* represent a camera with an off-center principal point, but in our decomposition above, this occurs in the `glOrtho` matrix, not in the camera matrix. This will make it more convenient when we simulate the calibrated camera shortly.

Notice that element (3,2) of the projection matrix is '-1'. This is because the camera looks in the negative-z direction, which is the opposite of the convention used by Hartley and Zisserman. This inversion of the camera's z-axis is achieved by left-multiplying the intrinsic camera matrix by a z-inverting matrix:

$$\begin{pmatrix} \alpha_x & s & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \alpha_x & x & -x_0 & 0 \\ 0 & \alpha_y & -y_0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Note that points with negative z-coordinates will now map to a homogeneous 2D point with positive w-coordinate. This is important because OpenGL only renders points whose 2D homogeneous 'w' coordinates are positive (http://stackoverflow.com/questions/2286529/why-does-sign-matter-in-opengl-projection-matrix).

An alternative interpretation is to obtain this matrix by negating the focal length and skew, and then multiplying the entire matrix by negative 1 (a no-op when dealing with homogenous matrices). We may interpret focal length as the *position* of the virtual image plane lying in front of the camera, and since the camera's direction has reversed, the camera focal lengths have become negative. Also, since inverting the z-axis means clockwise is now counter-clockwise, the skew parameter has become negated, too. In other words:

$$-1 * \begin{pmatrix} -\alpha_x & -s & x_0 & 0 \\ 0 & -\alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \alpha_x & x & -x_0 & 0 \\ 0 & \alpha_y & -y_0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Transforming this intrinsic camera matrix back into OpenGL form gives:

$$Proj = glOrtho * \begin{pmatrix} \alpha & s & -x_0 & 0 \\ 0 & \beta & -y_0 & 0 \\ 0 & 0 & X & Y \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where

$X = near + far$

$Y = near\ far$

With this matrix, we can simulate all five intrinsic camera parameters as opposed to `glFrustum`'s three (a single focal length, and (x,y), principal point). In addition, this representation decouples focal length from the near plane, which were bound to each other in `glFrustum()`. When skew and principal point offset are zero, pixel aspect ratio is unity, and focal length equals the near plane, the result is exactly the same result as `glFrustum()`.

Calling glOrtho Correctly

Before we close, I should note that implementing this correctly requires passing the proper parameters to `glOrtho()`. Specifically, you should pass the pixel coordinates of the left, right, bottom, and top of the window you used when performing calibration. For example, lets assume you calibrated using a 640×480 image. If you used a pixel coordinate system whose origin is at the top-left, with the y-axis increasing in the downward direction, you would call `glOrtho(0, 640, 480, 0, near, far)`. If you calibrated with an origin at zero and normal leftward/upward x,y axis, you would call `glOrtho(-320, 320, -240, 240, near, far)`.

If you'd prefer your implementation to be independent of your calibration approach, you should "standardize" your camera calibration matrix in a preprocessing step. Start by translating screen coordinates so the origin is in the center of the image:

EQUATION HERE

Then, flip the y-axis if you used a coordinate system where y increases in the downward direction:

EQUATION HERE

Now, you can use the same call to glOrtho for all situations: glOrtho(-width / 2, width/2, -height/2, height/2, near. far). Increasing width and height will result in showing more of the scene, as if you used a larger-sized film surface in a pinhole camera, or a larger CCD array in a digital camera. Of course, correct simulation requires the correct using values for width and height corresponding to the

image used during calibration, so we still haven't achieved an implementation that is independent of the specific calibration scenario. The last step is to transform camera coordintes into normalized device coordinates yourself:

EQUATION HERE

If you work with your camera matrix in this form, you don't have to call glOrtho at all. Now, changing the window or viewport size will scale the display, without showing any more or less of the scene. Of course, combining the preprocessing steps above will result in exactly multiplying by the glOrtho matrix with the appropriate parameters described above based on your calibration scenario.

So that's it! I hope this is helpful when you're designing your own simulation of a calibrated camera!

Reference

"Multiple View Geometry in Computer Vision (http://www.amazon.com/Multiple-View-Geometry-Computer-Vision/dp/0521540518/ref=sr_1_1?ie=UTF8&s=books&qid=1280966294&sr=8-1)", Hartley and Zisserman
"Computer Vision: A Modern Approach (http://www.amazon.com/Computer-Vision-Approach-David-Forsyth/dp/0130851981/ref=sr_1_1?ie=UTF8&s=books&qid=1280966331&sr=8-1)", Forsyth and Ponce
OpenGL glFrustum() Documentation (http://msdn.microsoft.com/en-us/library/dd373537(v=VS.85).aspx)
OpenGL glOrtho() Documentation (http://msdn.microsoft.com/en-us/library/dd373965(v=VS.85).aspx)
"OpenGL Projection Matrix (http://www.songho.ca/opengl/gl_projectionmatrix.html)", songho.ca (Very nice: derives the glFrustum matrix)
"Why does sign matter in opengl projection matrix (http://stackoverflow.com/questions/2286529/why-does-sign-matter-in-opengl-projection-matrix)", StackOverflow.com
"Questions concering the ARToolkit (http://www.hitl.washington.edu/artoolkit/mail-archive/message-thread-00653-Re--Questions-concering-.html)" ARToolkit Mailing List — Early inspiration for this approach

Tags: Augmented Reality (https://sightations.wordpress.com/tag/augmented-reality/), Camera Calibration (https://sightations.wordpress.com/tag/camera-calibration/), Graphics (https://sightations.wordpress.com/tag/graphics/), Intrinsic Parameters (https://sightations.wordpress.com/tag/intrinsic-parameters/), migrate (https://sightations.wordpress.com/tag/migrate/), OpenGL (https://sightations.wordpress.com/tag/opengl/)

- **COMMENTS** *4 Comments*
- **CATEGORIES** *Camera Calibration, Graphics, OpenGL*

4 Responses to "Simulating Calibrated Cameras in OpenGL"

CHOI May 29, 2013 at 7:38 pm #

Could you please explain how you set the modelview transformation(maybe using gluLookat) using the extrinsic parameter?

REPLY

Kyle Simek May 30, 2013 at 9:55 am #

I don't recommend using gluLookAt when working with an extrinsic matrix, because it's unnecessarily complicated. Not that it's terribly bad, but the alternative is so much simpler: call glLoadMatrix(Rt) where Rt is the extrinsic matrix, [R | t]. If it isn't working, there are a few common mistakes that I've seen. First, there are several equivalent ways to decompose the camera matrix into intrinsic/extrinsic matrices, several of which will result in invalid rendering in opengl. For example, your decomposition might result in a rotation matrix with a negative determinant, which will reverse your surface normals among other nasty things. I discuss camera matrix decomposition in the context of opengl at my new blog, which should help you ensure your decomposition is a good one. Second, be careful about reference frames. For example, t is the translation vector in camera coordinates, which describes the location of the origin relative to the camera. Importantly, it is **not** not the location of the camera in world coordinates, which is -R't. These issues of reference frame are easy to get wrong; I discuss them at length in my article on the extrinsic matrix. That article also describes how to use the extrinsic matrix with gluLookat, and gives an interactive demo of the three different ways of parameterizing the extrinsic matrix.

REPLY

CHOI June 4, 2013 at 12:31 am #

Thanks a lot, Kyle.

Your blog's are so well organized that I simulated a camera in OpenGL sucessfully after reading them.

They are very helpful!

Thiago June 4, 2013 at 6:33 am #

Could you post your solution, CHOI?

REPLY