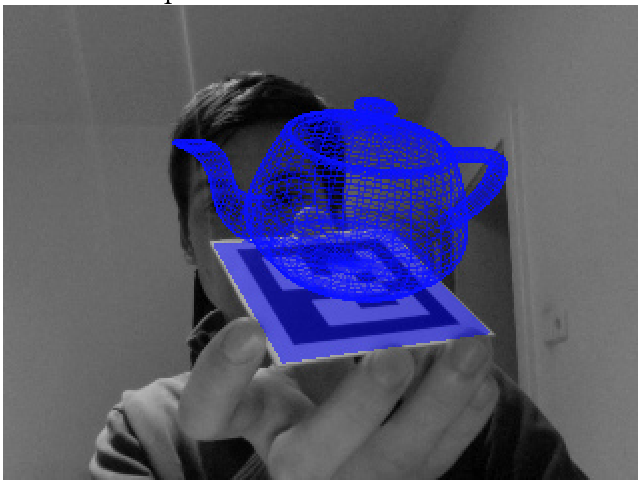
Calibrated Cameras in OpenGL without glFrustum

June 03, 2013
Author's note: some of this content appeared on my old blog as "Simulating Calibrated Cameras in OpenGL", which contained some errors and missing equations and suffered from general badness. I hope you'll find this version to be less terrible.
Update (June 18, 2013): added negative signs to definitions of C' and D'.
Update (August 19, 2013): James Gregson has posted an implementation in C++. I haven't tested it myself, but it looks quite nice.


Simulating a calibrated camera for augmented reality.
Credit: thp4

You've calibrated your camera. You've decomposed it into intrinsic and extrinsic camera matrices. Now you need to use it to render a synthetic scene in OpenGL. You know the extrinsic matrix corresponds to the modelview matrix and the intrinsic is the projection matrix, but beyond that you're stumped. You remember something about `gluPerspective`, but it only permits two degrees of freedom, and your intrinsic camera matrix has five. glFrustum looks promising, but the mapping between its parameters and the camera matrix aren't obvious and it looks like you'll have to ignore your camera's axis skew. You may be asking yourself, "I have a matrix, why can't I just use it?"

You can. And you don't have to jettison your axis skew, either. In this article, I'll show how to use your intrinsic camera matrix in OpenGL with minimal modification. For illustration, I'll use OpenGL 2.1 API calls, but the same matrices can be sent to your shaders in modern OpenGL.

glFrustum: Two Transforms in One

To better understand perspective projection in OpenGL, let's examine `glFrustum`. According to the OpenGL documentation,

> glFrustum describes a perspective matrix that produces a perspective projection.

While this is true, it only tells half of the story.

In reality, `glFrustum` does two things: first it performs perspective projection, and then it converts to normalized device coordinates (NDC). The former is a common operation in projective geometry, while the latter is OpenGL arcana, an implementation detail.

To give us finer-grained control over these operations, we'll separate projection matrix into two matrices *Persp* and *NDC*:

$$Proj = NDC \times Persp$$

Our intrinsic camera matrix describes a perspective projection, so it will be the key to the *Persp* matrix. For the *NDC* matrix, we'll (ab)use OpenGL's `glOrtho` routine.

Step 1: Projective Transform

Our 3x3 intrinsic camera matrix *K* needs two modifications before it's ready to use in OpenGL. First, for proper clipping, the (3,3) element of *K must* be -1. OpenGL's camera looks down the *negative* z-axis, so if K_{33} is positive, vertices in front of the camera will have a negative *w* coordinate after projection. In principle, this is okay, but because of how OpenGL performs clipping, all of these points will be clipped.

If K_{33} isn't -1, your intrinsic and extrinsic matrices need some modifications. Getting the camera decomposition right isn't trivial, so I'll refer the reader to my earlier article on camera decomposition, which will walk you through the steps. Part of the result will be the negation of the third column of the intrinsic matrix, so you'll see those elements negated below.

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

For the second modification, we need to prevent losing Z-depth information, so we'll add an extra row and column to the intrinsic matrix.

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

where

$$A = near + far$$
$$B = near * far$$

The new third row preserve the ordering of Z-values while mapping *-near* and *-far* onto themselves (after normalizing by *w*, proof left as an exercise). The result is that points between the clipping planes remain between clipping planes after multiplication by *Persp*.

Step 2: Transform to NDC

The *NDC* matrix is (perhaps surprisingly) provided by `glOrtho`. The *Persp* matrix converts a frustum-shaped space into a cuboid-shaped shape, while `glOrtho` converts the cuboid space to normalized device coordinates. A call to `glOrtho(left, right, bottom, top, near, far)` constructs the matrix:

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

where

$$\begin{aligned} t_x &= -\frac{right + left}{right - left} \\ t_y &= -\frac{top + bottom}{top - bottom} \\ t_z &= -\frac{far + near}{far - near} \end{aligned}$$

When calling `glOrtho`, the *near* and *far* parameters should be the same as those used to compute A and B above. The choice of top, bottom, left, and right clipping planes correspond to the dimensions of the original image and the coordinate conventions used during calibration. For example, if your camera was calibrated from an image with dimensions $W \times H$ and its origin at the top-left, your OpenGL 2.1 code would be

```
glLoadIdentity();
glOrtho(0, W, H, 0, near, far);
glMultMatrix(persp);
```

Note that *H* is used as the "bottom" parameter and *0* is the "top," indicating a y-downward axis convention.

If you calibrated using a coordinate system with the y-axis pointing upward and the origin at the center of the image,

```
glLoadIdentity();
glOrtho(-W/2, W/2, -H/2, H/2, near, far);
glMultMatrix(persp);
```

Note that there is a strong relationship between the `glOrtho` parameters and the perspective matrix. For example, shifting the viewing volume left by X is equivalent to shifting the principal point right by X. Doubling α is equivalent to dividing *left* and *right* by two. This is the same relationship that exists in a pinhole camera between the camera's geometry and the geometry of its film--shifting the pinhole right is equivalent to shifting the film left; doubling the focal length is equivalent to halving the dimensions of the film. Clearly the two-matrix representation of projection is redundant, but keeping these matrices separate allows us to maintain the logical separation between the camera geometry and the image geometry.

Equivalence to glFrustum

We can show that the two-matrix approach above reduces to a single call to `glFrustum` when α and β are set to *near* and s, x_0 and y_0 are zero. The resulting matrix is:

$$Proj = NDC * Persp$$

$$= \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & -\frac{2}{far-near} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

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

where

$$\begin{aligned} A' &= \frac{right + left}{right - left} \\ B' &= \frac{top + bottom}{top - bottom} \\ C' &= -\frac{far + near}{far - near} \\ D' &= -\frac{2\ far\ near}{far - near} \end{aligned}$$

This is equivalent to the matrix produced by glFrustum.

By tweaking the frame bounds we can relax the constraints imposed above. We can implement focal lengths other than *near* by scaling the frame:

$$\begin{aligned} left' &= \left(\frac{near}{\alpha}\right) left \\ right' &= \left(\frac{near}{\alpha}\right) right \\ top' &= \left(\frac{near}{\beta}\right) top \\ bottom' &= \left(\frac{near}{\beta}\right) bottom \end{aligned}$$

Non-zero principal point offsets are achieved by shifting the frame window:

$$\begin{aligned} left'' &= left' - x_0 \\ right'' &= right' - x_0 \\ top'' &= top' - y_0 \\ bottom'' &= bottom' - y_0 \end{aligned}$$

Thus, with a little massaging, `glFrustum` can simulate a general intrinsic camera matrix with zero axis skew.

The Extrinsic Matrix

The extrinsic matrix can be used as the modelview matrix without modification, just convert it to a 4x4 matrix by adding an extra row of *(0,0,0,1)*, and pass it to `glLoadMatrix` or send it to your shader. If lighting or back-face culling are acting strangely, it's likely that your rotation matrix has a determinant of -1. This results in the geometry rendering in the right place, but with normal-vectors reversed so your scene is inside-out. The previous article on camera decomposition should help you prevent this.

Alternatively, you can convert your rotation matrix to axis-angle form and use `glRotate`. Remember that the fourth column of the extrinsic matrix is the translation *after* rotating, so your call to `glTranslate` should come *before* `glRotate`. Check out this previous article for a longer discussion of the extrinsic matrix, including how to it with `glLookAt`.

Conclusion

We've seen two different ways to simulate a calibrated camera in OpenGL, one using glFrustum and one using the intrinsic camera matrix directly. If you need to implement radial distortion, it should be possible with a vertex shader, but you'll probably want a high poly count so the curved distortions appear smooth--does anyone have experience with this? In a future article, I'll cover how to accomplish stereo and head-tracked rendering using simple modifications to your intrinsic camera parameters.

Posted by Kyle Simek

Calibrated Cameras and gluPerspective → ← Dissecting the Camera Matrix, Part 2: The Extrinsic Matrix

monxarat • 7 months ago
How to using it in OpenCV Android (with JNI)
Reply • Share ›

Lei Wen • 3 years ago
Hi Kyle,
It is really great article series indeed!
But I am not sure for one thing. Could I have the assumption that intrinsic matrix is equaling the matrix produced by glFrustum, and extrinsic matrix is equaling to the modelview matrix in opengl?
Reply • Share ›

Stephen • 4 years ago
Hi, Kyle

Thanks for the sharing. I was following your "Dissecting the camera matrix" series all the way here and have learnt a lot. Regarding this blog, I understand that left, right, bottom and top can be determined by the dimension of the images but what are the values for near and far? Are they related to alpha and beta of the intrinsic matrix? I couldn't find the answer in this article.

Reply • Share ›

ksimek Mod → Stephen • 4 years ago

The near and far plane exist in OpenGL only for technical reasons. They don't correspond to any concept in a physical camera.

Reply • Share ›

Lei Wen → ksimek • 3 years ago

I don't understand this saying...
Isn't the near plane also is the focal plane in the term of opengl?

Reply • Share ›

Stephen → ksimek • 4 years ago

So their values are set based on an application's need, right?

Reply • Share ›

ksimek Mod → Stephen • 4 years ago

Right. You need some idea of the range of depths where the geometry of interest will occur and set the near and far planes accordingly. Set them too close together and geometry gets clipped. Set them too far apart and the depth buffer stops being able to distinguish geometry at similar depths.

Reply • Share ›

Stephen → ksimek • 4 years ago

Thanks for your clear explanation, really helpful.

Reply • Share ›

Cameron Lowell Palmer • 4 years ago

So how does this deal with the problem of aspect ratio of screen and camera image mismatch? I'm seeing a distortion on mismatch screen to camera image aspect ratio. In the case of an iPhone 5S I'm seeing correct movement with video frames streaming at 1280x720, but weird distortions at 640x480.

Reply • Share ›

ksimek Mod → Cameron Lowell Palmer • 4 years ago

It can be handled a few different ways. If you don't want distortion, you either have to crop the sides or add padding on the top and/or bottom. The simplest way is to rescale screen coordinates by right-multiplying the intrinsic matrix by diag([k,k,1]), where k is either 640/1280 or 480/720 (corresponding to padding or cropping, respectively) and then use glOrtho to set up a 640x480 clipping volume. If you're overlaying an image with a virtual scene, you'll need to perform the same scaling to your image, and then crop/pad it to 640x480.

This raises the question of which side to crop or pad (or both). Since this is pure scaling (no translation), the image origin will be unaffected, and the affect of cropping/padding depends on your image coordinate conventions during calibration. If you used a top-left origin, the padding will appear on the right, and cropping occurs at the bottom, because that leaves the origin where it started. In this case, glOrtho(0, 480 640, 0, near, far) would be the proper clipping operation.

In cases of extreme cropping/padding, this will shift the principal point significantly, causing the scene to look strange. Arguably a better way is apply 2D translation to the intrinsic matrix before scaling in order to remove any principal point offset. The simplest way to do this is to set K(0,3) = K(1,3) = 0. Then scaling will be centered on the principal point and glOrtho(-480/2, 480/2, 640/2, -640/2, near, far) results in symmetric clipping/padding. Then apply the same scaling and symmetric clipping/padding to the real image and you should be in good shape.

1 ^ | ˅ • Reply • Share ›

Cameron Lowell Palmer ➦ ksimek • 4 years ago

To clarify the letterbox scenario, I have provided two images that demonstrate the stretching along the x-axis. The background 640x480 video is correctly displayed, but as you move from the center to the edge you can see the image moves faster than one would like. This being a result of the perspective projection matrix stretching to fit the screen while the background orthographic projection remains correct.

I tried scaling the intrinsic matrix as you suggested, but that of course means that the y-axis will also be skewed and a simple scaling in this scenario will lead to strange results, and in fact did. Am I missing something here?

It just seems to me it would be great if I could place this camera projection matrix inside of an orthographic projection for the screen or in other words, scale the projection matrix into the coordinates of the screen projection. Another possibility would be to dodge the whole issue and handle the size of the OpenGL ES view from iOS such that the image and projection matrix match. However, that isn't terribly satisfying intellectually.

^ | ˅ • Reply • Share ›

ksimek Mod ➦ Cameron Lowell Palmer • 4 years ago

Thanks for the clarifying photos. Sorry, I misunderstood the scenario -- I thought your calibration image was larger and wider than the screen size, but it seems the opposite is true. In this case, use the opposite advice: scale in the intrinsic matrix by k=720/480=1.5 and use dimensions 1280 and 720 when calling glOrtho.

I understand your concern about scaling the intrinsic matrix equally on both axes. We aren't correcting the aspect ratio in this step, and you're right, if we only did this, nothing would be fixed. It's the new dimensions passed to glOrtho that squeezes the x-axis in the appropriate way. By using wide-screen dimensions in glOrtho, we're squeezing a larger world region into the same clipping space. As a result, when we display, things that move quickly in the world should move more slowly on the screen.

There are several ways to fix this issue, but the decomposition I've proposed above aims to preserve the logical separation of issues of projection (the Proj matrix) from issues of display (the NDC matrix). Since a pinhole projection should always scale x and y equally, we use equal

see more

1 ^ | ⌄ • Reply • Share ›

Cameron Lowell Palmer → Cameron Lowell Palmer • 4 years ago

Those images are larger than anticipated

^ | ⌄ • Reply • Share ›

Cameron Lowell Palmer → ksimek • 4 years ago

Thank you for the quick and thorough response. I'll try this out today.

^ | ⌄ • Reply • Share ›

Tomas • 5 years ago

I don't know about you guys, but James Gregson C++ implementation is missing some negation signs (line 52 for "u0", line 53 for "v0", line 55 - added two negation signs which are unnecessary) & etc. Problem: the created perspective transform inverts the depth - so a closer points obstruct farther ones.

Here's a the fixed code: http://pastebin.com/h8nYNWJY

^ | ⌄ • Reply • Share ›

ksimek Mod → Tomas • 5 years ago

Good catch--as you say, the third column in James's implementation is negated (from the comments, it looks like this was originally different but maybe he edited it). Thanks for sharing your implementation.

It's possible James's extrinsic matrix has a negated third row, which would cancel out the negated third column in the projection matrix. If your application only needs correct geometry this could work fine, but the normal vectors' z-component will likely be reversed, causing lighting to be off.

^ | ⌄ • Reply • Share ›

ksimek Mod • 5 years ago

Koshy George provides a simpler derivation when skew is zero: http://kgeorge.github.io/20...

^ | ⌄ • Reply • Share ›

Max • 5 years ago

Hi,

I do slam, thus I wanted to display the map on top of the video.
I tried both your solution and the one of
http://strawlab.org/2011/11...
I observed a slight offset between map's points and their image location with your approach while there's none (at least I don't see it) with the one from strawlab.
Either way, thank you very much for sharing!

^ | ⌄ • Reply • Share ›

ksimek Mod → Max • 5 years ago

Hi Max, thanks for the feedback. Which formula are you using? If you're using the matrix from the section "Equivalence to glFrustum", you'll definitely see an offset, because the principal point offset isn't included.

If you're using the earlier formula that uses the full matrix, I'd be interested to find out the source the discrepancy. Roughly how large is the offset you're seeing? 1-5 pixels? 10-50? 100-500? Maybe it's possible you haven't negated your principal point offset parameters (i.e. x_0 and y_0)?

Straw Lab's implementation is convenient, because it handles the most common case: a "Hartley and Zisserman" camera, and a 2D coordinate system centered at either the top-left or bottom right of the image. The approach I've described is slightly more general, but has several places for discrepancies to creep in, including the choice of intrinsic/extrinsic decomposition, the correction for differing 3D and 2D coordinate systems, and the choice of glOrtho parameters.

Of course, an error in my derivation is always a possibility too, and if this is the case, I'm eager to find it and correct it. I've had success with this approach in my projects, but I have far from tested it under all possible scenarios.

When I have some time, I'll work the math out and try to see where Straw Lab's equation and mine differ. If you come into any further insights on this issue, please do share! And if you're feeling extra generous and wanted to share some test code that reproduces the error, I'd be very grateful. Just contact me through the form on the "contact" page.

Thanks again for your comments,

Kyle

^ | ⌄ • Reply • Share ›

Max → ksimek • 5 years ago

Ah! My bad I overlooked the "equivalence to glfrustum" part, thus I didn't see the non-zero principal point offset handling.
Sorry for the wrong alert.
Thanks again.

^ | ⌄ • Reply • Share ›

x • 6 years ago

Pretty cool, thanks for the work.

^ | ⌄ • Reply • Share ›

ALSO ON **SIGHTATIONS**

Compiling ELSD (Ellipse and Line Segment Detector) on OS X

4 comments • 5 years ago

Avatar **philip andrew** — Do you know anything better than elsd now?

Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix

25 comments • 6 years ago

Avatar **Ben Hur Nascimento** — Thank you very much for this post, it helped me a lot on a computional vision project.

Q & A: Recovering pose of a calibrated camera - Algebraic vs. Geometric method?

1 comment • 4 years ago

Avatar **dissertation writing service** — Learning this 3D points projections is ideal in math subject. It gives the easy way to determine the …

Dissecting the Camera Matrix, Part 2: The Extrinsic Matrix

16 comments • 6 years ago

Avatar **Rajesh** — Hi, I am looking for a method to compute homography matrix between images(left & right) of stereo camera using …