augmented reality - computing the OpenGL projection matrix from intrinsic camera parameters

Andrew Straw • 05 Nov 2011 • Vienna

Summary

Here I describe how the Hartley-Zisserman (http://www.robots.ox.ac.uk/%7Evgg/hzbook/) (HZ) pinhole camera model differs from the OpenGL display pipeline and how to build an OpenGL projection matrix directly from the intrinsic camera parameter matrix of HZ. What is particular about this exposition is that I do this by calculating the matrix elements directly from the camera parameters rather than calling the glProjection() function. This allows for a more general camera model including pixels with skew. I also include the algebraic derivation (as a sympy (http://sympy.org) script) so you can follow my logic exactly.

Update (30 Jan 2013): The various implementations of the coordinate transform pipeline used in this post are available here (https://github.com/strawlab/opengl-hz).

Update (26 Dec 2014): I added a few more recent references at the end.

Note about image coordinates

In both OpenGL window coordinates and HZ image coordinate systems, (0,0) is the lower left corner with X and Y increasing right and up, respectively. In a normal image file, the (0,0) pixel is in the upper left corner. We have code paths to deal with this in one of two ways: first, we can draw our images upside down, so that all the pixel-based coordinate systems are the same. This is the code path used when "window_coords='y up'". Second, we can keep the images right side up and modify the projection matrix so that OpenGL will generate window coordinates that compensate for the flipped image coordinates. In this "window_coords='y down'" path, the generated OpenGL Y window coordinates are (height-y).

The OpenGL projection matrix from HZ intrinsic parameters

Enough of the preliminaries. We calculate the OpenGL Projection matrix when window_coords=='y up' to be:

```
[2*K00/width,  -2*K01/width,   (width - 2*K02 + 2*x0)/width,                          0]
[          0, -2*K11/height, (height - 2*K12 + 2*y0)/height,                          0]
[          0,             0, (-zfar - znear)/(zfar - znear), -2*zfar*znear/(zfar - znear)]
[          0,             0,                             -1,                          0]
```

With window_coords=='y down', we have:

```
[2*K00/width, -2*K01/width,    (width - 2*K02 + 2*x0)/width,                          0]
[          0, 2*K11/height, (-height + 2*K12 + 2*y0)/height,                          0]
[          0,            0,  (-zfar - znear)/(zfar - znear), -2*zfar*znear/(zfar - znear)]
[          0,            0,                             -1,                          0]
```

Where Knm is the (n,m) entry of the 3x3 HZ instrinsic camera calibration matrix K. (K is upper triangular and scaled such that the lower-right entry is one.) Width and height are the size of the camera image, in pixels, and x0 and y0 are the camera image origin and are normally zero. Znear and zfar are the standard OpenGL near and far clipping planes, respectively.

This is the cut-and-paste output of our sympy script projection_math.py (https://gist.github.com/1341472#file_projection_math.py). The approach is that we enter the operations of OpenGL vertex transformation pipeline and the HZ projection model into sympy, a computer algebra system (CAS). We have sympy solve for the OpenGL projection matrix so that the resulting pixel coordinate is the same for both the HZ camera model and the OpenGL pipeline. See the script for the implementation details. Of course this could be done by hand but is tedious and prone to mistakes.

Comparison with the OpenCV camera calibration

Although I have not directly used OpenCV for camera calibration (http://opencv.willowgarage.com/documentation/cpp/camera_calibration_and_3d_reconstruction.html), their parameterization of the pinhole camera is a subset of the full HZ model. In particular, their matrix A corresponds exactly to the HZ matrix K with pixel skew fixed at zero. Consequently, this page can be directly used with OpenCV camera calibrations by setting K01 to zero.

Experimental verification

To verify that this computation of the OpenGL projection matrix accurately captures the HZ camera model, we have calculated the projection of vertices into image coordinates three ways:

1. A CPU-based implementation of the HZ camera model. This performs matrix multiplication of the eye coordinates by the intrinsic parameter matrix K.
2. A CPU-based emulation of the OpenGL pipeline. This performs matrix multiplication of the eye coordinates by the OpenGL projection matrix to produce clip coordinates, transforming those to normalized device coordinates, and then finally using the glViewport parameters to establish the window coordinates.
3. Direct calls to the OpenGL pipeline, presumably running on your GPU. In this case, we directly load our OpenGL projection matrix by calling glLoadMatrixf() and use OpenGL to perform all vertex transformation.

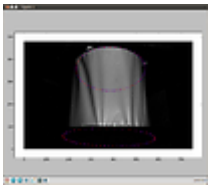
The first two examples are in the calib_test_numpy.py (https://gist.github.com/1341472#file_calib_test_numpy.py) example and their outputs are overlaid. The third (OpenGL) example is in calib_test_pyglet.py (https://gist.github.com/1341472#file_calib_test_pyglet.py). Each

of these three examples gives the same results, suggesting that our formulation is correct. All calculations were done with Python scripts, but the concepts and results should be easily adaptable to any language.

Here is a brief conceptual walkthrough of these programs. Each program starts by loading an image acquired by a (crudely) calibrated camera. This image (luminance.png, included in the zip download below) is of a roughly cylindrical object 1 meter in diameter and 1 meter high. Part of the cylinder is occluded and only the front surface is illuminated. The camera calibration is contained within the file cameramatrix.txt (https://gist.github.com/1341472#cameramatrix.txt). This calibration is decomposed into the intrinsic camera parameters and the rotation matrix, and the camera translation vector. A simple mathematical model of the cylinder is used to generate world coordinates of many vertices. Each of these world coordinates is transformed to camera coordinates -- also called eye coordinates in OpenGL. This is done using the extrinsic camera parameters specifying the camera's pose and is done either as a matrix multiplication with the extrinsic parameter matrix or by loading them into the OpenGL modelview matrix. From there, each of these vertices in eye coordinate is transformed to window coordinates using the methods described above.

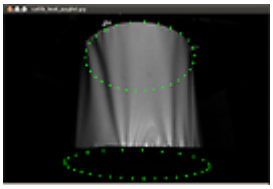
Output of CPU implementations

The blue crosses are the vertices after the HZ transformation. The red points are the vertices after the simulated OpenGL pipeline. Calculations done with numpy and scipy, and plotting done with matplotlib.


(/assets/calib_test_numpy.png)

Output of OpenGL implementation

The green points are the certices after the OpenGL pipeline. OpenGL called through pyglet.


(/assets/calib_test_pyglet.png)

Download these files

In addition to the individual scripts linked above, all the files are included in this zip file (/assets/augmented-reality-with-OpenGL.zip).

Remaining work: compensating for camera distortion not described by the pinhole model

You may want to implement the non-linear warping distortions (radial distortion, tangential distortion) that are often used to extend the pinhole model to be more realistic. A GLSL-based shader implementation of warping and de-warping may be the subject of a future post, but is not described here.

References

- A nice page on the OpenGL projection matrix (http://www.songho.ca/opengl/gl_projectionmatrix.html) by Song Ho Ahn
- A good description of the pinhole camera model (http://www.epixea.com/research/multi-view-coding-thesisse8.html) by Yannick Morvan
- Another description of the pinhole camera model mostly compatible with the terminology here (http://en.wikipedia.org/wiki/Camera_resectioning) at Wikipedia.
- The OpenGL specifications (http://www.opengl.org/documentation/specs/)
- Carlo Nicolini's blog post (https://braintrekking.wordpress.com/2013/06/02/a-c-code-to-compute-opengl-4x4-gl_modelview_matrix-from-2d-3d-points-homography/) and GitHub repository (https://github.com/CarloNicolini/OpenGL-CameraCalibration) with similar math and C++ code.
- A C++ implementation of the OpenGL matrix math (http://jamesgregson.blogspot.ca/2011/11/4x4-transformation-matching-opengl.html) by James Gregson, also on GitHub (https://github.com/jamesgregson/transformation).
- Another description of using calibrated cameras with OpenGL (http://ksimek.github.io/2013/06/03/calibrated_cameras_in_opengl/).

ARMK • 3 years ago

Amazing post. I tried implementing the code, it worked. But, this implementation is giving me huge offset in x-direction. Can it be the issue of frame buffer size and texture (image) mismatch, because in any case we can't control frame buffer size I guess.

^ v • Reply • Share ›

Mehran Maghoumi ➚ ARMK • 3 years ago

Weird... I'm experiencing the exact issue here: I have relatively large X offsets and I can't figure out why. I'm doing all the computations on CPU. Did you figure out what the problem was? Does anybody know what's wrong?

^ v • Reply • Share ›

Cameron Lowell Palmer • 4 years ago

What if the camera image aspect ratio and the screen aspect ratio don't match. You'll end up with a weird stretching. So if you correct the projection matrix to keep the correct aspect ratio of the camera background it won't line up with the projection matrix for the camera. Is there a way to reconcile the screen projection requirements with the camera?

astraw Mod ➚ Cameron Lowell Palmer • 4 years ago

In the above, width and height are from your call to glViewport(). There shouldn't be any aspect ratio trouble at all if you set those correctly.

Sam • 4 years ago

This is working great. But may be I am making a mistake somewhere. When I am placing a point light at positions (x, y, z,1.0) where z>0, lighting is not working. I need some help.

Max • 5 years ago

Works great, thanks!

Carlo • 6 years ago

I hope to be useful to the community posting my C++ code to compute homography and transform projection matrix from Hartley-Zisserman convention to OpenGL GL_PROJECTION and GL_MODELVIEW matrices:

https://github.com/CarloNic...

I've used your notes and the HZ book to write my code, I hope is simple enough to understand everything.

alberto • 6 years ago

the first link "A nice page on camera models and OpenGL by Wonwoo Lee"
is broken

astraw Mod ➚ alberto • 6 years ago

Thanks for the note. I deleted the link which referenced a page no longer there. There is an archive of the old page here http://web.archive.org/web/... but I cannot get it to render properly.

Ken_kagoshima • 6 years ago

I also think 2*K11/height should be for 'y up'. Actually i got the expected image with that. A possibility might be the difference how to treat the extrinsics.

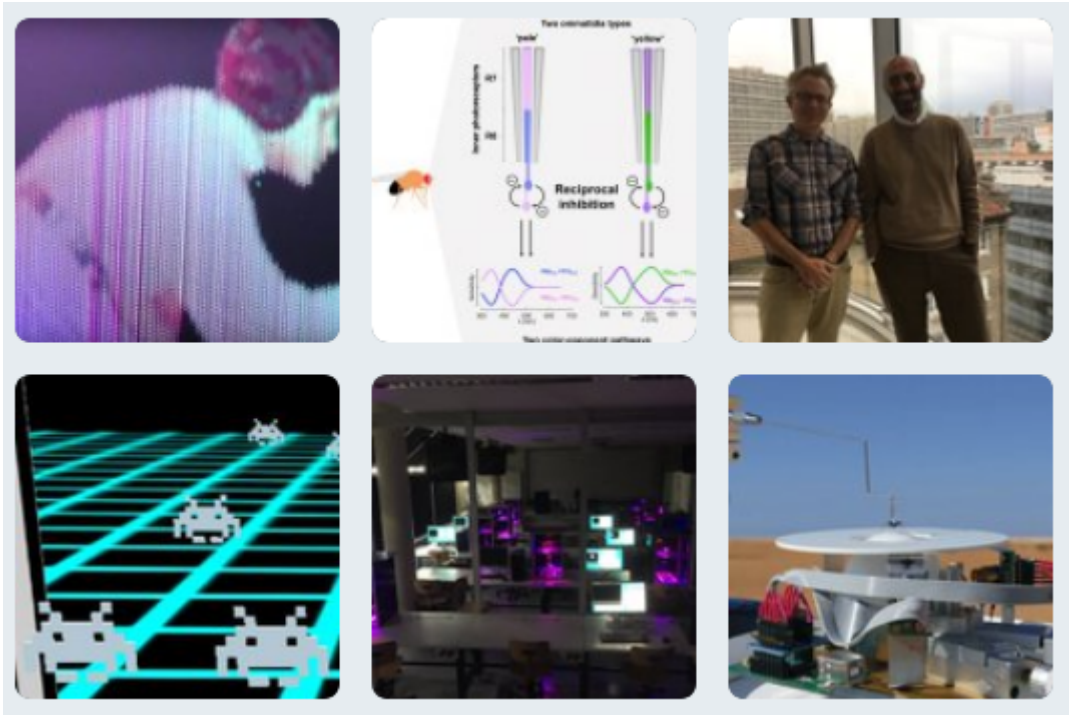
(https://www.bio1.uni-freiburg.de/tierphys-en)

Follow us on Twitter

Andrew Straw

@strawlab

animal behavior, neural circuits, optogenetics, virtual reality, cameras, vision, Drosophila, software



(https://twitter.com/strawlab)