Dissecting the Camera Matrix, Part 2: The Extrinsic Matrix

August 22, 2012

Welcome to the third post in the series "The Perspecive Camera - An Interactive Tour." In the last post, we learned how to decompose the camera matrix into a product of intrinsic and extrinsic matrices. In the next two posts, we'll explore the extrinsic and intrinsic matrices in greater detail. First we'll explore various ways of looking at the extrinsic matrix, with an interactive demo at the end.

The Extrinsic Camera Matrix

The camera's extrinsic matrix describes the camera's location in the world, and what direction it's pointing. Those familiar with OpenGL know this as the "view matrix" (or rolled into the "modelview matrix"). It has two components: a rotation matrix, R, and a translation vector $\boldsymbol{t}$, but as we'll soon see, these don't exactly correspond to the camera's rotation and translation. First we'll examine the parts of the extrinsic matrix, and later we'll look at alternative ways of describing the camera's pose that are more intuitive.

The extrinsic matrix takes the form of a rigid transformation matrix: a 3x3 rotation matrix in the left-block, and 3x1 translation column-vector in the right:

$$[R\,|\,\boldsymbol{t}] = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{array}\right]$$

It's common to see a version of this matrix with extra row of (0,0,0,1) added to the bottom. This makes the matrix square, which allows us to further decompose this matrix into a rotation *followed by* translation:

$$\left[\begin{array}{c|c} R & \boldsymbol{t} \\ \hline \boldsymbol{0} & 1 \end{array}\right] = \left[\begin{array}{c|c} I & \boldsymbol{t} \\ \hline \boldsymbol{0} & 1 \end{array}\right] \times \left[\begin{array}{c|c} R & \boldsymbol{0} \\ \hline \boldsymbol{0} & 1 \end{array}\right]$$

$$= \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \times \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & 0 \\ r_{2,1} & r_{2,2} & r_{2,3} & 0 \\ r_{3,1} & r_{3,2} & r_{3,3} & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

This matrix describes how to transform points in world coordinates to camera coordinates. The vector $\boldsymbol{t}$ can be interpreted as the position of the world origin in camera coordinates, and the columns of R represent represent the directions of the world-axes in camera coordinates.

The important thing to remember about the extrinsic matrix is that it describes how the *world* is transformed relative to the *camera*. This is often counter-intuitive, because we usually want to specify how the *camera* is transformed relative to the *world*. Next, we'll examine two alternative ways to describe the camera's extrinsic parameters that are more intuitive and how to convert them into the form of an extrinsic matrix.

Building the Extrinsic Matrix from Camera Pose

It's often more natural to specify the camera's pose directly rather than specifying how world points should transform to camera coordinates. Luckily, building an extrinsic camera matrix this way is easy: just build a rigid transformation matrix that describes the camera's pose and then take it's inverse.

Let *C* be a column vector describing the location of the camera-center in world coordinates, and let R_c be the rotation matrix describing the camera's orientation with respect to the world coordinate axes. The transformation matrix that describes the camera's pose is then $\left[R_c \mid C\right]$. Like before, we make the matrix square by adding an extra row of (0,0,0,1). Then the extrinsic matrix is obtained by inverting the camera's pose matrix:

$$
\begin{aligned}
\left[\begin{array}{c|c} R & \boldsymbol{t} \\ \hline \mathbf{0} & 1 \end{array}\right] &= \left[\begin{array}{c|c} R_c & C \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1} \\
&= \left[\left[\begin{array}{c|c} I & C \\ \hline \mathbf{0} & 1 \end{array}\right]\left[\begin{array}{c|c} R_c & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]\right]^{-1} && \text{(decomposing rigid transform)} \\
&= \left[\begin{array}{c|c} R_c & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1}\left[\begin{array}{c|c} I & C \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1} && \text{(distributing the inverse)} \\
&= \left[\begin{array}{c|c} R_c^T & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]\left[\begin{array}{c|c} I & -C \\ \hline \mathbf{0} & 1 \end{array}\right] && \text{(applying the inverse)} \\
&= \left[\begin{array}{c|c} R_c^T & -R_c^T C \\ \hline \mathbf{0} & 1 \end{array}\right] && \text{(matrix multiplication)}
\end{aligned}
$$

When applying the inverse, we use the fact that the inverse of a rotation matrix is it's transpose, and inverting a translation matrix simply negates the translation vector. Thus, we see that the relationship between the extrinsic matrix parameters and the camera's pose is straightforward:

$$
\begin{aligned}
R &= R_c^T \\
\boldsymbol{t} &= -RC
\end{aligned}
$$

Some texts write the extrinsic matrix substituting -*RC* for ***t***, which mixes a world transform (*R*) and camera transform notation (*C*).

The "Look-At" Camera

Readers familiar with OpenGL might prefer a third way of specifying the camera's pose using *(a)* the camera's position, *(b)* what it's looking at, and *(c)* the "up" direction. In legacy OpenGL, this is accomplished by the gluLookAt() function, so we'll call this the "look-at" camera. Let *C* be the camera center, ***p*** be the target point, and ***u*** be up-direction. The algorithm for computing the rotation matrix is (paraphrased from the OpenGL documentation):

1. Compute L = p - C.
2. Normalize L.
3. Compute s = L x u. (cross product)

4. Normalize s.
5. Compute u' = s x L.

The extrinsic rotation matrix is then given by:

$$R = \begin{bmatrix} s_1 & s_2 & s_3 \\ u'_1 & u'_2 & u'_3 \\ -L_1 & -L_2 & -L_3 \end{bmatrix}$$

(Updated May 21, 2014 -- transposed matrix)

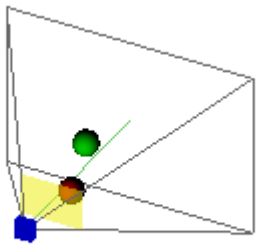
You can get the translation vector the same way as before, $\boldsymbol{t} = -RC$.

Try it out!

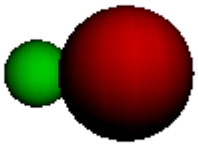
Below is an interactive demonstration of the three different ways of parameterizing a camera's extrinsic parameters. Note how the camera moves differently as you switch between the three parameterizations.

This requires a WebGL-enabled browser with Javascript enabled.

Scene



Image



Left: scene with camera and viewing volume. Virtual image plane is shown in yellow. *Right*: camera's image.

- Extrinsic (World)
- Extr. (Camera)
- Extr. ("Look-at")
- Intrinsic

$\boldsymbol{t}_x$
$\boldsymbol{t}_y$
$\boldsymbol{t}_z$
x-Rotation
y-Rotation
z-Rotation

Adjust extrinsic parameters above.

This is a "world-centric" parameterization. These parameters describe how the *world* changes relative to the *camera*. These parameters correspond directly to entries in the extrinsic camera matrix.

As you adjust these parameters, note how the camera moves in the world (left pane) and contrast with the "camera-centric" parameterization:

- Rotating affects the camera's position (the blue box).
- The direction of camera motion depends on the current rotation.
- Positive rotations move the camera clockwise (or equivalently, rotate the world counter-clockwise).

Also note how the image is affected (right pane):

- Rotating never moves the world origin (red ball).
- Changing t_x always moves the spheres horizontally, regardless of rotation.
- Increasing t_z always moves the camera closer to the world origin.

C_x
C_y
C_z
x-Rotation
y-Rotation
z-Rotation

Adjust extrinsic parameters above.

This is a "camera-centric" parameterization, which describes how the *camera* changes relative to the *world*. These parameters correspond to elements of the *inverse* extrinsic camera matrix.

As you adjust these parameters, note how the camera moves in the world (left pane) and contrast with the "world-centric" parameterization:

- Rotation occurs about the camera's position (the blue box).
- The direction of camera motion is independent of the current rotation.
- A positive rotation rotates the camera counter-clockwise (or equivalently, rotates the world clockwise).
- Increasing C_y always moves the camera toward the sky, regardless of rotation.

Also note how the image is affected (right pane):

- Rotating around y moves both spheres horizontally.
- With different rotations, changing C_x moves the spheres in different directions.

C_x
C_y
C_z
p_x
p_y
p_z

Adjust extrinsic parameters above.

This is a "look-at" parameterization, which describes the camera's orientation in terms of what it is looking at. Adjust p_x, p_y, and p_z to change where the camera is looking (orange dot). The up vector is fixed at (0,1,0)'. Notice that moving the camera center, *C*, causes the camera to rotate.

Focal Length
Axis Skew
x_0
y_0

Adjust intrinsic parameters above. As you adjust these parameters, observe how the viewing volume changes in the left pane:

- Changing the focal length moves the yellow focal plane, which chainges the field-of-view angle of the viewing volume.
- Changing the principal point affects where the green center-line intersects the focal plane.
- Setting skew to non-zero causes the focal plane to be non-rectangular

Intrinsic parameters result in 2D transformations only; the depth of objects are ignored. To see this, observe how the image in the right pane is affected by changing intrinsic parameters:

- Changing the focal length scales the near sphere and the far sphere equally.
- Changing the principal point has no affect on parallax.
- No combination of intrinsic parameters will reveal occluded parts of an object.

Conclusion

We've just explored three different ways of parameterizing a camera's extrinsic state. Which parameterization you prefer to use will depend on your application. If you're writing a Wolfenstein-style FPS, you might like the world-centric parameterization, because moving along (t_z) always corresponds to walking forward. Or you might be interpolating a camera through waypoints in your scene, in which case, the camera-centric parameterization is preferred, since you can specify the position of your camera directly. If you aren't sure which you prefer, play with the tool above and decide which approach feels the most natural.

Join us next time when we explore the intrinsic matrix, and we'll learn why hidden parts of your scene can never be revealed by zooming your camera. See you then!

Posted by Kyle Simek

Calibrated Cameras in OpenGL without glFrustum → ← Dissecting the Camera Matrix, Part 1: Extrinsic/Intrinsic Decomposition

pn = right image pixel position projected to left image(output)

L * pn = KL * R * KRi * PR --------- (1)

There was some suggestion of modifying homography matrix (H)

H = KL * R * KRi; (2)

see more

• Reply • Share ›

jeremiah johnson • 2 years ago

oh this makes my head hurt. all i need is the location and rotation of the camera in object space rather than vice versa *brain explodes*

• Reply • Share ›

Mihir Danecha • 2 years ago

Hey....i am working with creating 3d from multiple images.
is it right equation to find second camera matrix from relative rotation and translation from first refrence identity matrix with zero translation vector.
here part of code:

R2 = R1 * R;
T2 = T1 + (R1.t() * t);

R- relative rotation
t - relative translation
R1- Rotation first camera matrix (identity matrix taken)
T1 - Transaltion First camera matrix (zeros)
R2 - second camera rotarion matrix to be calculated
T2 - second camera translation matrix to be calculated

Please guys help me with this...
Thanks in advance.

• Reply • Share ›

essay writers online • 3 years ago

Learning this topic about Extrinsic Matrix is really important for those people who are using a camera to portray a good output on their work. Also, they can be able to gain a good technique that will help them build some new tool to achieve their desires for their work.

• Reply • Share ›

patmeansnoble • 4 years ago

So I'm doing a 3D reconstruction and I need to find the world coordinates X,Y, Z from a 2D pixel. I'm using a formula similar to the one [here](http://stackoverflow.com/qu... . I know everything in that formula save `s` and the world point `(X,Y,Z)`. Generally, they say you pick a point in the world? How do you find the values of s and each of (X,Y,Z)? I thought of Gaussian elimination, but then that leaves me having to assume a Z distance away from the camera center. Will be happy to know what you think.

• Reply • Share ›

ksimek Mod ➚ patmeansnoble • 4 years ago

I'm not sure I understand the question exactly, but if you want to recover world coordinates from 2D pixels, it isn't possible to estimate depth from a 2D pixel without additional information like a second view. If all you want is a point at an arbitrary depth, just pick a value for s or z and use method 1 or 2 at the link you provided to convert from camera coordinates to world coordinates.

^ v • Reply • Share ›

patmeansnoble ➚ ksimek • 4 years ago

Thanks for replying. At this point I am doing a point at an arbitrary depth.
I found a (seemingly easy) formula in Zisserman's book (eqn 8.1, p196) where the homography is computed from the K, R and T matrices as
H = K[r1, r2, t] ,
where K is the camera intrinsic matrix, r1 and r2 are the first two columns of R and t is the translation vector. Couldn't I just do the projection as follows:

p = [u v 1]; // where u, are the 2D pixel coordinates
proj = H*p; // project
p= proj /t3; // normalization

?

^ v • Reply • Share ›

ksimek Mod ➚ patmeansnoble • 4 years ago

The passage you quoted isn't quite what you're looking for. You want backprojection, not projection, and you probably want to use the full camera matrix, not just a homography.

Check out equations 6.13 and 6.14 in H&Z for the formula for backprojection. To paraphrase, let [M | p4] = K * [R | t] be the 3x4 camera matrix with the 3x3 submatrix M and column vector p4. The backprojection equation is then,

P = C + lambda* inv(M) * p.

Here, p is the 2D homogenous column vector [u v 1]', C is the camera center in nonhomogeneous world coordinates (given in homogeneous coordinates by C = -inv(M) p4), and lambda is an arbitrary scalar that controls the depth. This will give the nonhomogeneous 3D point P. To backproject to an exact depth d, set lambda = d * ||m3|| where m3 is the third row of M (I'm speaking of the top of my head and haven't tested this).

^ v • Reply • Share ›

HSKapasi ➚ ksimek • 2 years ago

Hi
Thanks for excellent article.

I have a question on Homographic equation.

1): Can I not just take inverse of Homography and compute the Real World X,Y coordinates?

I tried inverting Homography and it gives me exact world cooridnates I had used to estimate Homography.

I also tried to use the fomula 6.14 from H&Z, but back projection is not exac. When I compared Homography Matrix I get from Open CV using a single image and product of K[R1 R2t] with Z=0, which I get from Matlab, they do not match up. I tried to compare the scales also but there is no single scale that maps two.
< This is fixed. We can breakdown Homgraphy into sub matrices>

2):So, my question is, am I making any error is estimating Homography

see more

• Reply • Share ›

patmeansnoble ksimek • 4 years ago

OK, cool. I have tested this formula but got confused as to what lambda is. X(lambda) is the equation of a line. P+ seems to be the slope and lambda*C would correspond to the intercept. They didn't quite specify lambda in the text. My initial guess was it was a count of iterations of 2D points we are projecting until I read your comment. But based on my thinking above, it seems to be a scale factor for how the (u,v) points changes in the image. So my questions:

(i) How did you arrive at lambda = d * ||m3|| if I may ask? I get the last row of M as m3 part. What about d? The (X, Y, Z) which we do not yet know?

(ii) And the (u,v) points, I do not need to correct for radial distortion and other scaling in the retrieved points since the camera is already calibrated right? I'm talking about item 2 in the question here:

http://stackoverflow.com/qu...

• Reply • Share ›

ksimek Mod patmeansnoble • 4 years ago

Hartley and Zisserman's formulation in (6.13) is slightly different than mine, but they are equivalent. They use homogeneous coordinates, so they place lambda on C, not M. They use the pseudoinverse of P, which handles degenerate cameras. Since we have a finite camera, you can use (6.14), which is the formula I used.

(i) Recall [M | p4] = K [R | t], so M = KR. Since we're operating in homogeneous coordinates, we can multiply M by any constant c and have the same camera: cM = (cK)R. It is nice if (cK) has a 1 in element (3,3), because it leaves the camera z-coordinate unchanged, which means it will also preserves z-lengths in the backprojection equation. So we choose c = 1/K(3,3). Recall that K is diagonal, and the rows of R are unit vectors, so K(3,3) = ||m3||. Thus, c = 1/||m3||. Plugging that into the backprojection equation we get lambda inv(M/||m3||) p = lambda ||m3|| inv(M). Using p = (u,v,1)' means that ||m3|| inv(M) p will have a distance of 1 along the

camera's z-coordinate. Scaling by d with adjust this distance. I just rolled d and ||m3|| into lambda = d ||m3||. The resulting P is the (X,Y,Z) we're looking for.

(ii) Yes, all discussion here assumes you've already corrected for nonlinear distortion.

Reply • Share ›

patmeansnoble • 4 years ago

Nice post!
You probably need to fix the OpenGL documentation page. The link seems broken :(

Reply • Share ›

ksimek Mod ➔ patmeansnoble • 4 years ago

Thanks, fixed!

Reply • Share ›

Student • 5 years ago

Hi, great tutorial and great explanations. I just have a question.

Isn't your R matrix in the lookAt section inverted. The matrix you give sends the world coordinates to the camera coordinates :

R(1 0 0) = s, R(0 1 0)=u, R(0 0 1) = -L.
If we have the coordinates of the object in camera reference, then this matrix gives us the coordinates in world reference.

But what we want here is to send the camera coordinates to the world coordinates. that means taking the transpose of the matrix you gave.

To verify this, just suppose that t = (0,0,0) (for simplification) and that the axes of the camera and world are not aligned. we have a point P = (1 0 0) in world coordinates, we wont to know its coordinates relative to the camera. Multiplying by R(1 0 0) = s which is obviously not the right coordinates (do a little drawing for that). While R^T(1 0 0) = (s1 u1 -L1), which is the projection of the (1 0 0) vector on the camera axes, and that what we want.

Again I thank you for the great resources you put online and sorry for my bad english

Reply • Share ›

ksimek Mod ➔ Student • 5 years ago

You are absolutely correct, thank you. I've edited the post above.

I foolishly used the matrix from the IBM OpenGL documentation, which shows the matrix in transposed form. They never explain why, but I assume they have chosen to display the camera's transformation relative to the world coordinates, which is the inverse of the matrix actually sent to OpenGL. This is a confusing documentation choice when the goal is to describe how the math is implemented, and luckily it is a choice that isn't repeated in the official OpenGL documentation.

I've left the IBM link in the article for attribution purposes, as I have paraphrased its derivation of the basis vectors L, s, and u, but hopefully this comment will serve as an

explanation for the astute reader who notices our matrices differ.

Thank you again for your sharp eye and your clear argument. After nearly 2 years and over 5,000 page views, you are the first to point out this glaring mistake!

• Reply • Share ›

Anonymous • 6 years ago

Nice explanation!

• Reply • Share ›

ALSO ON **SIGHTATIONS**

The Perspective Camera - An Interactive Tour

7 comments • 6 years ago


Avatar **chengleilei** — Thank you very much! This helped me a lot!

Dissecting the Camera Matrix, Part 1: Extrinsic/Intrinsic Decomposition

6 comments • 6 years ago


Avatar **csukuangfj** — yes, you're correct.

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?

Calibrated Cameras in OpenGL without glFrustum

21 comments • 6 years ago


Avatar **ksimek** — Thanks for the clarifying photos. Sorry, I misunderstood the scenario -- I thought your calibration image was larger …