How do I reverse-project 2D points into 3D?

Asked 16 years, 3 months ago Modified 4 years, 1 month ago

Viewed 65k times

I have 4 2D points in screen-space, and I need to reverse-project them back into 3D space. I know that each of the 4 points is a corner of a 3D-rotated rigid rectangle, and I know the size of the rectangle. How can I get 3D coordinates from this?

I am not using any particular API, and I do not have an existing projection matrix. I'm just looking for basic math to do this. Of course there isn't enough data to convert a single 2D point to 3D with no other reference, but I imagine that if you have 4 points, you know that they're all at right-angles to each other on the same plane, and you know the distance between them, you should be able to figure it out from there. Unfortunately I can't quite work out how though.

This might fall under the umbrella of photogrammetry, but google searches for that haven't led me to any helpful information.

3d geometry 2d photogrammetry reverseprojection

Share Follow

edited Feb 17, 2009 at 17:02

Neil N

25.3k ● 17 ● 86 ● 146

asked Sep 16, 2008 at 19:42

Joshua Carmody

13.7k ● 16 ● 67 ● 84

If I'm not mistaken, you could have two options for 3D coordinates. – Omer van Kloeten Sep 16, 2008 at 19:46

And the question, unclear as it is, implies perspective (so the larger the z distance, the smaller the projection) – tzot Sep 19, 2008 at 11:13

Hello Joshua, I've been struggling with a lot as well. Did you ever find out a satisfying solution ? Thanks, T – Theo.T Mar 1, 2009 at 17:29

Not yet Theo. The project I was working on has kinda been put on the back burner, but I will eventually need to figure this out. I was considering starting a bounty. – Joshua Carmody Mar 2, 2009 at 6:14

Hi Joshua, any luck yet? I'd very much like to see the code that solved it if you did please! :D – CodeAndCats Sep 7, 2009 at 8:42

13 Answers

Sorted by: Highest score (default)

107

Alright, I came here looking for an answer and didn't find something simple and straightforward, so I went ahead and did the dumb but effective (and relatively simple) thing: Monte Carlo optimisation.



Very simply put, the algorithm is as follows: Randomly perturb your projection matrix until it projects your known 3D coordinates to your known 2D coordinates.

Here is a still photo from Thomas the Tank Engine:

Let's say we use GIMP to find the 2D coordinates of what we think is a square on the ground plane (whether or not it is really a square depends on your judgment of the depth):

I get four points in the 2D image: `(318, 247)`, `(326, 312)`, `(418, 241)`, and `(452, 303)`.

By convention, we say that these points should correspond to the 3D points: `(0, 0, 0)`, `(0, 0, 1)`, `(1, 0, 0)`, and `(1, 0, 1)`. In other words, a unit square in the y=0 plane.

Projecting each of these 3D coordinates into 2D is done by multiplying the 4D vector `[x, y, z, 1]` with a 4x4 projection matrix, then dividing the x and y components by z to actually get the perspective correction. This is more or less what gluProject() does, except `gluProject()` also takes the current viewport into account and takes a separate modelview matrix into account (we can just assume the modelview matrix is the identity matrix). It is very handy to look at the

`gluProject()` documentation because I actually want a solution that works for OpenGL, but beware that the documentation is missing the division by z in the formula.

Remember, the algorithm is to start with some projection matrix and randomly perturb it until it gives the projection that we want. So what we're going to do is project each of the four 3D points and see how close we get to the 2D points we wanted. If our random perturbations cause the projected 2D points to get closer to the ones we marked above, then we keep that matrix as an improvement over our initial (or previous) guess.

Let's define our points:

```
# Known 2D coordinates of our rectangle
i0 = Point2(318, 247)
i1 = Point2(326, 312)
i2 = Point2(418, 241)
i3 = Point2(452, 303)

# 3D coordinates corresponding to i0, i1, i2, i3
r0 = Point3(0, 0, 0)
r1 = Point3(0, 0, 1)
r2 = Point3(1, 0, 0)
r3 = Point3(1, 0, 1)
```

We need to start with some matrix, identity matrix seems a natural choice:

```
mat = [
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
```

```
    [0, 0, 0, 1],
]
```

We need to actually implement the projection (which is basically a matrix multiplication):

```
def project(p, mat):
    x = mat[0][0] * p.x + mat[0][1] * p.y + mat[0]
[2] * p.z + mat[0][3] * 1
    y = mat[1][0] * p.x + mat[1][1] * p.y + mat[1]
[2] * p.z + mat[1][3] * 1
    w = mat[3][0] * p.x + mat[3][1] * p.y + mat[3]
[2] * p.z + mat[3][3] * 1
    return Point(720 * (x / w + 1) / 2., 576 - 576
* (y / w + 1) / 2.)
```

This is basically what `gluProject()` does, 720 and 576 are the width and height of the image, respectively (i.e. the viewport), and we subtract from 576 to count for the fact that we counted y coordinates from the top while OpenGL typically counts them from the bottom. You'll notice we're not calculating z, that's because we don't really need it here (though it could be handy to ensure it falls within the range that OpenGL uses for the depth buffer).

Now we need a function for evaluating how close we are to the correct solution. The value returned by this function is what we will use to check whether one matrix is better than another. I chose to go by sum of squared distances, i.e.:

```
# The squared distance between two points a and b
def norm2(a, b):
```

```
    dx = b.x - a.x
    dy = b.y - a.y
    return dx * dx + dy * dy

def evaluate(mat):
    c0 = project(r0, mat)
    c1 = project(r1, mat)
    c2 = project(r2, mat)
    c3 = project(r3, mat)
    return norm2(i0, c0) + norm2(i1, c1) +
norm2(i2, c2) + norm2(i3, c3)
```

To perturb the matrix, we simply pick an element to perturb by a random amount within some range:

```
def perturb(amount):
    from copy import deepcopy
    from random import randrange, uniform
    mat2 = deepcopy(mat)
    mat2[randrange(4)][randrange(4)] += uniform(-
amount, amount)
```

(It's worth noting that our `project()` function doesn't actually use `mat[2]` at all, since we don't compute z, and since all our y coordinates are 0 the `mat[*][1]` values are irrelevant as well. We could use this fact and never try to perturb those values, which would give a small speedup, but that is left as an exercise...)

For convenience, let's add a function that does the bulk of the approximation by calling `perturb()` over and over again on what is the best matrix we've found so far:

```
def approximate(mat, amount, n=100000):
    est = evaluate(mat)
```

```
    for i in xrange(n):
        mat2 = perturb(mat, amount)
        est2 = evaluate(mat2)
        if est2 < est:
            mat = mat2
            est = est2

    return mat, est
```

Now all that's left to do is to run it...:

```
for i in xrange(100):
    mat = approximate(mat, 1)
    mat = approximate(mat, .1)
```

I find this already gives a pretty accurate answer. After running for a while, the matrix I found was:

```
[
    [1.0836000765696232,  0,  0.16272110011060575,
-0.448110649351155 97],
    [0.09339193527789781, 1, -0.7990570384334473,
0.539087345090207  ],
    [0,                   0,  1,
0                  ],
    [0.06700844759602216, 0, -0.8333379578853196,
3.875290562060915  ],
]
```

with an error of around `2.6e-5`. (Notice how the elements we said were not used in the computation have not actually been changed from our initial matrix; that's because changing these entries would not change the result of the evaluation and so the change would never get carried along.)

We can pass the matrix into OpenGL using `glLoadMatrix()` (but remember to transpose it first, and remember to load your modelview matrix with the identity matrix):

```
def transpose(m):
    return [
        [m[0][0], m[1][0], m[2][0], m[3][0]],
        [m[0][1], m[1][1], m[2][1], m[3][1]],
        [m[0][2], m[1][2], m[2][2], m[3][2]],
        [m[0][3], m[1][3], m[2][3], m[3][3]],
    ]

glLoadMatrixf(transpose(mat))
```

Now we can for example translate along the z axis to get different positions along the tracks:

```
glTranslate(0, 0, frame)
frame = frame + 1

glBegin(GL_QUADS)
glVertex3f(0, 0, 0)
glVertex3f(0, 0, 1)
glVertex3f(1, 0, 1)
glVertex3f(1, 0, 0)
glEnd()
```

For sure this is not very elegant from a mathematical point of view; you don't get a closed form equation that you can just plug your numbers into and get a direct (and accurate) answer. HOWEVER, it does allow you to add additional constraints without having to worry about complicating your equations; for example if we wanted to incorporate height as well, we could use that corner of the house and say (in our evaluation function) that the distance from the ground to the roof should be so-and-so, and run the algorithm again. So yes, it's a brute force of sorts, but works, and works well.

Share Follow

edited Nov 30, 2015 at 8:59

answered Nov 28, 2015 at 21:39

Vegard

2,271 ● 1 ● 19 ● 27

12 Impressive visualisations! – Morgan Wilde Dec 2, 2015 at 2:49

16 Just a note for someone who will found this question/answer. In computer vision, this problem is called Perspective-n-Point (PnP). A computer vision library (OpenCV, Matlab, etc.) will probably have a function for this. Using a stochastic approach (Monte Carlo) could work but it exists already in the literature some methods to solve this kind of problem. – Catree May 10, 2017 at 11:16

I think your answer is wrong. There are many matrices, which map a "unit square" to your 2D points. But if you use your solution to get the 3D coordinates of a square in space, it will not be a square (no right angles, different edge length). – Ivan Kuckir Jun 8, 2022 at 19:15

@IvanKuckir It's because of the way he parameterized (or rather, did not parameterize) the matrix -- which was fine here because the 3D space is just an undesired intermediate anyways and the end goal is the 2D reprojection. But you can maintain the geometry in 3D by using a similar method but, instead of operating on the matrix directly, generate a transform from parameters that preserve the shape in 3D. For example, build a matrix from roll/pitch/yaw/translation, plus a second projection matrix from e.g. camera FOV. Then randomly perturb *those* params, and you'll always have a clean result. – Jason C Oct 25, 2022 at 3:40

(E.g. optimizing parameters to functions similar to glRotate + glTranslate + gluPerspective + glViewport, and performing the projection with the three resulting matrices, will yield reasonable 3D coordinates after the modelview transform, ambiguities in depth notwithstanding. Whereas, the method here of randomly perturbing the matrix elements will almost certainly lead to some nonorthogonal, nonhomogenous transform that you can't separate the frustum transform from). – Jason C Oct 25, 2022 at 3:45

10

This is the Classic problem for marker based Augmented Reality.

You have a square marker (2D Barcode), and you want to find its Pose (translation & rotation in relation to the camera), after finding the four edges of the marker.

Overview-Picture

I'm not aware of the latest contributions to the field, but at least up to a point (2009) RPP was supposed to outperform POSIT that is mentioned above (and is indeed a classic approach for this) Please see the links, they also provide source.

- http://www.emt.tugraz.at/~vmg/schweighofer
- http://www.emt.tugraz.at/publications/EMT_TR/TR-EMT-2005-01.pdf
- http://www.emt.tugraz.at/system/files/rpp_MATLAB_ref_implementation.tar.gz

(PS - I know it's a bit old topic, but anyway, the post might be helpful to somebody)

Share Follow

answered Dec 18, 2012 at 16:21

dim_tz

1,531 ● 5 ● 21 ● 38

I appreciate your answer! It may be an old question, but it's one that has never been answered satisfactorily. I'll take a look at those links. – Joshua Carmody Dec 18, 2012 at 18:12

5

D. DeMenthon devised an algorithm to compute the *pose* of an object (its position and orientation in space) from feature points in a 2D image when knowing the model of the object -- **this is your exact problem**:

> We describe a method for finding the pose of an object from a single image. We assume that we can detect and match in the image four or more noncoplanar feature points of the object, and that we know their relative geometry on the object.

The algorithm is known as **Posit** and is described in it classical article "Model-Based Object Pose in 25 Lines of Code" (available on its website, section 4).

Direct link to the article: http://www.cfar.umd.edu/~daniel/daniel_papersfordownload/Pose25Lines.pdf OpenCV implementation: http://opencv.willowgarage.com/wiki/Posit

The idea is to repeatedly approximating the perspective projection by a *scaled orthographic projection* until converging to an accurate pose.

Share Follow

answered Aug 24, 2010 at 8:38

Julien-L

5,416 3 35 51

5

For my OpenGL engine, the following snip will convert mouse/screen coordinates into 3D world coordinates. Read the commments for an actual description of what is going on.

```
/*   FUNCTION:        YCamera ::
CalculateWorldCoordinates
```

```
     ARGUMENTS:       x          mouse x coordinate
                      y          mouse y coordinate
                      vec        where to store
coordinates
     RETURN:          n/a
     DESCRIPTION:     Convert mouse coordinates
into world coordinates
*/
```

```
void YCamera :: CalculateWorldCoordinates(float x,
float y, YVector3 *vec)
{
    //  START
    GLint viewport[4];
    GLdouble mvmatrix[16], projmatrix[16];

    GLint real_y;
    GLdouble mx, my, mz;

    glGetIntegerv(GL_VIEWPORT, viewport);
    glGetDoublev(GL_MODELVIEW_MATRIX, mvmatrix);
    glGetDoublev(GL_PROJECTION_MATRIX,
projmatrix);

    real_y = viewport[3] - (GLint) y - 1;   //
viewport[3] is height of window in pixels
    gluUnProject((GLdouble) x, (GLdouble) real_y,
1.0, mvmatrix, projmatrix, viewport, &mx, &my,
&mz);

    /*  'mouse' is the point where mouse
projection reaches FAR_PLANE.
        World coordinates is intersection of
line(camera->mouse) with plane(z=0) (see LaMothe
306)

        Equation of line in 3D:
            (x-x0)/a = (y-y0)/b = (z-z0)/c

        Intersection of line with plane:
            z = 0
            x-x0 = a(z-z0)/c  <=> x = x0+a(0-z0)/c
```

```
<=> x = x0 -a*z0/c
           y = y0 - b*z0/c

     */
```

Share Follow

edited Nov 19, 2020 at 21:41

karlphillip

93.3k ● 37 ● 250 ● 438

answered Sep 16, 2008 at 22:42

user14208

59 ● 1

4

From the 2-D space there will be 2 valid rectangles that can be built. Without knowing the original matrix projection, you won't know which one is correct. It's the same as the "box" problem: you see two squares, one inside the other, with the 4 inside vertices connected to the 4 respective outside vertices. Are you looking at a box from the top-down or the bottom-up?

That being said, you are looking for a matrix transform T where...

{{x1, y1, z1}, {x2, y2, z2}, {x3, y3, z3}, {x4, y4, z4}} x T = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}}

(4 x 3) x T = (4 x 2)

So T must be a (3 x 2) matrix. So we've got 6 unknowns.

Now build a system of constraints on T and solve with Simplex. To build the constraints, you know that a line passing through the first two points must be parallel to the

line passing to the second two points. You know a line passing through points 1 and 3 must be parallel to the lines passing through points 2 and 4. You know a line passing through 1 and 2 must be orthogonal to a line passing through points 2 and 3. You know that the length of the line from 1 and 2 must equal the length of the line from 3 and 4. You know that the length of the line from 1 and 3 must equal the length of the line from 2 and 4.

To make this even easier, you know about the rectangle, so you know the length of all the sides.

That should give you plenty of constraints to solve this problem.

Of course, to get back, you can find T-inverse.

@Rob: Yes, there are an infinite number of projections, but not an infinite number of projects where the points must satisfy the requirements of a rectangle.

@nlucaroni: Yes, this is only solvable if you have four points in the projection. If the rectangle projects to just 2 points (i.e. the plane of the rectangle is orthogonal to the projection surface), then this cannot be solved.

Hmmm... I should go home and write this little gem. This sounds like fun.

Updates:

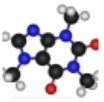
1. There are an infinite number of projections unless you fix one of the points. If you fix on of the points of

the original rectangle, then there are two possible original rectangles.

Share Follow

edited Sep 17, 2008 at 13:17

answered Sep 16, 2008 at 20:00



Jarrett Meyer
19.6k ● 6 ● 60 ● 52

@"Hmmm... I should go home and write this little gem. This sounds like fun." I'd be very grateful if you did! :-)
– Joshua Carmody Sep 16, 2008 at 20:14

I would like someone to explain how a given rectangle (say, a ticket) —with my eye stabilized and a transparent glass between it and the ticket— it can have infinite places while its corners project to the exact same 4 spots on the glass. Please. – tzot Sep 19, 2008 at 11:18

You sound like you were so close Jarrett! I don't think you really believe there are infinite projections. I mean if you hold up a piece of paper in front of you, there is only one position and orientation in 3d space you can hold it that it will appear exactly the same (ignoring flipping/rotating the paper). Furthermore, if you close your eyes and I hold up a piece of paper up in front of you, when you open your eyes you can reason pretty well where that paper is positioned in 3d space by only seeing the world as a 2d image and knowing the size of the paper. So surely math can do it too? – CodeAndCats Sep 22, 2009 at 6:11

Just to clarify, by "ignoring flipping/rotating the paper" I mean ignoring duplicate cases where you've just rotated the rectangular paper 180 degrees. – CodeAndCats Sep 22, 2009 at 6:14

2

Assuming that the points are indeed part of a rectangle, I'm giving a generic idea :

Find two points with max inter-distance: these most probably define a diagonal (exception: special cases where the rectangle is almost paralell to the YZ plane, left for the student). Call them A, C. Calculate the BAD, BCD

angles. These, compared to right angles, give you orientation in 3d space. To find out about z distance, you need to correlate the projected sides to the known sides, and then, based on the 3d projection method (is it 1/z?) you're on the right track to know distances.

Share Follow

answered Sep 16, 2008 at 19:58

tzot

95.8k ● 30 ● 149 ● 208

2

To follow up on Rons approach: You can find your z-values if you know how you've rotated your rectangle.

The trick is to find the projective matrix that did the projection. Fortunately this is possible and even cheap to do. The relevant math can be found in the paper "Projective Mappings for Image Warping" by Paul Heckbert.

http://pages.cs.wisc.edu/~dyer/cs766/readings/heckbert-proj.pdf

This way you can recover the homogenous part of each vertex back that was lost during projection.

Now you're still left with four lines instead of points (as Ron explained). Since you know the size of your original rectangle however nothing is lost. You can now plug the data from Ron's method and from the 2D approach into a linear equation solver and solve for z. You get the exact z-values of each vertex that way.

Note: This just works because:

1. The original shape was a rectangle
2. You know the exact size of the rectangle in 3D space.

It's a special case really.

Hope it helps, Nils

Share Follow

answered Sep 18, 2008 at 15:50

Nils Pipenbrinck

86.2k • 33 • 155 • 223

2

Thanks to @Vegard for an excellent answer. I cleaned up the code a little bit:

```
import pandas as pd
import numpy as np

class Point2:
    def __init__(self,x,y):
        self.x = x
        self.y = y

class Point3:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

# Known 2D coordinates of our rectangle
i0 = Point2(318, 247)
i1 = Point2(326, 312)
i2 = Point2(418, 241)
i3 = Point2(452, 303)
```

```
# 3D coordinates corresponding to i0, i1, i2, i3
r0 = Point3(0, 0, 0)
r1 = Point3(0, 0, 1)
r2 = Point3(1, 0, 0)
r3 = Point3(1, 0, 1)

mat = [
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
]

def project(p, mat):
    #print mat
    x = mat[0][0] * p.x + mat[0][1] * p.y + mat[0]
[2] * p.z + mat[0][3] * 1
```

The approximate call with .1 did not work for me, so I took it out. I ran it for a while too, and last I checked it was at

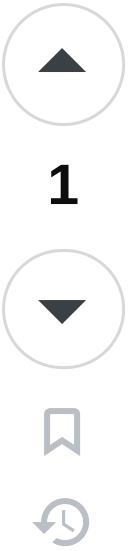
```
[[0.7576315397559887, 0, 0.11439449272592839,
-0.314856490473439],
[0.06440497208710227, 1, -0.5607502645413118,
0.38338196981556827],
[0, 0, 1, 0],
[0.05421620936883742, 0, -0.5673977598434641,
2.693116299312736]]
```

with an error around 0.02.

Share Follow

answered Oct 5, 2016 at 14:53

BBSysDyn
4,561 ● 10 ● 51 ● 65



The projection you have onto the 2D surface has infinitely many 3D rectangles that will project to the same 2D shape.

Think about it this way: you have four 3D points that make up the 3D rectangle. Call them (x0,y0,z0), (x1,y1,z1), (x2,y2,z2) and (x3,y3,z3). When you project these points onto the x-y plane, you drop the z coordinates: (x0,y0), (x1,y1), (x2,y2), (x3,y3).

Now, you want to project back into 3D space, you need to reverse-engineer what z0,..,z3 were. But any set of z coordinates that a) keep the same x-y distance between the points, and b) keep the shape a rectangle will work. So, any member of this (infinite) set will do: {(z0+i, z1+i, z2+i, z3+i) | i <- R}.

Edit @Jarrett: Imagine you solved this and ended up with a rectangle in 3D space. Now, imagine sliding that rectangle up and down the z-axis. Those infinite amount of translated rectangles all have the same x-y projection. How do you know you found the "right" one?

Edit #2: Alright, this is from a comment I made on this question -- a more intuitive approach to reasoning about this.

Imagine holding a piece of paper above your desk. Pretend each corner of the paper has a weightless laser pointer attached to it that points down toward the desk.

The paper is the 3D object, and the laser pointer dots on the desk are the 2D projection.

Now, how can you tell how high off the desk the paper is by looking at *just* the laser pointer dots?

You can't. Move the paper straight up and down. The laser pointers will still shine on the same spots on the desk regardless of the height of the paper.

Finding the z-coordinates in the reverse-projection is like trying to find the height of the paper based on the laser pointer dots on the desk alone.

Share Follow

edited Sep 16, 2008 at 20:28

answered Sep 16, 2008 at 20:00

Rob Dickerson

1,469 ● 9 ● 6

1 Yes, but I stated in the question that I know the size of the rectangle as well. There should only be one set of Z-coordinates wherein all 4 points are the proper distance apart. Sliding it up and down the z-axis would result in larger or smaller polygons. – Joshua Carmody Sep 16, 2008 at 20:11

No, the polygons would be the same size. Hold a piece of paper above your desk. Imagine where the corners of the paper trace down to the desk surface. Now, move the paper up and down. The points hit the same spot on the desk in all of those locations, but the paper remains the same size. – Rob Dickerson Sep 16, 2008 at 20:22

Rob, you totally ignore the possibility of perspective. For some reason you think only of orthogonal projection. – tzot Sep 19, 2008 at 11:11

Indeed. The ortogonal case is only 1 case in the infinite set of projections. On top of that it's the leasst probable one (the viewpoint must lie infinitely far fom the plane you project onto). -1. – xtofl Nov 14, 2008 at 15:00

homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARBLE/high/… – freakTheMighty Apr 15, 2011 at 6:42

1

When you project from 3D to 2D you lose information.

In the simple case of a single point the inverse projection would give you an infinite ray through 3d space.

Stereoscopic reconstruction will typically start with two 2d images and project both back to 3D. Then look for an intersection of the two 3D rays produced.

Projection can take different forms. Orthogonal or perspective. I'm guessing that you are assuming orthogonal projection?

In your case assuming you had the original matrix you would have 4 rays in 3D space. You would then be able to constrain the problem by your 3d rectangle dimensions and attempt to solve.

The solution will not be unique as a rotation around either axis that is parallel to the 2d projection plane will be ambiguous in direction. In other words if the 2d image is

perpendicular to the z axis then rotating the 3d rectangle clockwise or anti clockwise around the x axis would produce the same image. Likewise for the y axis.

In the case where the rectangle plane is parallel to the z axis you have even more solutions.

As you don't have the original projection matrix further ambiguity is introduced by an arbitary scaling factor that exists in any projection. You cannot distinguish between a scaling in the projection and a translation in 3d in the direction of the z axis. This is not a problem if you are only interested in the relative positions of the 4 points in 3d space when related to each other and not to the plane of the 2d projection.

In a perspective projection things get harder...

Share Follow

answered Sep 16, 2008 at 20:58

morechilli

9,817 ● 7 ● 37 ● 55

1

If you know the shape is a rectangle in a plane, you can greatly further constrain the problem. You certainly cannot figure out "which" plane, so you can choose that it is lying on the plane where z=0 and one of the corners is at x=y=0, and the edges are parallel to the x/y axis.

The points in 3d are therefore {0,0,0},{w,0,0},{w,h,0},and {0,h,0}. I'm pretty certain the absolute size will not be

found, so only the ratio w/h is releavant, so this is one unknown.

Relative to this plane the camera must be at some point cx,cy,cz in space, must be pointing in a direction nx,ny,nz (a vector of length one so one of these is redundant), and have a focal_length/image_width factor of w. These numbers turn into a 3x3 projection matrix.

That gives a total of 7 unknowns: w/h, cx, cy, cz, nx, ny, and w.

You have a total of 8 knowns: the 4 x+y pairs.

So this can be solved.

Next step is to use Matlab or Mathmatica.

Share Follow

answered Sep 16, 2008 at 21:13

spitzak

1

I'll get my linear Algebra book out when I get home if nobody answered. But @ D G, not all matrices are invertible. Singular matrices aren't invertible (when determinant = 0). This will actually happen all the time, since a projection matrix *must* have eigenvalues of 0 and 1, and be square (since it is idempotent, so p^2 = p).

An easy example is, [[0 1][0 1]] since the determinant = 0, and that is a projection on the line x = y!

Share Follow

edited Mar 9, 2009 at 18:52

answered Sep 16, 2008 at 19:59

nlucaroni

47.9k ● 6 ● 66 ● 86

0

Yes, Monte Carlo works, but I found better solution for this issue. This code works perfectly (and uses OpenCV):

```
Cv2.CalibrateCamera(new List<List<Point3f>>() { points3d }, new List<List<Point2f>>() { points2d }, new Size(height, width), cameraMatrix, distCoefs, out rvecs, out tvecs, CalibrationFlags.ZeroTangentDist | CalibrationFlags.FixK1 | CalibrationFlags.FixK2 | CalibrationFlags.FixK3);
```

This function takes known 3d and 2d points, size of screen and returns rotation (rvecs[0]), translation (tvecs[0]) and matrix of intrinsics values of camera. It's everything you need.

Share Follow

edited May 7, 2017 at 13:33

answered May 5, 2017 at 9:48

Inflight

526 ● 4 ● 13

OpenCV also has FindHomography() method, which works perfectly too. – Inflight May 21, 2017 at 16:31