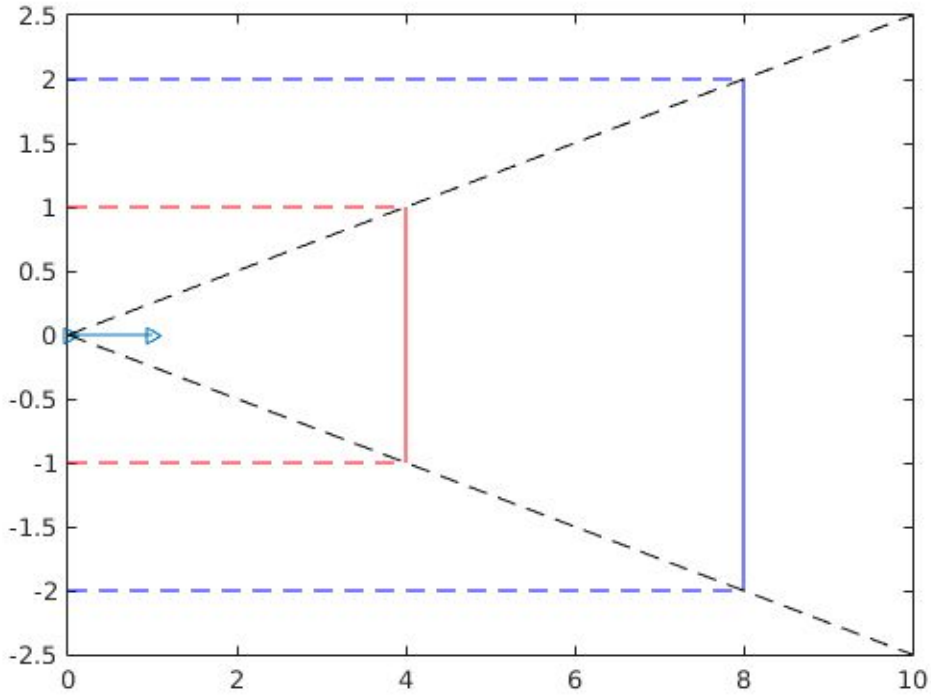
Introduction

We seek to develop the mathematical notions of projective geometry and homogeneous coordinates, and show how these notions allow perspective projections in computer graphics. Interestingly, we find that perspective projection can be accomplished with one matrix multiplication! We introduce homogeneous coordinates and the addition of another dimension to allow us to perform transformations and projections as they relate to computer graphics.

Intro to Homogeneous Coordinates - 2D Objects

To understand the notions of perspective space, it is helpful to first see what we wish to accomplish in two dimensions before generalizing. To this end, consider what images a camera embedded in a two-dimensional scene would capture:



In this scene, we have red and blue line segments which we would like an image of, with our camera facing along the x axis, shown by the unit vector in blue.

With orthogonal projection, we see a line that is blue on [-2,-1], red on (-1,1), and blue on [1,2] - this is shown with the red and blue dashed lines. With perspective, we see the red segment completely obstructing the blue segment. But how can we represent this image mathematically?

One way to represent a perspective view of the world is to rescale parts of the image based on how far away from the observer they are before performing the orthogonal projection. To do this, we will expand our coordinate system to include a scale factor, often called W. When we want to find the scaled coordinates, we just divide all coordinates by the W coordinate.

The points in our 2d world are concatenated into this matrix:

```
cat_pts =
    8     8     4     4
    2    -2     1    -1
```

Turning these points into homogeneous coordinates is as simple as adding a value of 1 for the W coordinate, since no rescaling has occurred:

```
hom_pts =
    8     8     4     4
    2    -2     1    -1
    1     1     1     1
```

Now, we need a transformation that assigns the W coordinates. In perspective, we want points that are directly in front of the camera to remain in front of the camera, and points that are oblique to the camera angle to become increasingly far away from the center, out to infinity (the camera cannot see points orthogonal to its aim). The dot product of the camera vector and the point is what we are after. Let's write it as a matrix:

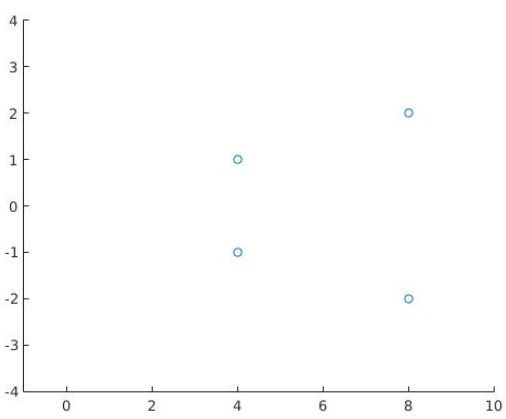
```
camera =
    1     0     0
    0     1     0
    1     0     0
```

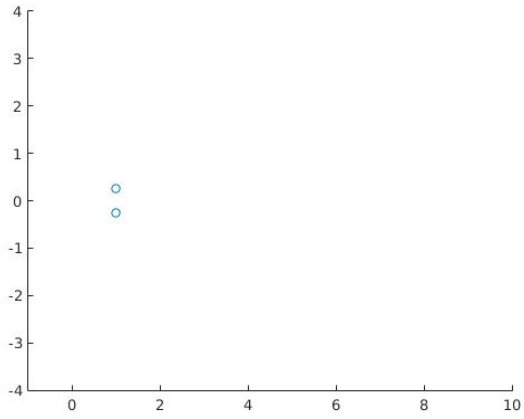
```
per_pts = camera*hom_pts
```

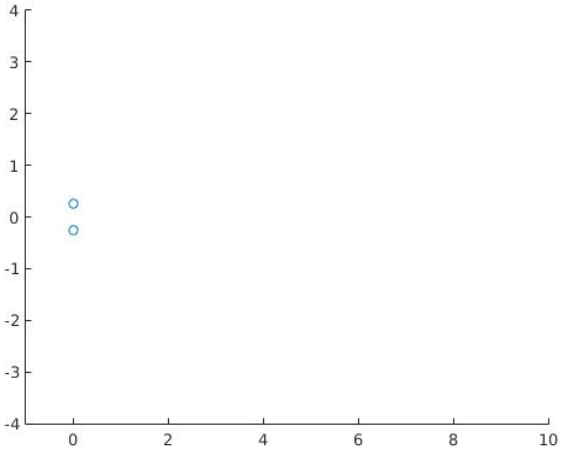
```
per_pts =
    8     8     4     4
    2    -2     1    -1
    8     8     4     4
```

After rescaling these coordinates so that W=1, something interesting has happened:

```
per_pts =
   1.0000    1.0000    1.0000    1.0000
   0.2500   -0.2500    0.2500   -0.2500
   1.0000    1.0000    1.0000    1.0000
```







The last two points are the same as the first two - we have accomplished the obstruction of one line segment by another! Now all that is left is to perform an orthogonal projection to the line perpendicular to the camera:

```
proj_mat =
     0     0     0
     0     1     0
     0     0     1

proj_mat*per_pts

ans =
         0         0         0         0
    0.2500   -0.2500    0.2500   -0.2500
    1.0000    1.0000    1.0000    1.0000
```

And we have the coordinates for our image in one dimension. Note that we chose to represent the dot product operation as a matrix. This is because we can combine our transformations:

camera =

```
    0     0     0
    0     1     0
    1     0     0
```

Now performing per_pts = camera*hom_pts gives us:

per_pts =

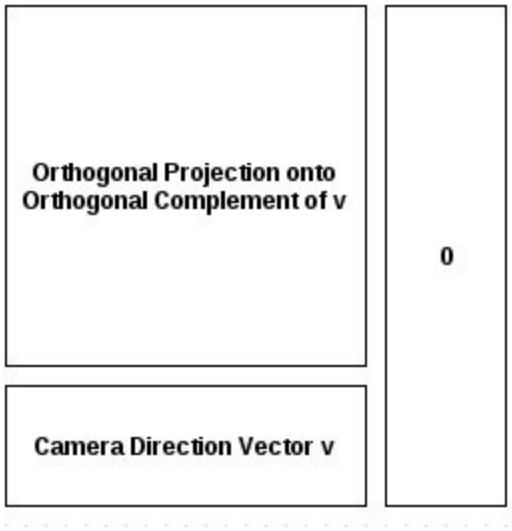
```
    0     0     0     0
    2    -2     1    -1
    8     8     4     4
```

After rescaling, we have:

per_pts =

```
         0         0         0         0
    0.2500   -0.2500    0.2500   -0.2500
    1.0000    1.0000    1.0000    1.0000
```

Which is exactly what we got before. Generally, we can construct the camera matrix like this:



This matrix construction generalizes into higher dimensions, so we can use this tactic to develop perspective projections from 3d to 2d. Let's try it!

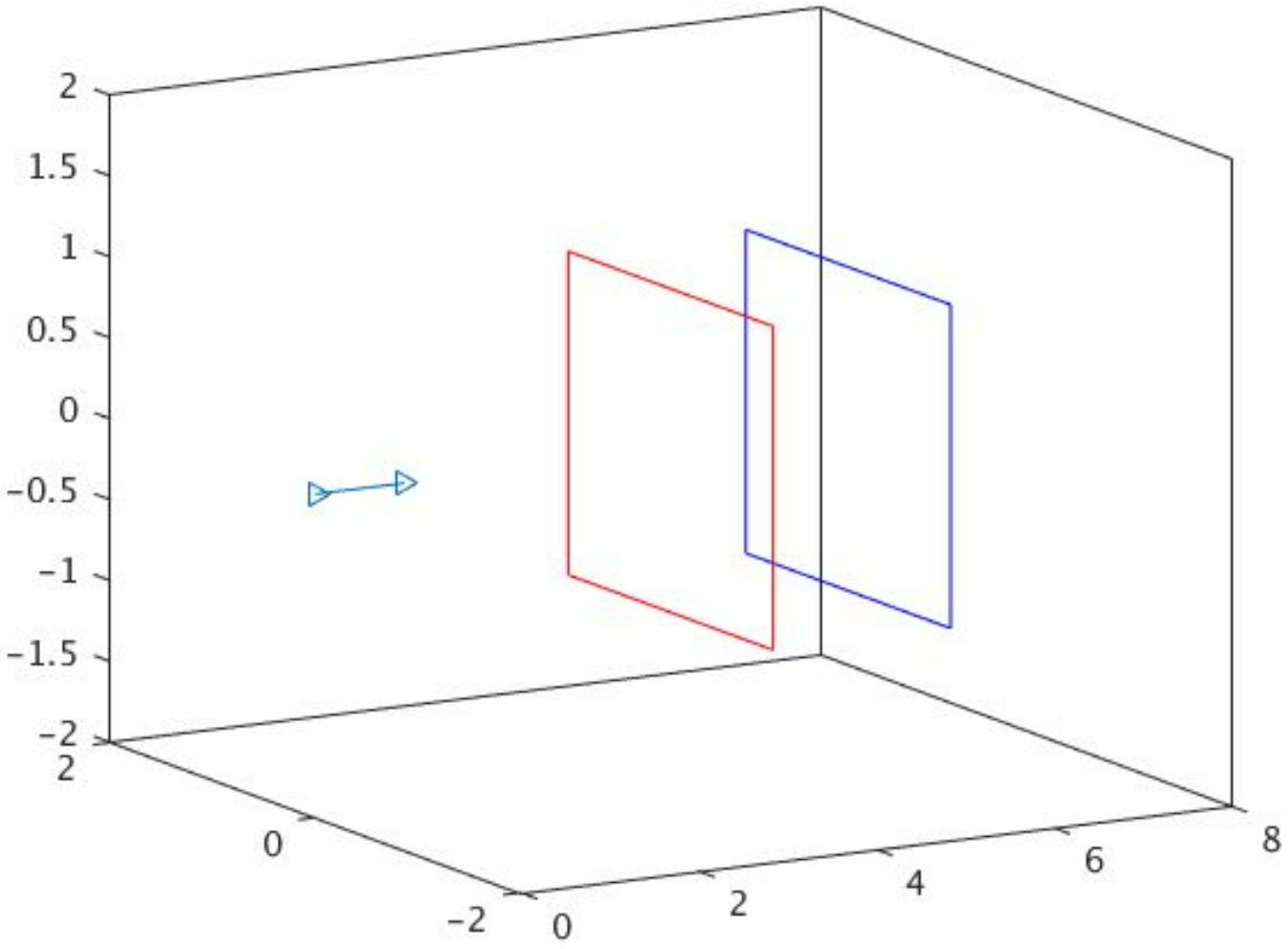
Homogeneous Coordinates - 3D Objects

red_square =

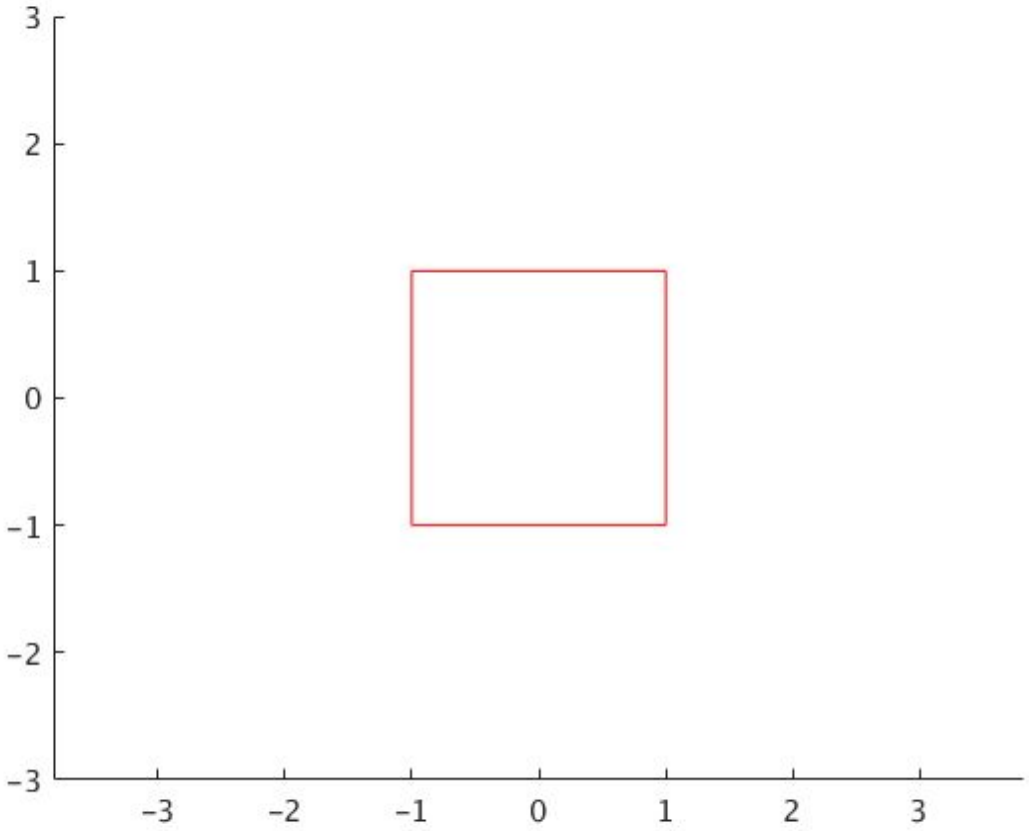
4	4	4	4
1	1	-1	-1
1	-1	-1	1

blue_square =

6	6	6	6
1	1	-1	-1
1	-1	-1	1



Note that the orthogonal projection of this figure shows the red square perfectly overlapping the blue square:



This is boring, so let's do the procedure we developed for perspective projections. Construct the camera matrix:

camera =

0	0	0	0
0	1	0	0
0	0	1	0
1	0	0	0

Add the W dimension to put the shape into homogeneous coordinates:

hom_pts =

4	4	4	4	6	6	6	6
1	1	-1	-1	1	1	-1	-1
1	-1	-1	1	1	-1	-1	1
1	1	1	1	1	1	1	1

Apply the camera matrix: per_pts = camera*hom_pts

per_pts =

0	0	0	0	0	0	0	0
1	1	-1	-1	1	1	-1	-1
1	-1	-1	1	1	-1	-1	1
4	4	4	4	6	6	6	6

Rescale the coordinates to make W=1 and then move back to standard coordinates:

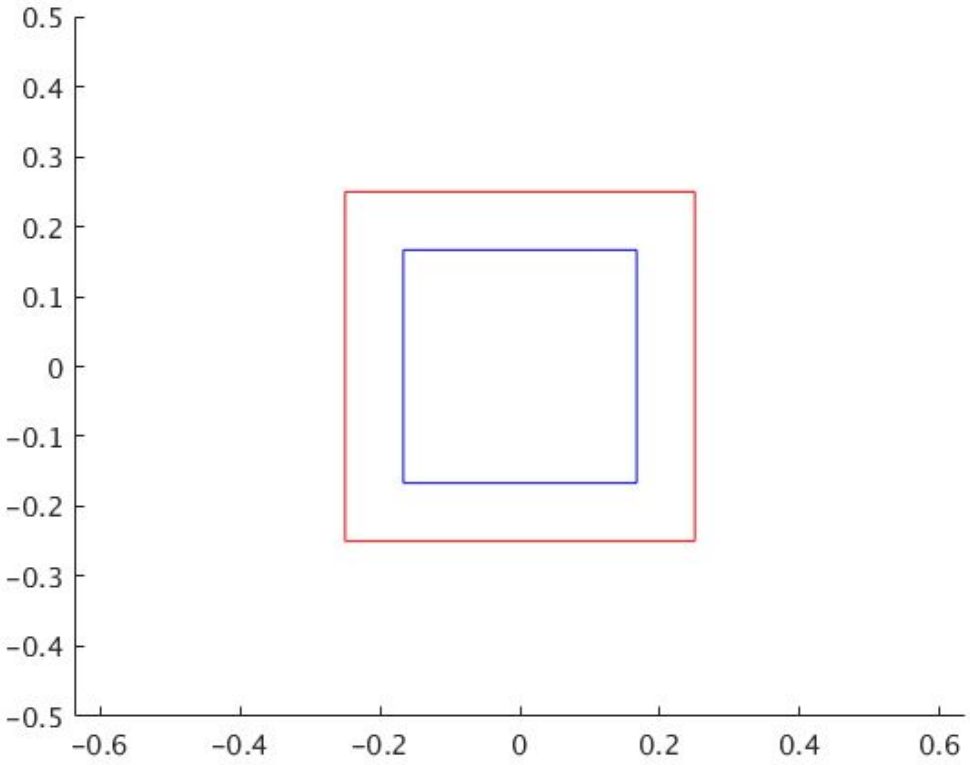
per_pts =

0	0	0	0	0	0	0	0
0.2500	0.2500	-0.2500	-0.2500	0.1667	0.1667	-0.1667	-0.1667
0.2500	-0.2500	-0.2500	0.2500	0.1667	-0.1667	-0.1667	0.1667
1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

final_pts =

0	0	0	0	0	0	0	0
0.2500	0.2500	-0.2500	-0.2500	0.1667	0.1667	-0.1667	-0.1667
0.2500	-0.2500	-0.2500	0.2500	0.1667	-0.1667	-0.1667	0.1667

We get exactly the perspective image we had hoped for:



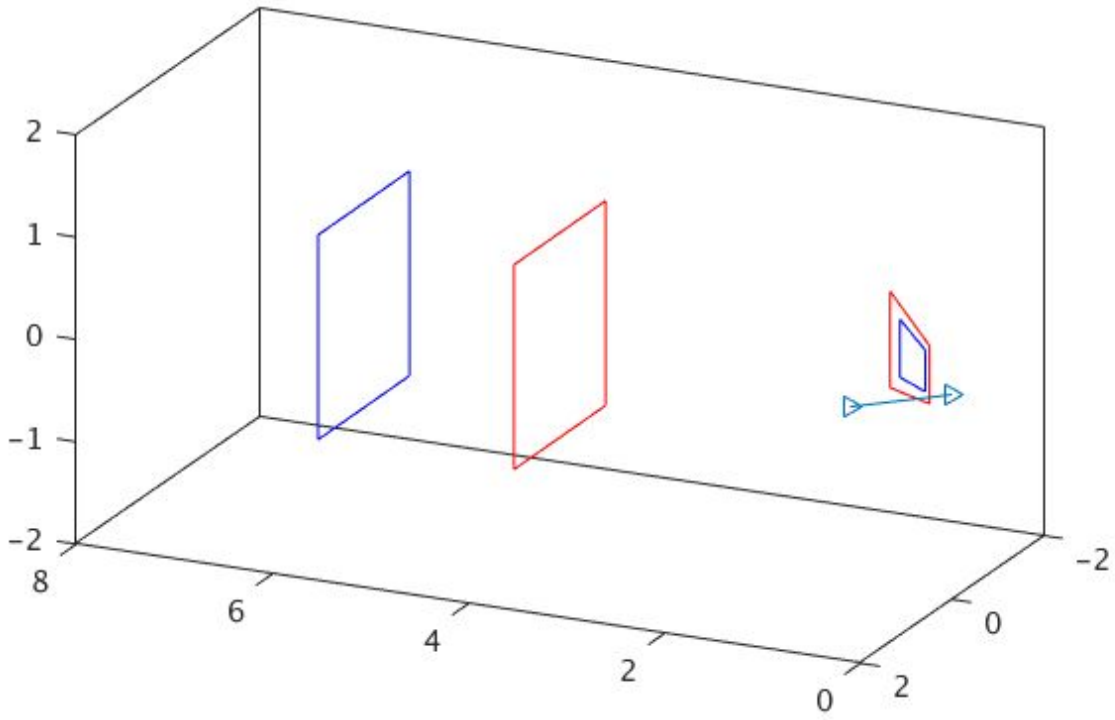
Creating the camera matrix

If we have a vector that describes the direction the camera is facing, we need to find the matrix of the orthogonal projection onto the orthogonal complement of that vector. We do this by finding an orthonormal basis for R3 where the first two vectors are in the easel plane, and then creating the matrix that moves coordinates to that basis, drops the component in the camera direction to zero, and then moves back into standard basis. This MATLAB code does that:

```
easel = null(transpose(camera_vector)) % ONB for the easel plane
to_easel_basis = [easel camera_vector]
camera_proj = to_easel_basis*[1 0 0; 0 1 0; 0 0 0]*inv(to_easel_basis);
camera = [vertcat(camera_proj,transpose(camera_vector)) zeros(4,1)]
```

And now we can rotate the camera and draw the skewed image the camera sees on the plane orthogonal to the camera vector.

```
camera_vector = [1; 1; 0];
camera_vector = camera_vector/norm(camera_vector)
```



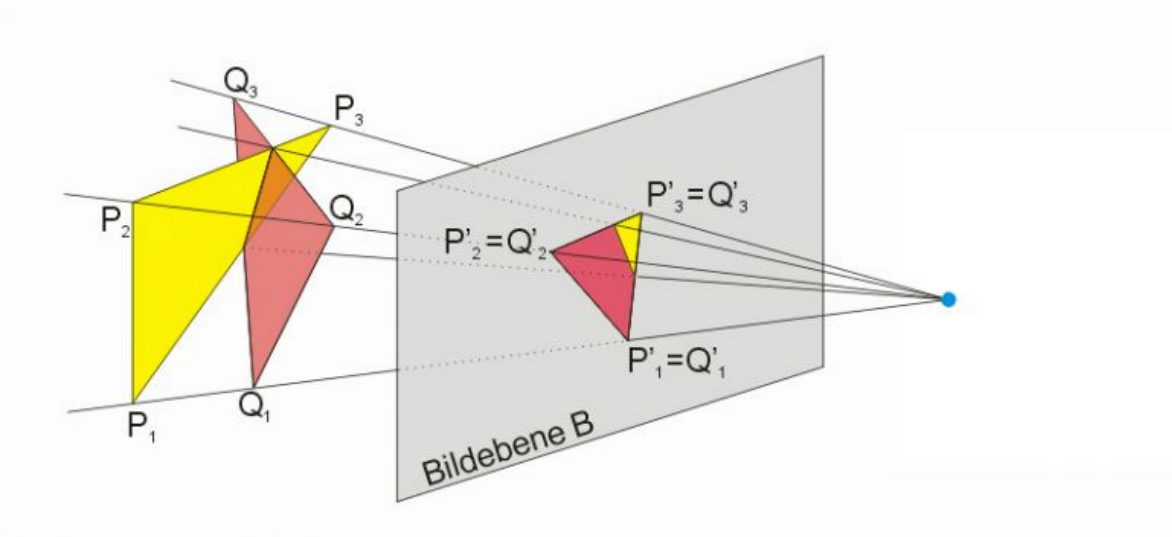


Image Credit: Tom Dalling

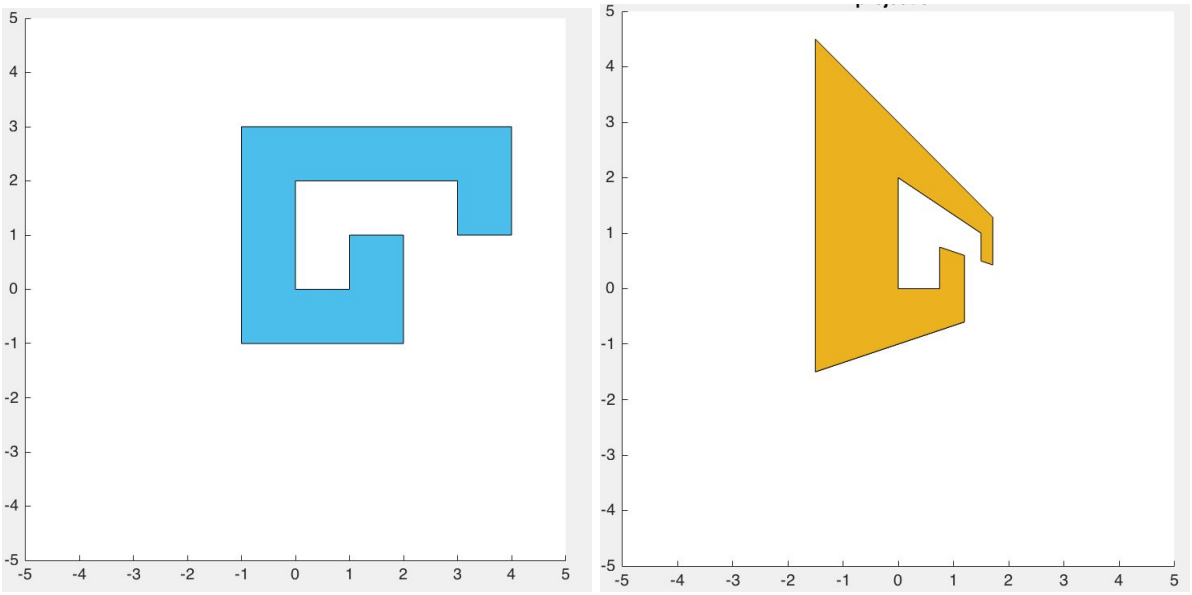


Image Credit: Mike Garrity

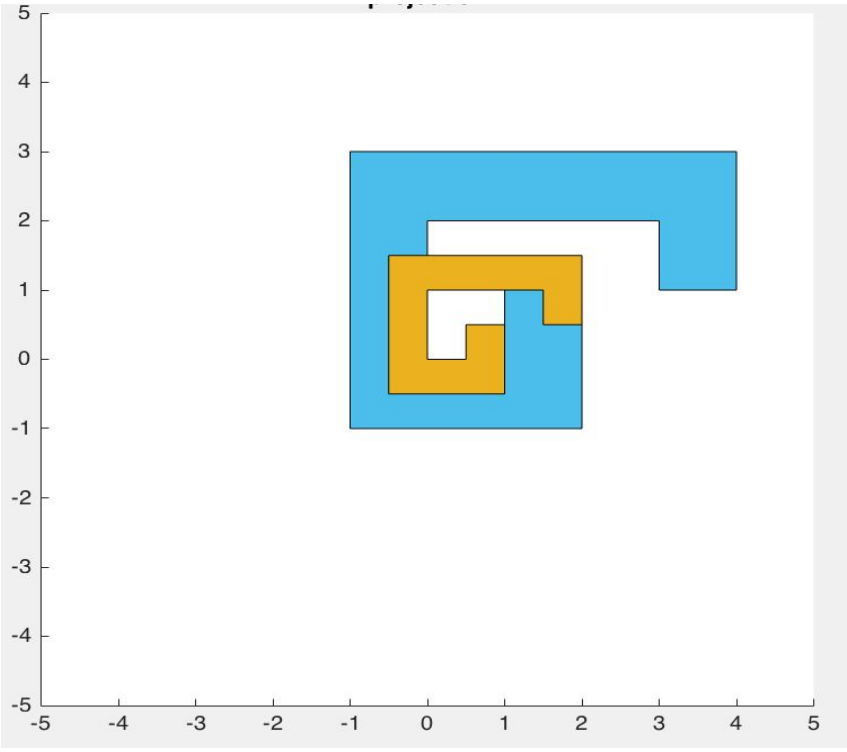
Extending to 3D Objects

It is difficult to imagine the scaling factor W in 3-dimensions, as there is no spatial representation for it (we have to add a fourth dimension). Instead we must acknowledge how increasing the scaling factor will cause each coordinate to shrink and vice-versa with decreasing or increasing W. After performing transformations on the space, we can change W accordingly then rescale it to 1 to place it in the viewing plane.

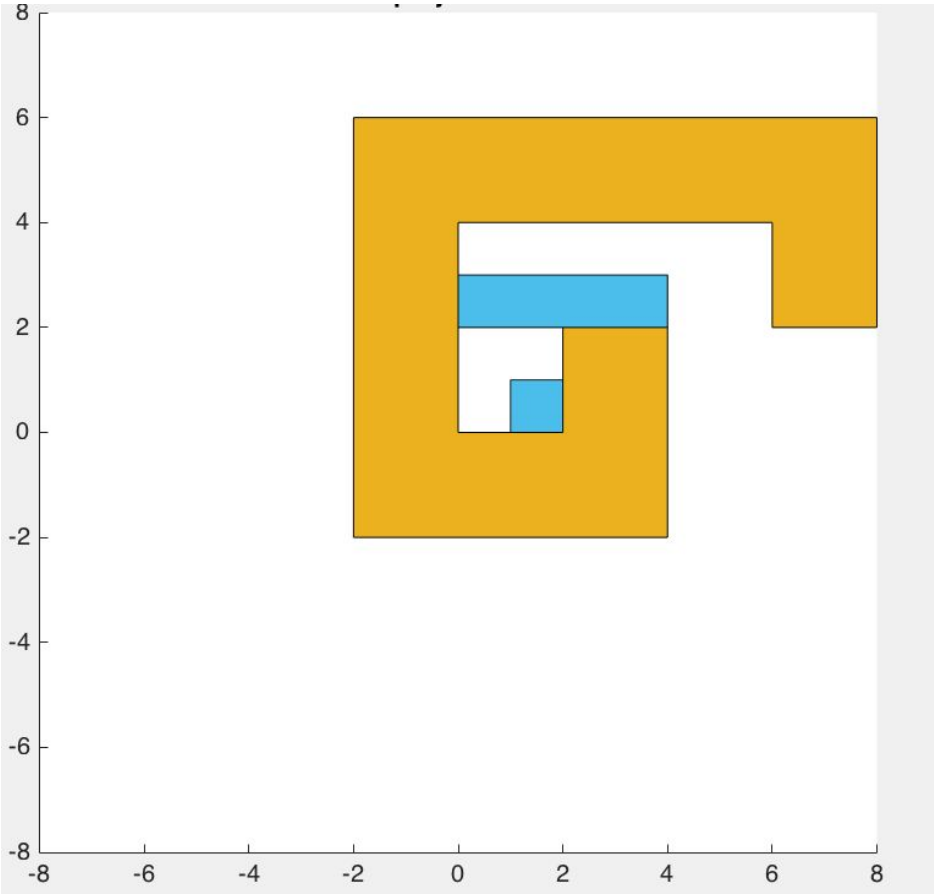
Homogeneous Coordinates - Scaling, Rotating, and Beyond

Here, we leverage the use of homogeneous coordinates to show how moving away or toward the projection screen (our 2-dimensional world) will change our perception of the projected object. The added dimension ‘W’ to our x-y plane will allow us to represent the distance between the viewer and the projection screen (ie. the distance between the camera and the 2-dimensional world). For example, if we increase our distance from the screen, we expect all points of an object to shrink to a common focal point, while maintaining the integrity of the object. Conversely, if we decrease our distance to the projection screen, we will notice all points on the object to diverge, namely, approach infinity. If we stand in the same plane as the projection screen, we develop the notion of points being projected to infinity. Since we eliminate the third (added) dimension by setting W = 0, all points are being scaled by dividing by 0, thus sending every point of the object to infinity. Now we will explore how homogeneous coordinates allow us to perform different transformations in the 2-dimensional realm, something easier to visualize and grasp.

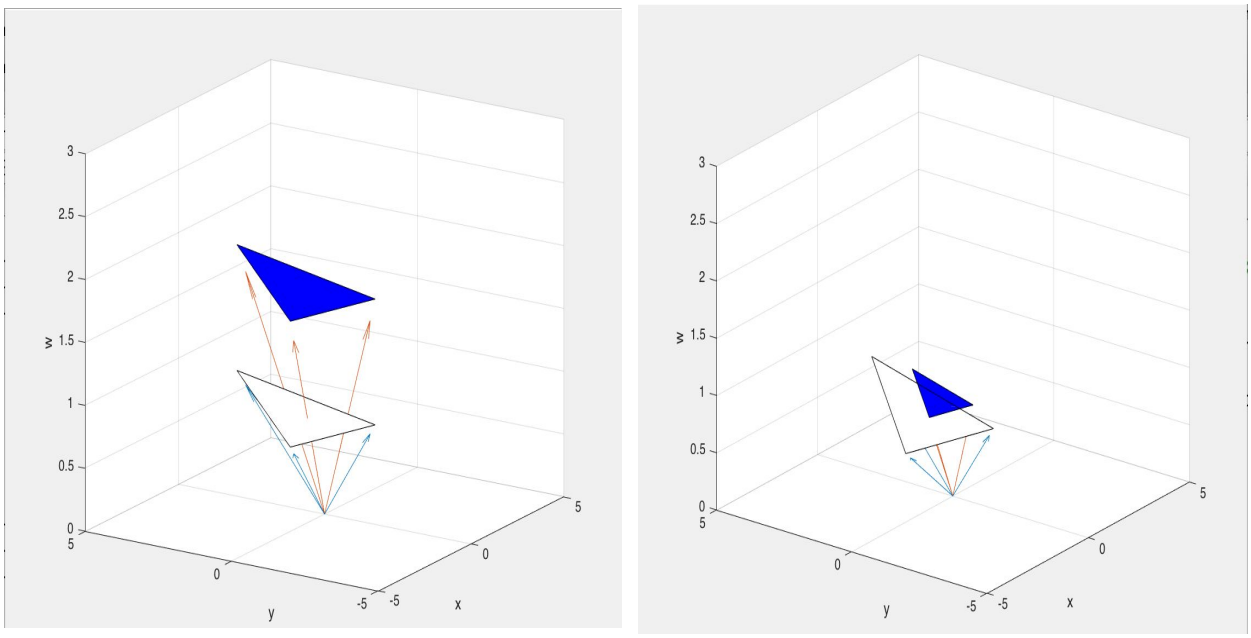
Here’s an example of a simple scaling using homogeneous coordinates. We realize that if we wanted to scale this 2-dimensional object without homogeneous coordinates, we would have to apply a different transformation matrix for every point. The addition of another coordinate (ie. adding another dimension) will allow us to create just one transformation matrix. Scaling W up is analogous to traveling farther away from the object in question, thus we see the object shrink while keeping the integrity of the shape.



Likewise, here is a case where we scale W down (our extra coordinate/dimension). This would be analogous to the camera getting closer to the object in question, thus we see the object enlarge while maintaining the shape of the object.

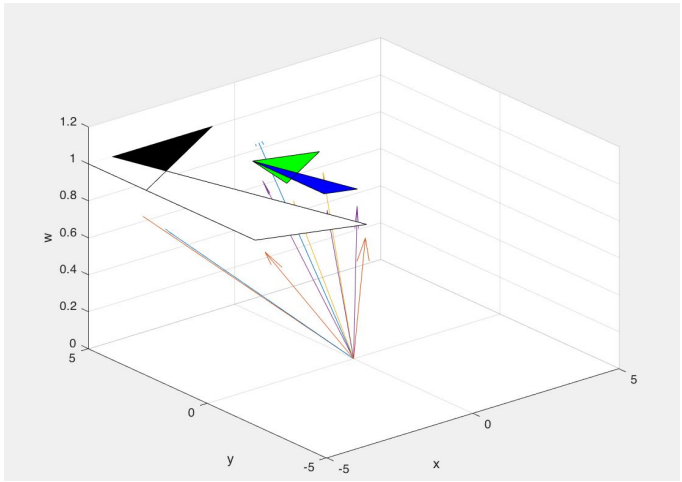


Another Geometric Interpretation of the W coordinate



Scaling ‘W’ Coordinate Based Upon “Distance from R^2 plane”

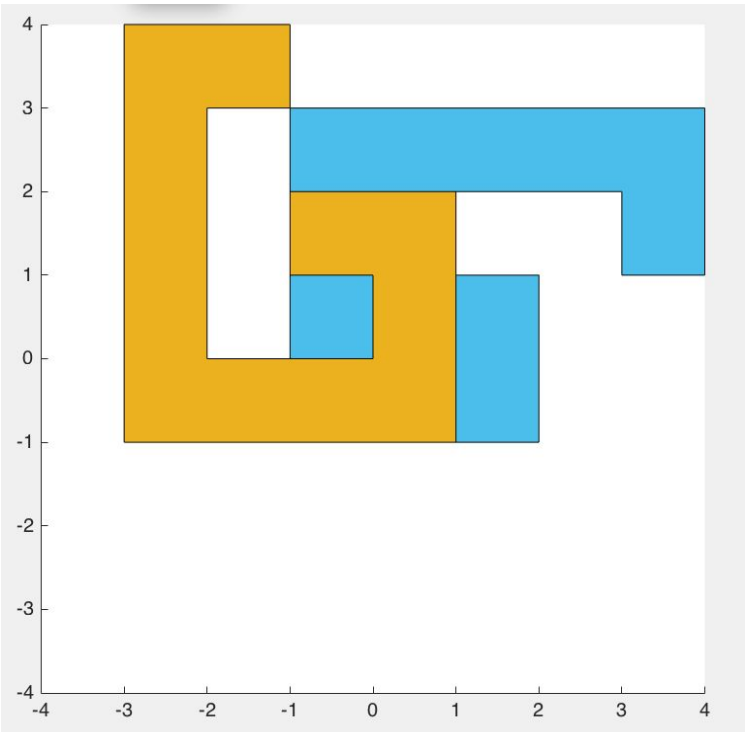
The figure above shows how the use of the w coordinate allows us to scale from an origin point while keeping the integrity of the shape. The figure on the left shows changing w from 1 (white triangle) to 2 (blue triangle). The object must always exist in the same world R^2 thus we rescale all points for w to be 1. When we rescale all the points to allow w to be 1, the blue triangle is now in the same plane as its original self (white triangle). Its image now becomes where the vectors cross w = 1 on the left image.



Bringing Objects into view

This allows allows us to bring objects into view that originally existed at “infinity”. This comes from the idea that if an object isn’t in view, it still exists in R^2 but exists at infinity for our view. It comes into our plane when we “stand far enough back” allows us to be more of the world around us. When we decrease w, the opposite happens, sending objects out of view to infinity. The blue triangle has its corresponding original white triangle while the green has its corresponding original black triangle.

We can also leverage the use of homogeneous coordinates to do other types of transformations, namely, rotations. We can construct our rotation matrix just like we would for a simple 2-dimensional case, but with the addition of a third dimension that represents the W-dimension. This will allow us to complete the transformation in one simple matrix, without the necessity to transform every point individually.




```
pts = [4  4 -1 -1  2  2  1  1  0  0  3  3;
       1  3  3 -1 -1  1  1  0  0  2  2  1;
       2  2  2  2  2  2  2  2  2  2  2  2;
       1  1  1  1  1  1  1  1  1  1  1  1];
axis equal
xlim([-4 4])
ylim([-4 4])
zlim([-4 4])
h1 = patch('FaceColor',[0.3010 0.7450 0.9330]);
h1.XData = pts(1,:) ./ pts(4,:);
h1.YData = pts(2,:) ./ pts(4,:);
h1.ZData = pts(3,:) ./ pts(4,:);

mrot = [cos(pi/2) -sin(pi/2) 0 0;
        sin(pi/2) cos(pi/2) 0 0'
        0 0 1 0;
        0 0 0 1];

p2 = mrot*pts;

h2 = patch('FaceColor',[0.9290 0.6940 0.1250]);
h2.XData = p2(1,:) ./ p2(4,:);
h2.YData = p2(2,:) ./ p2(4,:);
h2.ZData = p2(3,:) ./ p2(4,:);
```

References

Bretscher, Otto. *Linear Algebra with Applications*. 5th ed. Upper Saddle River, New Jersey: Pearson Education, 2013. Print.

Clark, Gabrien. "Computer Graphics." *Computer Graphics and Linear Algebra*. University of North Texas, 5 May 2010. Web. 4 Dec. 2015.

Dalling, Tom. "Explaining Homogeneous Coordinates and Projective Geometry." Disqus, 24 Feb. 2014. Web. 24 Nov. 2015.

Garrity, Mike. "Homogeneous Coordinates." *MATLAB Central*. The Mathworks, Inc., 28 Sept. 2015. Web. 2 Dec. 2015.

Hoggar, S.G. *Mathematics for Computer Graphics*. Vol. 14. Cambridge UP, 1993. Print.

Vince, John. *Mathematics for Computer Graphics*. 3rd ed. Springer, 2010. Print.

Yip, Tang. "Matrices in Computer Graphics." University of Washington, 3 Dec. 2001. Web. 17 Nov. 2015.