Part V – Mouse to Plane coordinates, second step to implementing panorama hotspots

In the previous post, we implemented a vector pointing at the cube plane under your mouse. In this post we will look at deriving the local x,y coordinate within that plane, and with it the local x,y coordinate of the pixel within the plane's texture under your mouse. (Although for hotspots, any local coordinate system will do).

In order to do so, we need to project the vector at the plane it is pointing at. This results in a coordinate that lies within the plane. The coordinate comes from a set of coordinates that all lie within that plane, and this set can be mapped to a range representing the texture coordinates (or hotspot coordinates as we will see later). Although this might sound complicated, it is not so bad as it sounds.

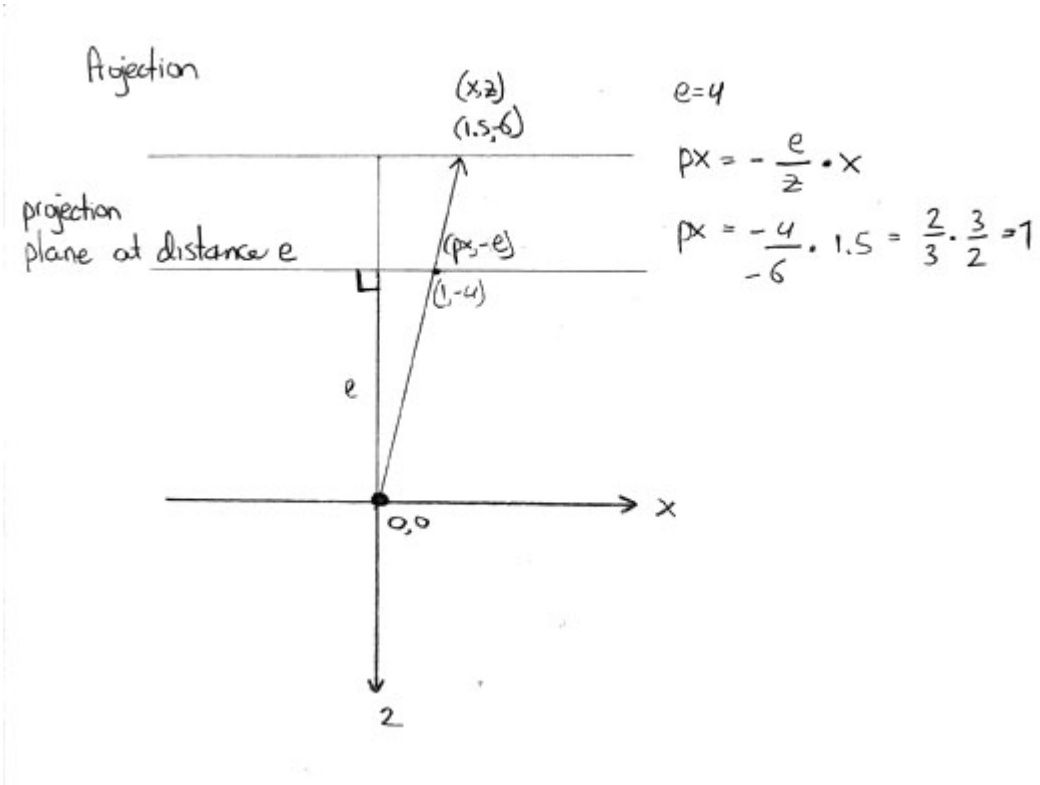
Let's review projection first.

The formula for projection was:

px = – (projectionplanedistance / original_z) * original_x
py = – (projectionplanedistance / original_z) * original_y

It doesn't matter whether the original z lies in front or behind the projection plane.

My next two crappy sketches demonstrate this point, first behind the plane (I have left the y coordinate out of the images for clarity, so it is as if we are viewing the 3d scene from the top):

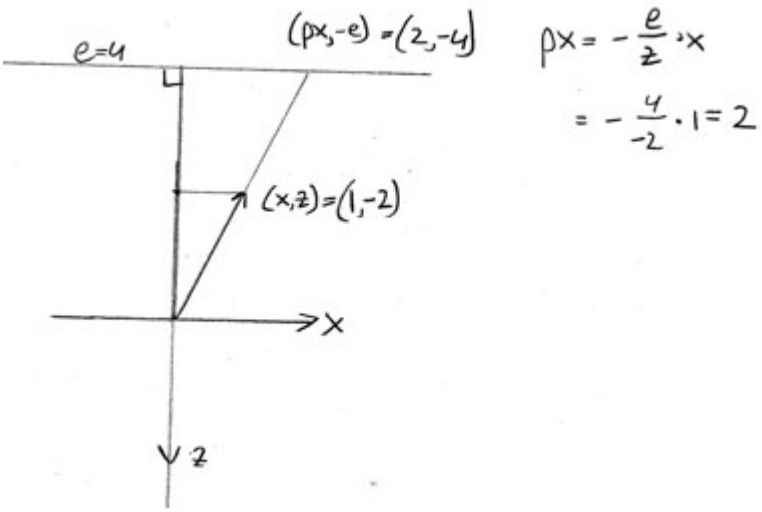


Our original point lies at an (x,z) of (1.5,-6) (in other words x,y,z = 1.5,0,-6).
Our projectionplane lies at a distance of 4 from the origin:

px = – (4 / -6) * 1.5) =>
px = 2/3 * 1.5
px = 1

Our projected point has an x of 1, at a z of -e.

A point in front of the projectionplane is handled the same way:



$$px = -\frac{e}{z} \cdot x$$

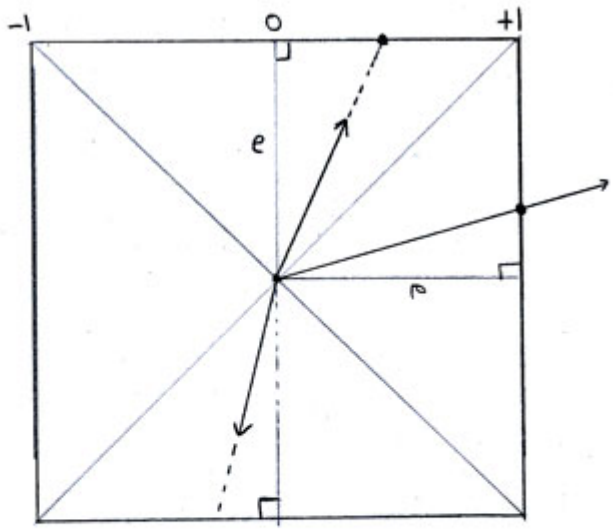
$$= -\frac{4}{-2} \cdot 1 = 2$$

Our original point lies at an (x,z) of (1,-2) (in other words x,y,z = 1,0,-2).
Our projectionplane lies at a distance of 4 from the origin.

px = – (4 / -2) * 1) =>
px = 2

Our projected point has an x of 1, at a z of -e.

In a cube we can do the same, by projecting each vector at the plane it is pointing at no matter what the length of the original vector was:



$$px = -\frac{e}{z} \cdot x$$

$$e = 1 \Rightarrow px = -\frac{x}{z} \quad \text{where} \quad -1 \leq px \leq 1$$

The distance e we choose is arbitrary, it merely influences the range of the resulting projected values which have to be normalized/mapped to the texture coordinates anyway. However if we choose 1, it simplifies our projection formulas:

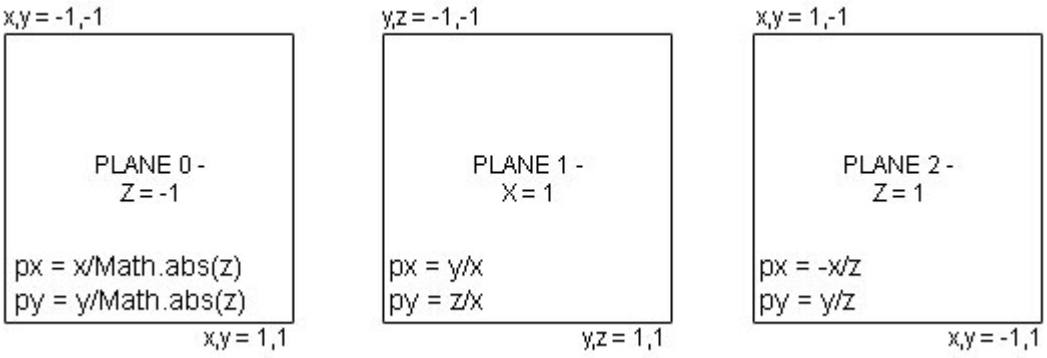
```
px = – (1 / original_z) * original_x
py = – (1 / original_z) * original_y
```

which is

```
px = -(original_x / original_z)
py = -(original_y / original_z)
```

Which will result in a mapping of the vector to a projected px and py within the range -1 to 1.

BUT which coordinate values we should use as x, y and z’s for our projection formulas depends on the orientation of the plane we are projecting our vector at, take a look at the following image:



The projection formulas are based on the projection of coordinates having a negative z with respect to the plane we are projection points on. So for plane 0 we get what you would expect:

```
px = -(original_x / original_z)
py = -(original_y / original_z)
```

And in plane 0, z is negative, so this is the same as:

```
px = (original_x / Math.abs(original_z))
py = (original_y / Math.abs(original_z))
```

The only reason I'm writing it that way is that we already calculated the maximum absolute values of x,y,z to determine which plane we were pointing at. We see the original upperleft corner in plane 0 has an x,y value of -1,-1 to a lowerright x,y value of 1,1, so there is nothing extra we have to do to map these coordinates from topleft to lowerright to a range of -1 to 1.

For plane 1, we see that the negative z value we need for our projection formulas is actually a positive x value, and that the local x,y is represented by a pair of y,z coordinates. So we negate our x value and fill in the rest:

```
px = -(original_y / -original_x)
py = -(original_z / -original_x)
```

which is

```
px = (original_y / original_x)
py = (original_z / original_x)
```

Since the topleft y,z = -1,-1 to bottomright y,z 1,1 is already mapped correctly no further action is required.

(If you are confused about the planes and their corner coordinates, check out the previous post which displays the cube with all it's coordinates per plane.)

For plane 2, we see the negative required z value is actually a positive z value, and the local x,y pair is actually a local x,y pair, so we get:

```
px = -(original_x / -original_z)
py = -(original_y / -original_z)
```

which is

```
px = (original_x / original_z)
py = (original_y / original_z)
```

BUT WAIT! We see that the upperleft coordinate of the plane is not -1,-1. It's 1,-1. And the lowerright is not 1,1 but it is -1,1. In other words the x range has been flipped. To correct this we need to negate the px:

```
px = -(original_x / original_z)
py = (original_y / original_z)
```

We can do this for each plane, but I will spare you the pain, since I already did that in the source code.

Now we have the calculations that map our mouse position within a plane to a local x,y coordinate with x and y both in the range -1, 1. We can map this to a texture pixel coordinate. Assuming our projected local x and y are represented by planeX and planeY we get:

```
texture_x = (plane_x/2) * texture_size + (texture_size/2);
```

which is the same as:

texture_x = ((plane_x+1) * texture_size) /2;

and ofcourse for y:

texture_y = ((plane_y+1) * texture_size) /2;

The next example demonstrates these principles. Check out the updated checkPlane method. Note that the texture size is 400 x 400 pixels.

(Please give the panorama a moment to load)
Download the source here: [Download not found]

Finally in our next post we will implement the hotspot detection and then we have only one post to go I think which demonstrates hotspots in a 3d panorama that light up as you mouse over them and are correctly transformed in perspective along with the rest of the cube.