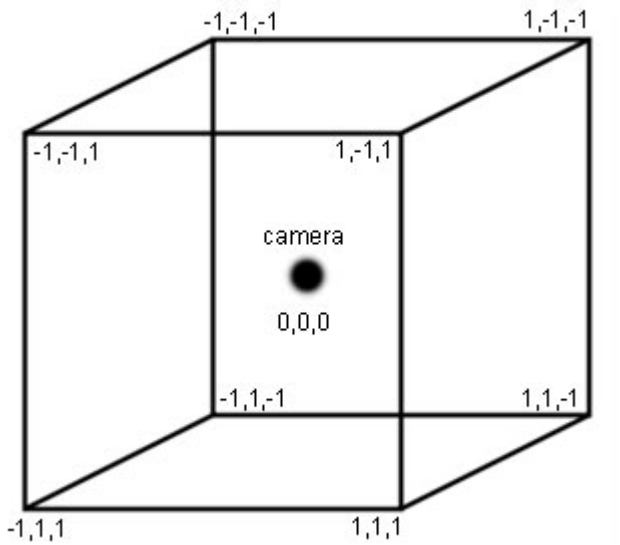
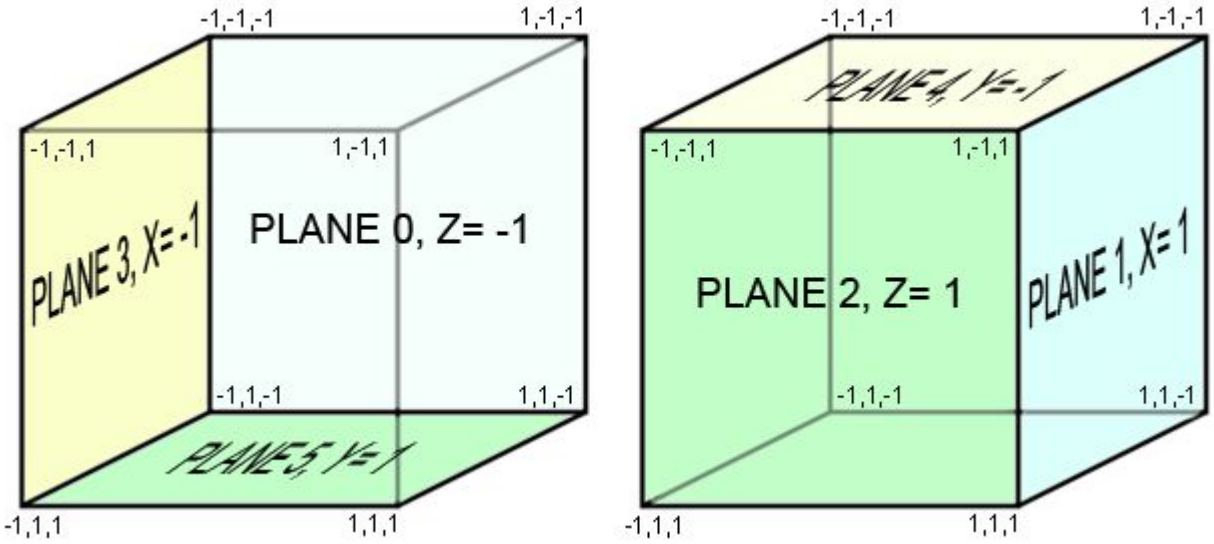
Part IV – Plane detection, a first step to implementing panorama hotspots

In order to be able to detect hotspots under the mouse, the first thing we need to do is find some way to detect which plane the mouse is currently over and what the local x,y coordinates of the mousecursor are in the local space of that plane.

Instead of providing all the theoretical background I want to try and make it conceptually clear how we can do this. So let go of the mouse coordinates for a moment and take a look at our untransformed starting cube we discussed in a previous post:



We can see that a cube is nothing more than a space enclosed by 6 planes. We can even specify the equations for these planes:

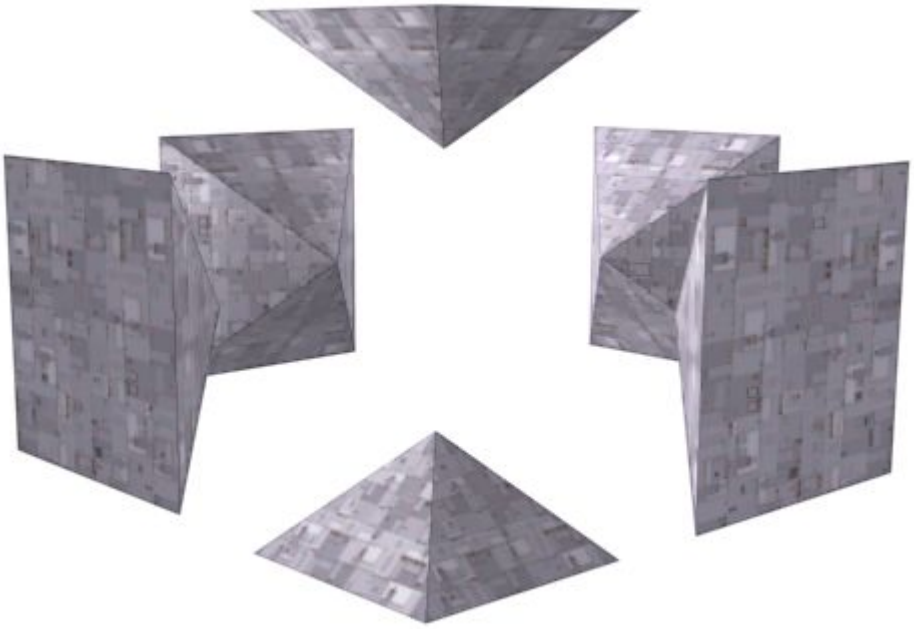


The plane numbers correspond to how we have defined our planes in the panoramas these past examples. Contained within this cube is the set of all (x,y,z) coordinates with an absolute value for x, y and z less or equal to 1. In addition for the coordinates that lie exactly within a plane of the cube, 'special' equations hold. For example for plane 0 we could say that Math.abs(x) <= 1 AND Math.abs (y) <= 1, which is the same as saying that for all points in plane 0, Math.abs(x) <= Math.abs(z) AND Math.abs (y) <= Math.abs(z). But this is true for plane 2 as well, so how do we differentiate between these two planes? Simple, check the sign of z. If z < 0 and Math.abs(x) <= Math.abs(z) and Math.abs (y) <= Math.abs (z), or even shorter, if z < 0 and Math.abs(x) >= z and Math.abs (y) >= z, we are dealing with points in plane 0.

It gets even better, if the planes are closer or further away from the origin (0,0,0) and in effect the cube is smaller or larger, this will still hold. In order words seeing the z instead of -1 and 1 in the example equations above means that these equations hold for planes parallel to our planes as well. And we can write these kind of equations for each plane. So given any point you can tell in which plane (or rather quadrant) the point lies by looking at "the sign of the coordinate having the largest absolute value" (from page 142 from Mathematics for 3d Game Programming and Computer

Graphics).

Visually this means that given any point contained in the cube you cannot only tell in which plane it lies, but in which quadrant as well. In other words if you view a cube as 6 pyramids put together, you know in which ‘pyramid’ a point lies by looking at “the sign of the coordinate having the largest absolute value”.



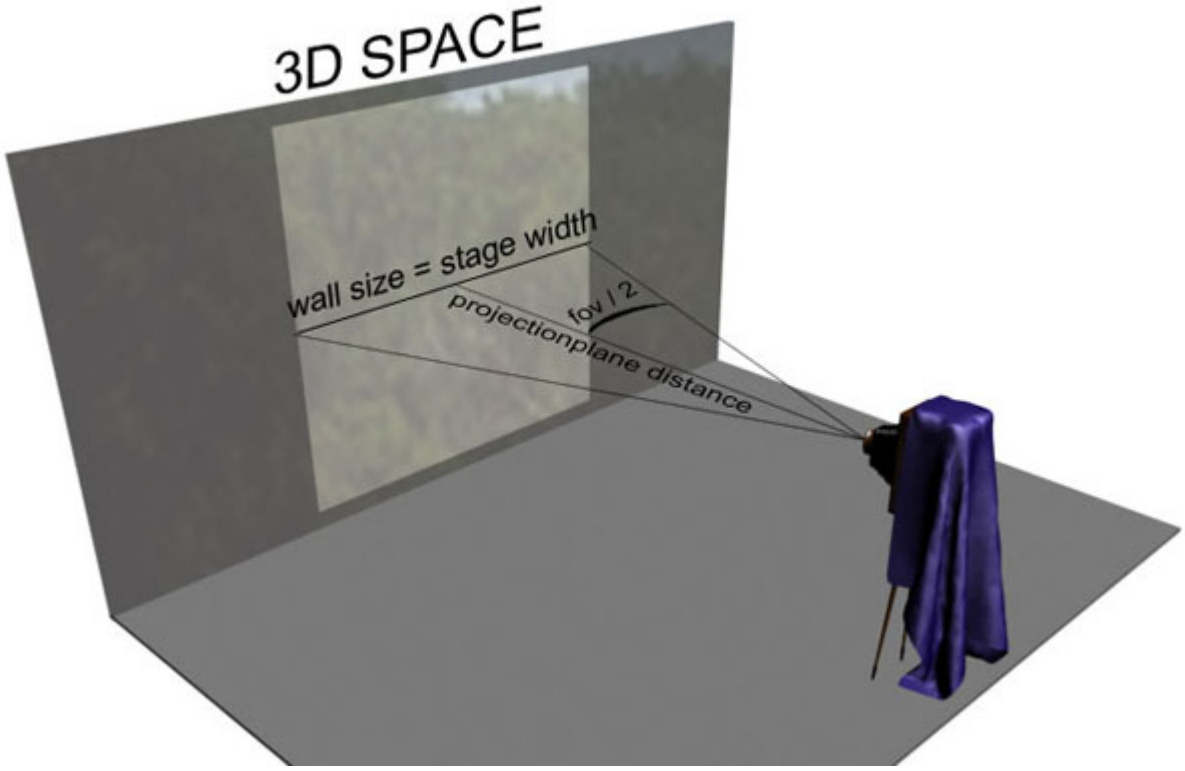
(A cube’s content split into 6 pyramids)

So the next problem presents itself: when we are pointing at a certain pixel in our cube, what is it’s coordinate? We can calculate the coordinate by pointing a vector in the direction of our mouse pointer and see where it intersects one of the 6 planes. But how do we calculate the direction of the vector based on the position of the mouse pointer?

Remember the information we discussed in an earlier post about relating the field of view, wall size and wall distance (projection plane distance)?:

projectionplane_distance = (wallsize/2) / tan (fov_in_radians * 0.5)

Visually:

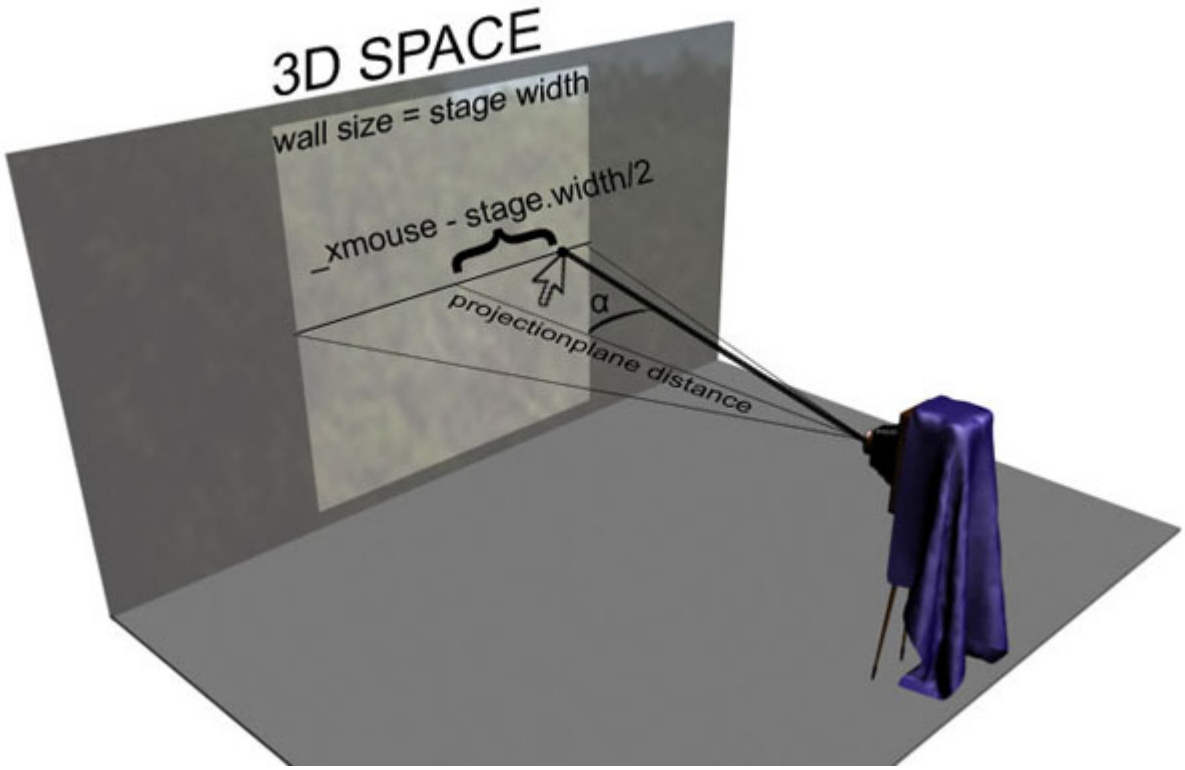


The area within the rectangle on the wall, is the area of our 3d space actually visible on the screen.

(Note that we are using the same field of vision horizontally as we are vertically, like it has been in our panorama’s all this time)

What this did in effect, was given a wall size (stage size in pixels), and a field of vision, calculate the projection plane distance.

Once we have a projection plane distance, we could pick a point in the visible area of our 3d space, in other words a point on our stage, in other words at a certain distance in pixels from the center of the wall/stage, and calculate what the angle of a vector towards that point on the wall would be:

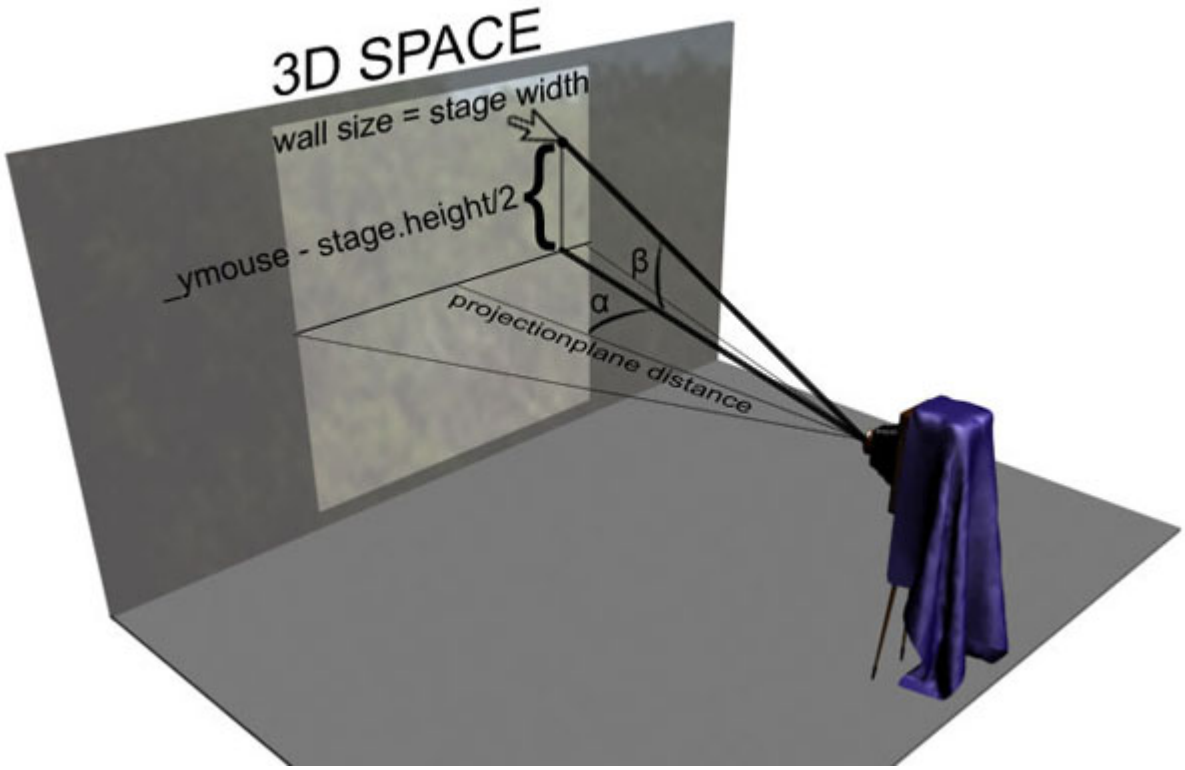


First for the angle alpha which represents the angle between a vector pointing straight at the screen and the mouse x position:

projectionplane_distance = x_offset_from_center / tan (alpha)
=>
projectionplane_distance * tan (alpha) = x_offset_from_center
=>
tan (alpha) = x_offset_from_center / projectionplane_distance
=>
alpha = atan (x_offset_from_center / projectionplane_distance)

The x_offset_from_center is the difference between the center of the stage and the mouse x in pixels.

Of course the mouse position does not only present an offset across the x axis but across the y axis as well:



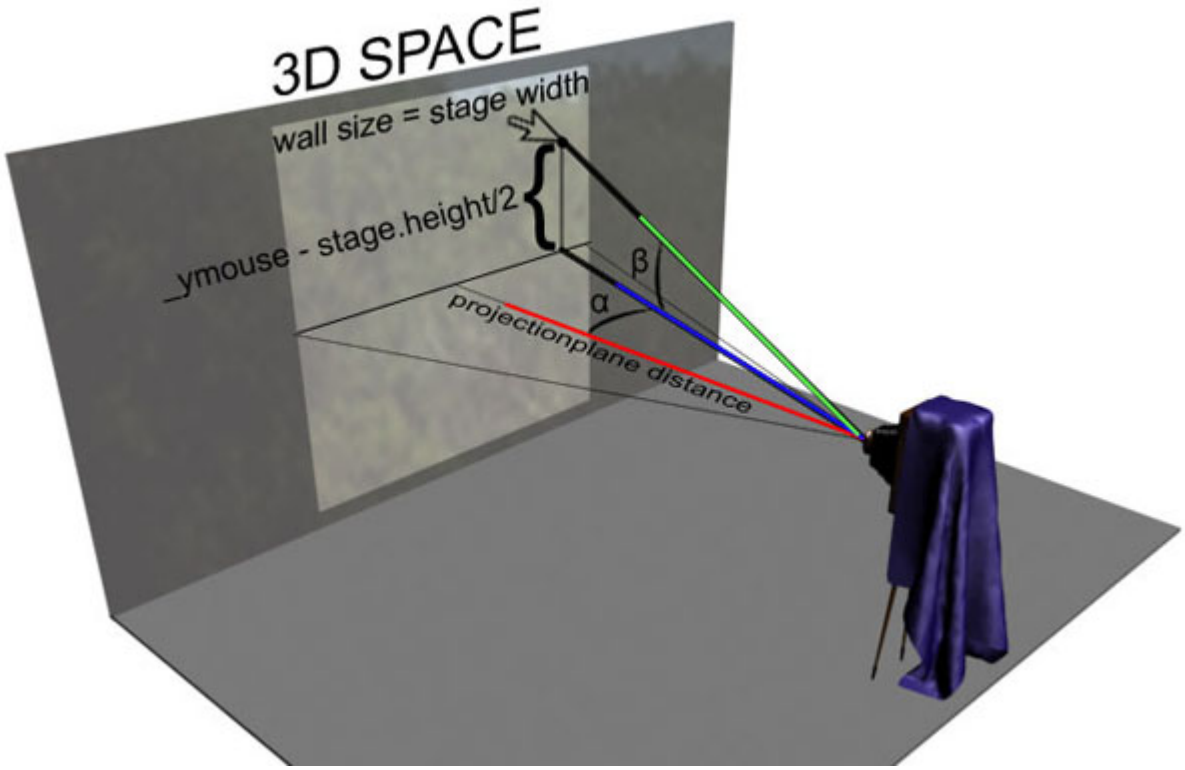
We can see the same formula applies, however instead of the projectionplane_distance we need to use pythagoras on the plane distance and the mouse y offset from the center, to calculate the length of our adjacent, which gives us the following y angle:

```
beta = atan (y_offset_from_center / sqrt (projectionplane_distance^2 + x_offset_from_center^2))
```

So now, given any vector that goes through our camera and points at the center of the screen, we can create a vector that points at the pixel under our mouse by rotating it by the calculated angles.

```
rotatedvector = [0 0 z].rotateY (alpha).rotateX(beta)
```

Visually:



Pick whatever you want for z for example [0 0 -1] or [0 0 -10] or whatever. As long as x and y are zero and we are pointing away from the camera.

So now we have rotated vector, what can we do with it?

If we view our rotated vector as a point, we can determine in what quadrant of the cube we are. However we are faced with another problem. The principle that “the sign of the coordinate having the largest absolute value” determines that quadrant only works for an unrotated cube. And we are constantly rotating our cube as we are looking around in our panorama. So we have to ‘undo’ the rotation of the cube on our rotated vector, which gives us the vector we would have had if we were able to point at the same spot in the unrotated cube and use that one to determine in which quadrant (is quadrant actually the right term or should it be pyramid?) our vector lies:

The example in this post demonstrates that principle. You can look around in our cube, in which I have replaced the shrine images by images containing the plane number, and while you

are doing that a textfield displays in which plane the calculations think you are currently moving your mouse:

(Please give the panorama a moment to load)
Download the source here: 3d Panorama v0.4 (567 downloads)

Next time we will see how we can process this info to find the x and y coordinate local to that plane's texture, which brings us one step closer to our hotspot implementation.