Lab 2: Cameras & Projections

Introduction

This lab aims to provide exercises that help you understand how cameras work in 3D graphics and how the 3D geometry in world space can be transformed and to 2D screen-space.

General instructions.

As before, *future* lab/lecture material in a module called 'magic'. You are free, encouraged even, to look at this code, which will be commented to explain what it is used for and how it works, but it will be explained in the future (or it does not need explanation for this course). Also, there is a module called 'lab_utils' where code that we have already covered is placed – remember that this is there for you to look at to recall how things are done. The labs are designed to tie in with the lectures, but are much more concrete and specific to OpenGL, whereas the lectures aim at a more general level.

When you have finished the tasks, or if you feel uncertain about anything, please speak to a tutor. This is not required to pass, but is a good idea to make sure you have not missed anything important.

Preliminaries

In the previous lab we worked in the two spaces *normalized device coordinates* (NDC) and *screen space.* These spaces are dictated by the API and the physical nature of the screen (and window size).

In this lab we will instead shift to a more abstract frame of reference as we move to 3D proper. Computationally speaking this lab is just a continuation of the previous lab, the goal is always to transform coordinates to *clip space* (which you recall is the 4D homogeneous version of NDC, more details further down).

Since this part is the same, we just need to re-define the spaces and transformations and the code works just as before.

World Space

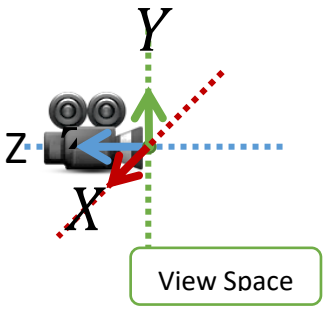
In this lab we will thus move to defining our geometry in world space – which we can arbitrarily decide will be metric (with 1 unit = 1 meter) and have the Y-axis as meaning 'up'. These definitions are completely arbitrary, but keeps the 'up' direction consistent with the screen space we used before (unless we turn the camera upside down or something).

Clip Space

As mentioned in the previous lab, and explained in Lecture 3, a 3D **point** $P = (P_x, P_y, P_z)$ can be expressed in 4D homogeneous coordinates as: (wP_x, wP_y, wP_z, w), for *some arbitrary* $w \neq 0$. This means that if someone hands us a *homogeneous* point: (H_x, H_y, H_z, w), we can convert this to a vanilla 3D point by simply using the definition above as: $(H_x/w, H_y/w, H_z/w)$. The vertex shader produces output coordinates in homogeneous clip space, which are then *clipped* (more on this later) by the hardware and converted to 3D NDC by performing the above division.

Camera or View Space

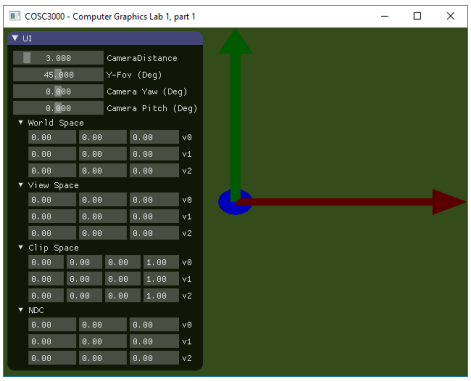
This space is defined as having the origin in the camera position, and the X-axis to the right, Y-axis up, and the Z-axis *straight* ***into*** the camera. That



means, that the *negative* Z axis is forwards. This is a standard OpenGL convention that we will follow. More on this is in Task #2.

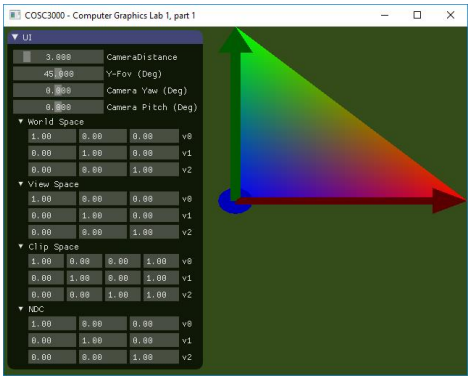
Tasks

1. Define a 3D triangle



Start with '`lab_template1.py`' and change the definition of the triangle ('`g_triangleVerts`') which you recognize from the last lab) such that the corners are on the three coordinate axes, one unit along each.

Notice that we have removed all the translations and scaling from the code again, and th us the resulting triangle is visible on screen, since the positions we defined are in NDC without transformation. This is really more of a coincidence than a feature, and we'll work on transforming the triangle to clip space from world space over the next few tasks.

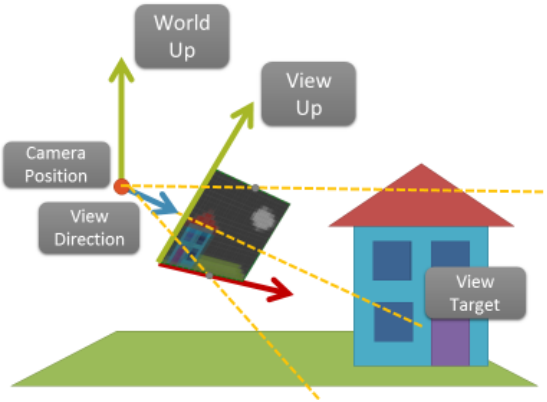


The result should look like the image to the right, note how we added some fragment shader code to use the input world-space positions as colours. Thus each axis has one primary colour, x = (1,0,0) = red, y = (0,1,0) = green, and z = (0,0,1) = blue. Attribute *interpolation* takes care of creating the gradient across the triangle (more on this in a later lecture & lab).

2. Placing the camera in the world

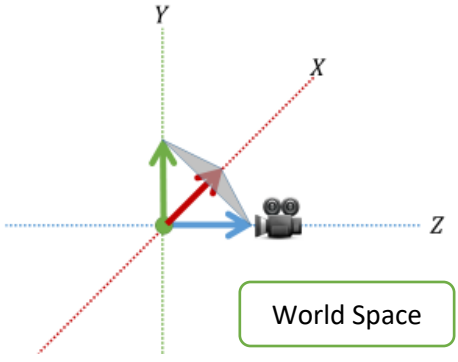
As we talked about in Lecture 3, we want to be able to place and orient a virtual camera, or viewer, in the world. A convenient way to do this is by defining:

1. a position of the camera (or viewer, or eye),
2. a direction to aim the camera in, and
3. an 'up' direction for the camera.



The direction is often expressed by defining a point to 'look at' and the direction is just the difference between this point and the camera location. This is a matter of convenience, and one can be converted to the other. For example:

```
eyePos = [0,0,1]
lookTarget = [0,0,0]
up = [0,1,0]
worldToViewTransform = \
  magic.make_lookAt(eyePos, lookTarget, up)
```



Means 'place the camera 1 unit along the positive world-space z-axis, aim at the origin, and use the y-axis as the general up direction'. Like in the illustration to the right:

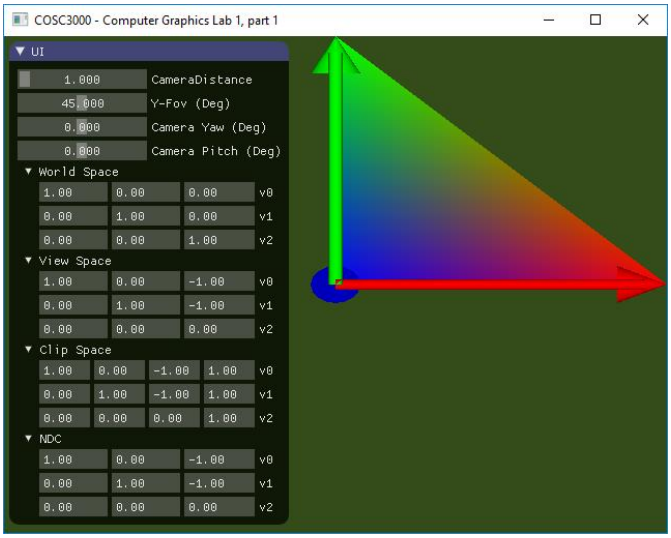
Let's try this out by changing the code to the following:

```
worldToViewTransform = \
          magic.make_lookAt([0,0,g_cameraDistance], [0,0,0], [0,1,0])
```

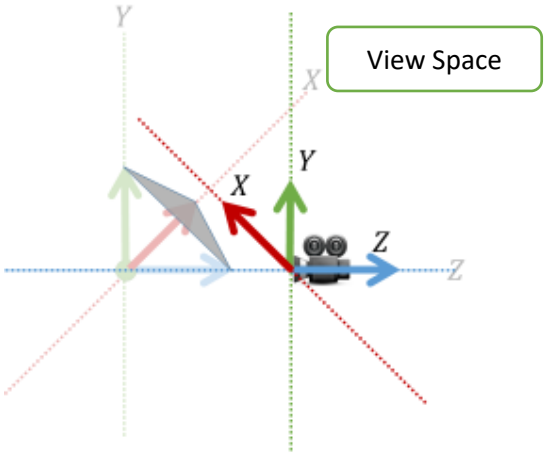
Note how this transformation matrix is combined with the 'viewToClipTransform' (which is currently identity) to implement the full transform from world to clip space:

```
worldToClipTransform = viewToClipTransform * worldToViewTransform
magic.drawVertexDataAsTriangles(g_triangleVerts, worldToClipTransform)
```

'`g_cameraDistance`' is defined to start with the value 1.0, and has an UI slider associated. The UI also shows the transformed positions. The result should look like this. Not much seems to have changed, but if you look at the view space coordinates you can see that they are offset along the z-axis by -1. Play with the camera distance s lider, and see what happens. Why might this be?



Recall from Lecture 3 that view space (or camera space), is defined in OpenGL as having the origin at the camera location and looking down the ***negative*** Z-axis. Consequently the '`worldToViewTransform`' we defined above has the effect illustrated to the right: transforming the triangle into the space of the camera.



This using –Z as forwards *is* kind of crazy, and likely goes back to a desire to have X and Y match up to screen space when OpenGL was defined. In principle you could change this definition (since the hardware only cares about clip space) but you'd have to change the projection as well, and it'd cause a lot of grief when everyone else that uses OpenGL use the default way. So let's just stick with it. '`magic.make_lookAt`' creates a transformation to make this happen.

Coming back to why the triangle dissapears when you move the camera back? Recall that NDC is in range [-1,1], and as corners of the transformed triangle move out of this range, they are clipped. When the last vertex goes below -1 the whole thing dissapears.

Again, the reason we see *anything* is the happy accident of the fact that the camera is initially placed such that the view-space vertices of the triangle just so happens to be in the 'in'-region of NDC. As we change the camera distance this quickly breaks down.

To fix this, we need a projection transformation to explicitly define how the view-space coordinates are transformed to clip space.

3. A very simple perspective projection

Recall from Lecture 3 that the goal of a perspective projection is to simulate the foreshortening that makes objects far away smaller. In the lecture we saw a very simple projection transform:

$\mathrm{M_p} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$, which achieves this. A point $(P_x, P_y, P_z, 1)$ transformed by this matrix gets the Z component duplicated in the W (fourth) component: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = (P_x, P_y, P_z, P_z).$

If we hand this over to OpenGL as the clip-space homogeneous coordinate, the (hardware-provided) transformation to 3D NDC will cause a division by P_z, which sounds exactly like what we want. The one thing which we must change is to use -1 for the z transform as the view-space z-coordinates are negative. Not doing this causes the triangle to be clipped (for reasons that are outside the scope of this lab, and this course, just note that clipping happens in homogeneous 4D and is pretty hard to get your head around the details of!).

Let's put this into practice and see what happens:

```
viewToClipTransform = lu.Mat4([[1,0, 0,0],
                               [0,1, 0,0],
                               [0,0,-1,0],
                               [0,0,-1,0]])
```

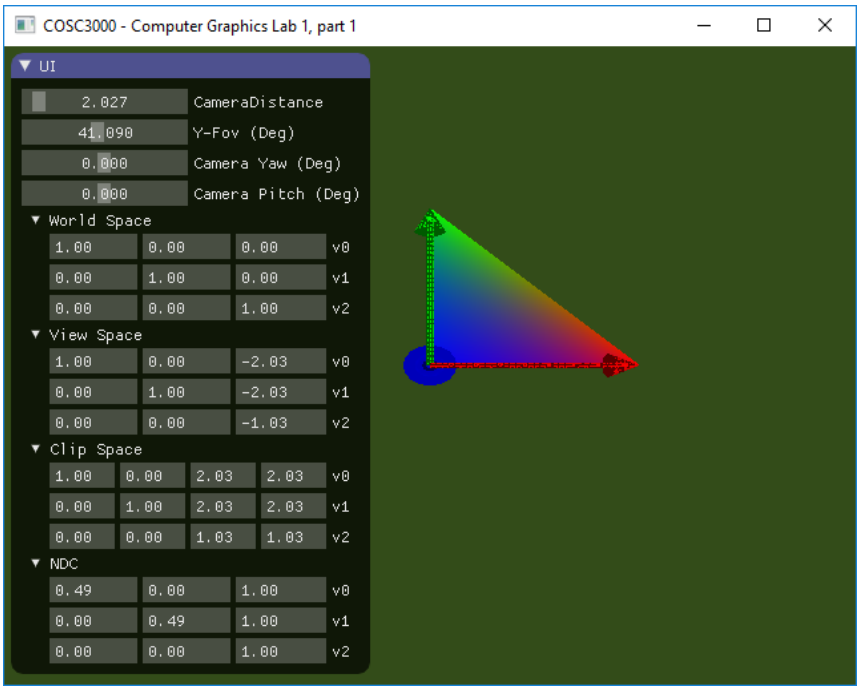
Much to our surprize the triangle is now blue (NVIDIA), or gone (intel HW)! And other things look kind of odd.

How to explain? If you look at the NDC coordinates in the UI you'll see that the last vertex is undefined!

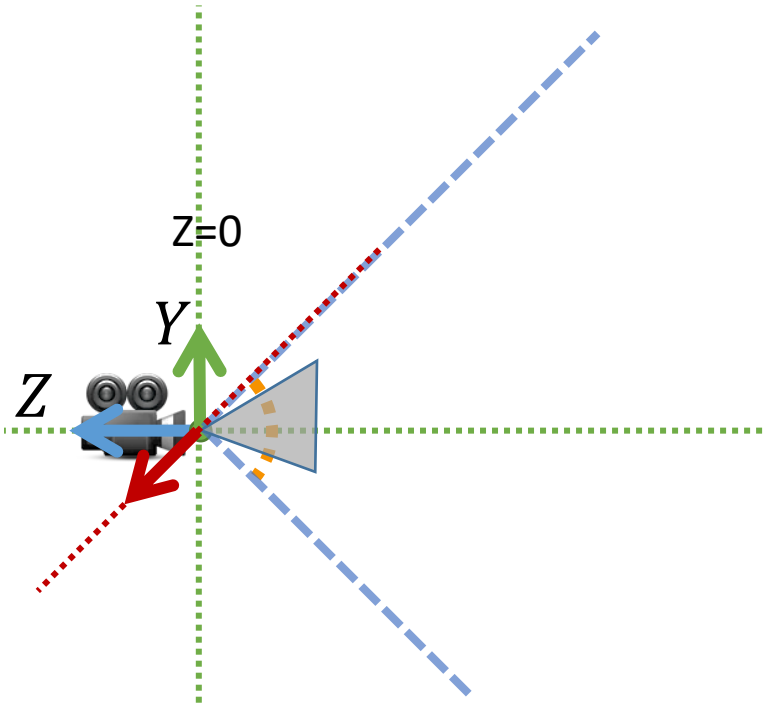
Backing up to view space we see that it has a z of zero, and as this gets copied into the fourth component, w, when the perspective division comes around we get a division by zero which is either represented by infinity (inf) or 'not a number' (nan, which is used for 0/0). Now, OpenGL performs clipping before the division, so it can avoid the infinities (unlike what we show in the UI). It is thus up to the implementation how it handles the z=0 case, and likely comes down to how rounding errors are handled.

Move the camera back ever so slightly using the 'CameraDistance' slider and it appears! Observe the non-zero z coordinates. Now you can see how the perspective projection actually works pretty ok, making the triangle smaller far away.

The problem when Z=0 can be though of in several ways. Intuitively the problematic point is placed with a z coordinate on the plane where *the centre of projection* is. This is in fact defined as a point, a *singularity* with no extent, and so the projected position of something in this plane is simply *undefined*. It's sort of like trying to figure out in which direction something in the middle of your eyeball might be (we don't want to try, right?).



The figure to the right shows the situation. Everythng on the Z=0 plane gets squashed into a single point by the projection, and thus *cannot* be represented.

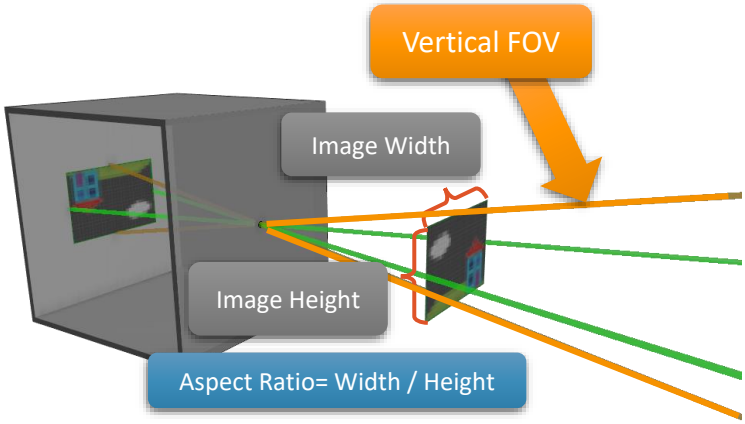


To work around this we simply need to ***avoid trying*** to project things that are located at the plane of projection. To do this in real-time graphics we typically define a minimum distance, the *near plane*, and *clip* any primitives that cross this (we'll demo this later).

Our simple projection is not capable of representing this, and also does not support a varying field of view, or aspect ratio. Note in particular how NDC Z is always 1.0 (look in the UI and move the camera back and forth), which is because clip space Z is always equal to W.

4. A better projection

We'll now move to the full pin-hole camera model as used in real-time graphics (the near and far distances are covered in detail in Lecture #4).



This model gives us far more flexibility and uses the following parameters:

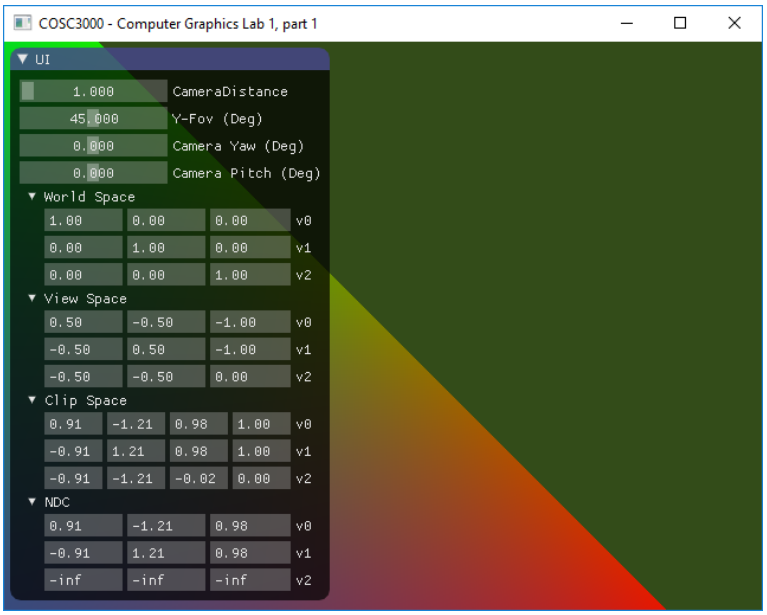
1. Vertical *Field of View* (FOV)
2. Aspect ratio
3. Near and Far clip *distances*

Change the view to clip space tranformation to use the function '`make_perspective`' from the magic.

```
viewToClipTransform = \
          magic.make_perspective(g_yFovDeg, width/height, 0.01, 50.0)
```

Now playing with the camera distance seems to not really have changed much, and the triangle still dissapears when we get to 1.0. This, however, is because we have lined up the camera with the vertex that is in the origin, meaning that it is not removed at this point, but we're trying to draw the thing edge-on. Move the camera up and to the right, or in other words offset the target and eye position by 0.5 in both X and Y, i.e.:

```
worldToViewTransform = \
     magic.make_lookAt([0.5,0.5,g_cameraDistance], [0.5, 0.5, 0], [0,1,0])
```

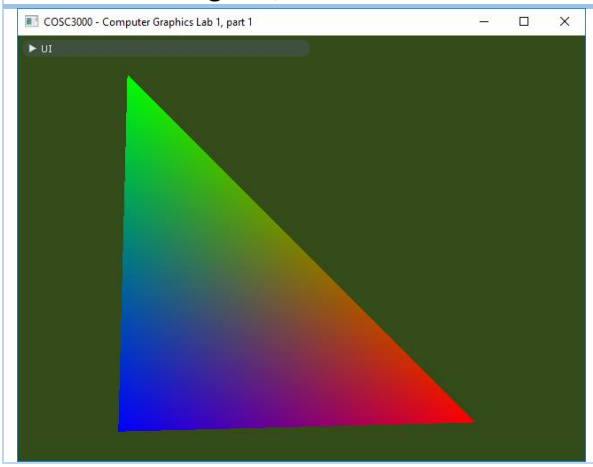
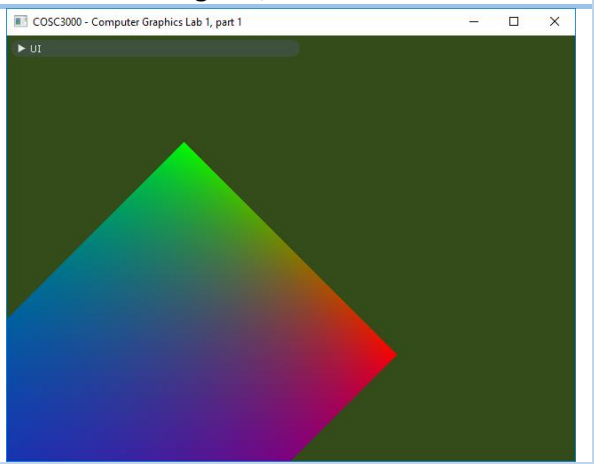


We now get the result to the right. Notice that our NDC coordinate is still infinite. However, the triangle is rendered fine as OpenGL performs *clipping* (as mentioned earlier) *before* the perspective division, and we set the near clip distance to 0.01 – avoiding Z=0.

Try changing the near clip distance to 0.9, what happens?

Notice also how playing with the field of view seems to have very similar effects to the camera distance. This is no different from a zoom-lens in a camera, and much like that this means that a large FOV is akin to a fish-eye lens which warps the image.

For example:

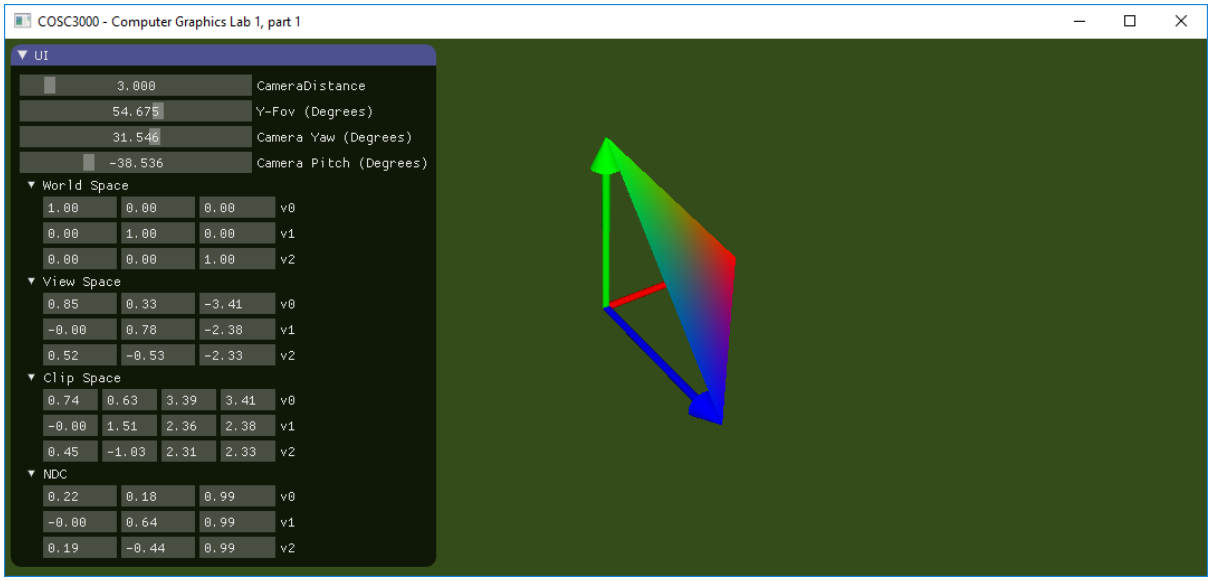
FOV = 3.5 degrees, Camera Distance = 20	FOV=90 degrees, Camera Distance = 1
	

5. Control the camera

Next we want to do some more interesting camera work. Using the model where we work at the level of position and target of the camera makes this simple. Let's rotate the camera around the target (which happily happens to be at the origin). Perform the rotation around the y axis.

```
yawRad = math.radians(g_cameraYawDeg)
eyePos = lu.Mat3(lu.make_rotation_y(yawRad)) * [0,0,g_cameraDistance]
worldToViewTransform = magic.make_lookAt(eyePos, [0,0,0], [0,1,0])
```

Next also add 'pitch' which usually an angle that describes the vertical slant of the camera. Apply this in the appropriate place in the transformation of the eye position. Since the *target location* is always the origin, the camera is kept pointed at the triangle as we rotate the camera postion around.

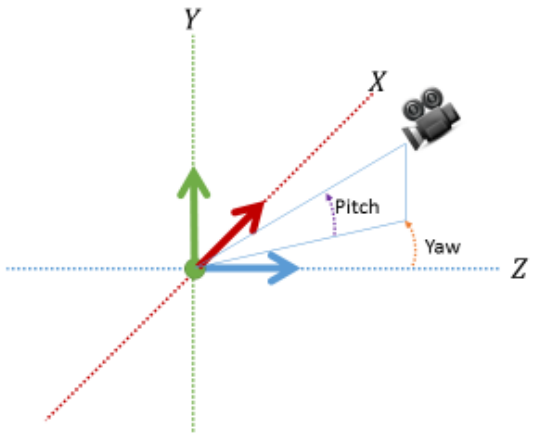
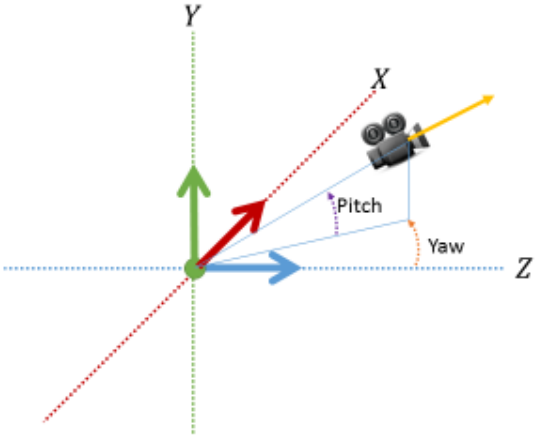


This makes for a very practical model for a camera, when we have a given centre of attention (here, the origin). If we want to move around freely in the world it is less useful.

6. Create a free-moving camera

To create a simple free-moving camera the easiest way is to again use two angles to describe the orientation of the camera, but use a position to describe the location. This sounds similar to what we

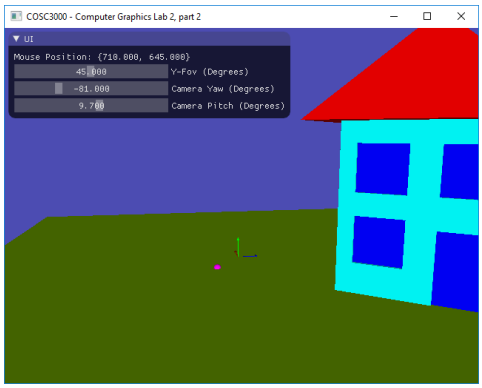
just did. However, instead of interpreting the angles of rotation around the origin, we now interpret them as the orientation of the camera itself:

Previous camera Model	Free-flight model
	

This representation makes it easy to compute incremental updates to the orientation to allow a camera to 'look around' using the mouse, or to turn using the keyboard. In '`lab2_template2.py`' you will find the code needed to update the camera angles. Add the code to compute the direction vector in the '`renderFrame`' function. This is very similar to the previous task.

```
cameraDirection = ???
worldToViewTransform = \
    magic.make_lookFrom(g_cameraPosition, cameraDirection, [0,1,0])
```

If implemented correctly you should get the view to the right. As an aside, note how we have changed from '`magic.make_lookAt`' to '`make_lookFrom`'. The reason is somewhat subtle, but it is more numerically robust when we already have a direction vector (and not a look target position). You should now be able to look around using the mouse: hold down the left mouse button and move the cursor around. The logic for this is implemented in the function '`update`' which is called by the magic every frame.



Now implement back and forth movement in '`update`', this is just a matter of adding the camera direction to '`g_cameraPosition`'. Notice how we scale the speed with 'dt' which represents the number of seconds that has passed since update was last called. This means we will get the same speed of movement regardless of the speed of the computer. The value of '`cameraSpeed`' is set in the update function by looking at the state of 'w' and 's' (or up and down arrow keys).

```
g_cameraPosition += np.array(cameraDirection)[1] * cameraSpeed * dt
```

[1] We cast/convert to '`numpy.array`' as it provides the functionality of multiplying the array element-wise by a scalar value.

Finally, we'll add 'strafing', which is a term from first-person shooter games that means 'moving sideways without turning'. If we think in terms of the camera coordinate system, this means moving along the x-axis.

Since we're already constructing a 3x3 rotation matrix to transform the forwards direction, this is easy. All we have to do is transform the 'sideways' direction (i.e., the x-axis):

```
cameraRotation = \
  lu.Mat3(lu.make_rotation_y(math.radians(g_cameraYawDeg))) * \
  lu.Mat3(lu.make_rotation_x(math.radians(g_cameraPitchDeg)))

sideDir = cameraRotation * [1,0,0]
g_cameraPosition += np.array(sideDir) * cameraStrafeSpeed * dt
```

Now you should have a fully operational camera that can move around and also strafe. Test it by using 'a' and 'd' to strafe. The keys 'w' and 's' are mapped to forwards and backwards, which frees the right hand up to use the mouse to look around.

While at it also play around a bit with the FOV, to see how this affects the visuals in this scene which has a bit more interesting geometry compared to the single-triangle scene we've seen so far.

This way of representing the camera orientation using angles is simple, but has a few limitations. First of all, it gets very confusing unless you constrain the pitch to the range [-90,90] degrees – strictly no loop-the-loops (remove the clamping from the code and try this!). Second, it is easy to go from the angles to a direction vector, but harder to go the other way, which means that if your camera is supposed to follow another object for a while (for example) the usefulness is going down very quickly.

So for a more general camera representation it is better to represent the camera directly using a direction vector and a position. So be able to do arbitrary rotation we also need to store the up direction (since it needs to come along during a loop or roll). In the end, defining a full rotation like this is a good fit for 3x3 matrix, and since we want to represent position as well, why not just use a 4x4 matrix? Well, this is not a bad idea, and as long as we are careful to construct the incremental transformations correctly it works just fine (though this can be a bit tricky to get right). I feel it is slightly more intuitive to keep the position and orientation separate and update them.

Summary

This lab has demonstrated how to set up a camera and projection for use in OpenGL. The techniques for camera control are applicable in any rendering setup, as they do not depend on anything inherent to OpenGL.

If you want an extra challenge, try to modify the free moving camera to represent the orientation and use either a 3x3 matrix or even a 4x4 matrix (in which case you should represent the position also), instead of using angles.