CS3388B: Lecture 9

February 14, 2023

9 Introduction to 3D Graphics

We've made it past two dimensions. Enter, the third dimension. The hardest part about this is having to think about concepts written in two dimensions describing transformations in three dimensions. It's a brain workout to be sure.

9.1 Three-Dimensional Affine Transforms

We already know translation, scale, rotation in two dimensions. Now let's do it in three.

3D scaling

As you might expect we just have a 3 by 3 matrix with now scaling in the z axis added as the third element on the diagonal.

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

Of course, thinking of homogeneous coordinates, we can also define a 3D scale in a 4 by 4 matrix.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D translation

Recall that in two dimensions we needed homogeneous coordinates and a three by three matrix in order to encode a two-dimensional translation. The same is true for three dimensions. We need now a 4 by 4 matrix to encode a three-dimensional translation. It extends naturally from the two-dimensional case. To translate a point by $\vec{t} = (t_x, t_y, t_z)$, we multiply by:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Rotations

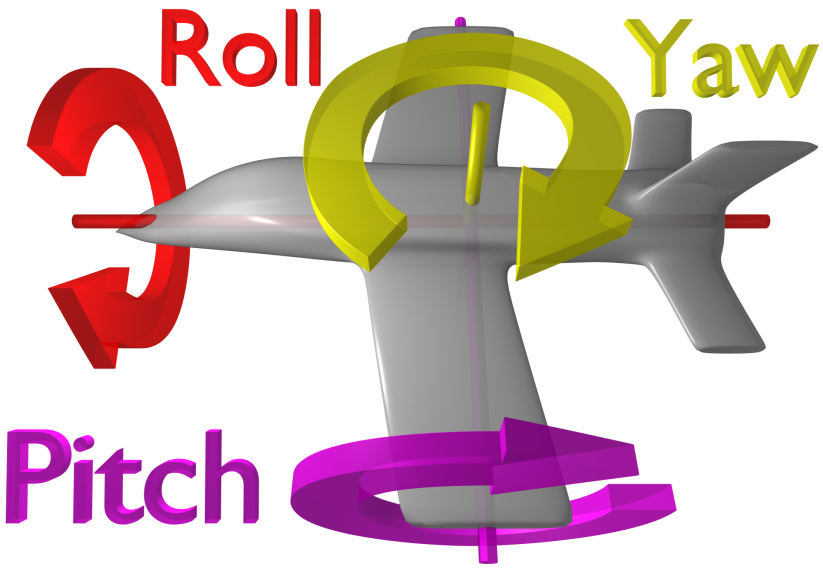
Three dimensional rotations are the biggest difference from the two-dimensional case.

In two dimensions, rotations only have **one degree of freedom**. You can rotate clockwise or counter-clockwise in the plane. This is a single degree of freedom (indeed recall rotating clockwise by θ is the same as rotating counterclockwise by $-\theta$).

On the other hand, consider now three dimensions. How many degrees of freedom do you have in rotation?

...

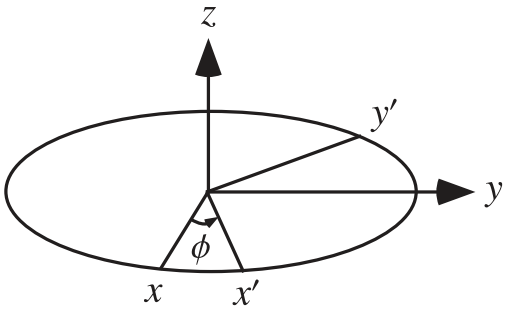
Three!



The building blocks of a three-dimensional rotation are individual rotations around each of the three individual axes.

Let's think about two dimensions again for a moment. One way we though about two dimensional rotations is moving points around a circle of a particular radius. Thus, the point's distance to the origin never changes.

While that's not incorrect, another way to think about it is that we are rotating the X-Y plane around the Z-axis. Indeed, we can view two dimensions as being embedded inside three dimensional space but fixed to the $z = 0$ plane. Then, you are viewing that plane in a line of sight that is parallel to the z-axis.



In two dimensions, we represented this rotation using cosine and sine:

$$\begin{bmatrix} \cos(\theta) & -sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

If we want this same rotation to occur, we need to modify x and y by the same amount, but we want to keep z unchanged. So, make that third row/column be that of the identity.

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This defines a 3D rotation **around the z axis**. In a right-handed coordinate system, we use the right-hand-rule to determine the direction of the rotation around this axis. Make the "thumbs up" gesture. Point your thumb in the direction of the axis of rotation. The direction of your fingers from knuckle to tip show the direction of the rotation.

We can similarly define the other rotations around the x or y axes by fixing one of the axes to be unchanged and rotating around that axis.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

We can combine these matrices (through multiplication) to get a general rotation matrix in three dimensions. Indeed, any three-dimensional rotation can be described as the product of these three **elementary rotations** around the three coordinate axes.

But how do you combine them? There's many possible choices. Naturally, the choice of order of the matrix multiplication changes the result. The most common order is to first rotate around the z-axis, then rotate around the x-axis, then rotate around the y-axis.

In this view, where the "world coordinates" remain fixed and we rotate around those fixed axes, the rotations are called **extrinsic rotations**. We are viewing objects as being rotated about some external **frame of reference**. The actual math of extrinsic rotations follows what we have been doing all along.

$$R_y R_x R_z v = v'$$

will first rotate v around the z-axis, then around the x-axis, then around the y-axis.

In many graphics libraries and other APIs, these three elementary rotations are combined into a single matrix with a function named something like "Euler Angles". Euler angles describe a 3D rotation as the three different angles of rotation around the three coordinate axes.

Note: each library may implement Euler angles differently, even though the inputs may appear the same. `EulerAngles(a,b,c)` will describe a rotation of `a` degrees around the x-axis, `b` degrees around the y-axis, and `c` degrees around the z axis. However, the order in which those rotations ar combined will definitely change the output result. For example, the game engine **Unity** rotates around z first, then x, then y.

```
See MVPTutorial step 0, substeps 1-8
```

Euler's Rotation Theorem

Euler angles are certainly useful, but inherently ambiguous. Another option is given by **Euler's rotation theorem**. It says that any three-dimensional rotation can be described in terms of *some* axis of rotation, and a angle of rotation around that axis. This is called the **axis-angle** formulation of rotation.

The elementary rotations can be seen as special cases of the axis-angle formulation. For example, rotation around the z axis is a rotation around the axis $\hat{k} = (0, 0, 1)$.

The hardest part of this formulation is to find the axis of rotation. But that itself is not *that* hard.

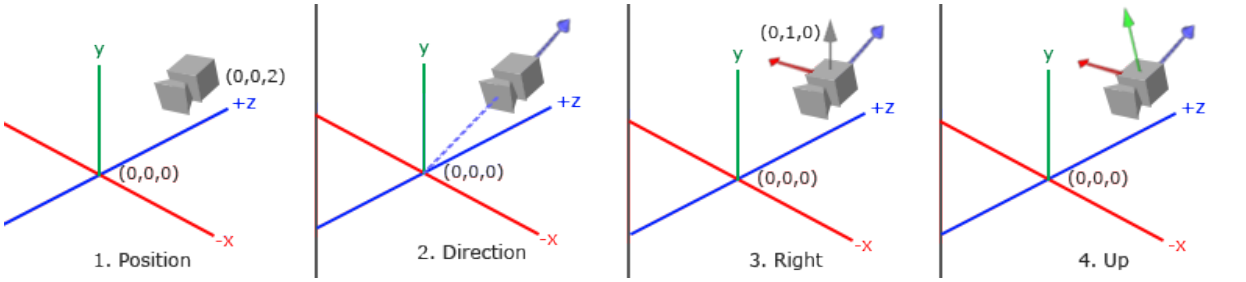
Consider that, for any rotation, the axis of rotation is unchanged by the rotation. For the elementary rotation R_z, the z axis does not move, but the x and y axes do. Therefore, given a general rotation matrix R, the axis of rotation is some direction vector $\hat{u}$ such that $R\hat{u} = \hat{u}$.

I'll skip the details...

The important thing is to always **know what you're doing**. You can always build up a rotation matrix through multiplications of elementary rotations. You can also use `glRotatef(theta, x, y, z)` which uses the axis-angle formulation to define a 3D rotation.

9.2 The Camera

In any graphics library there is a concept of a camera. It is the "thing" from which we are viewing the scene. You can imagine the camera is somewhere in the world, and then what we render and see in the viewport is whatever the camera sees (after mapping to NDC and to the viewport).



The camera can be thought of as existing at a particular point in the world and "looking at" another point. That is, a camera has a position and a direction.

We have seen so far how, with some transformation matrix M we can "move" a vertex from its "local coordinates" to "world coordinates".

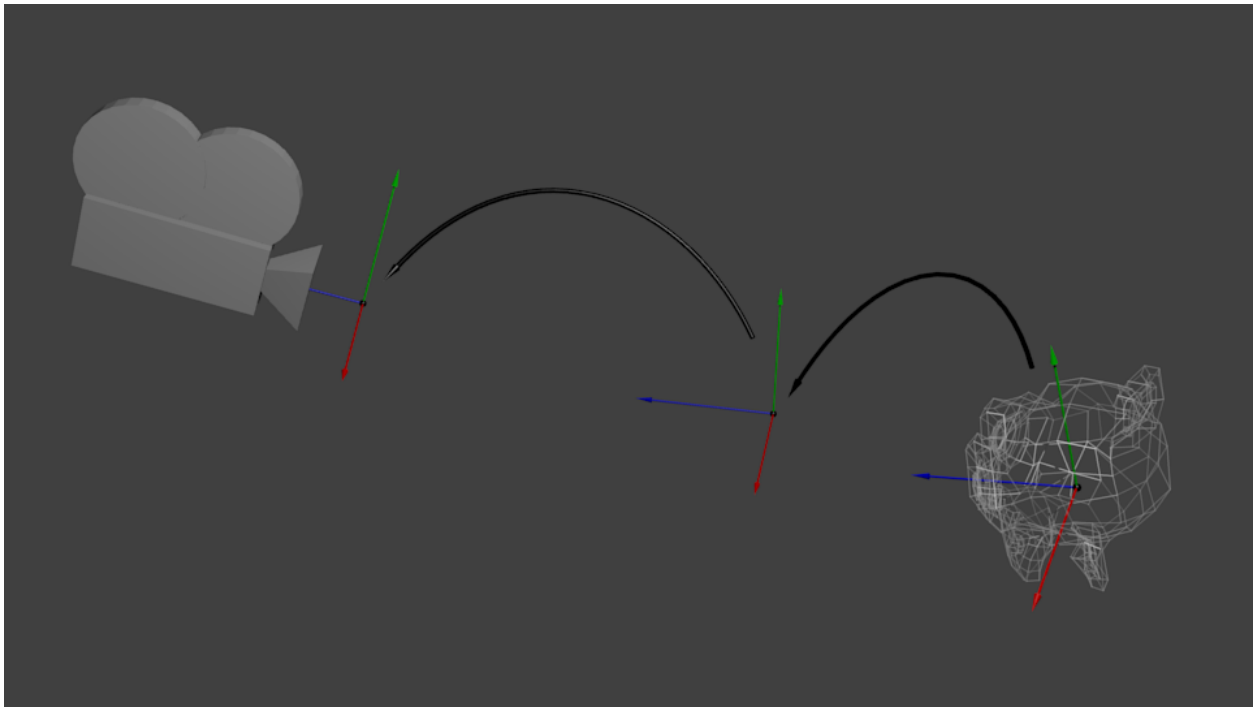
$$Mv_{local} = v_{world}$$

Now, if the camera has a particular position in world coordinates and view direction, then this can be described as an affine transform as well. We call that transform V **view matrix**.

The weirdest thing about graphics to get our head around is that, in reality, the camera never moves. The view matrix gives us a transformation *from world coordinates* **to camera coordinates**.

$$VMv_{local} = v_{camera}$$

In **camera space**, the camera is defined to exist at the origin (0, 0, 0) and looking down the negative z-axis.



Again, our view matrix tells us how to move from world coordinates to camera coordinates. If we want to "move the camera left" we actually "move the world right".

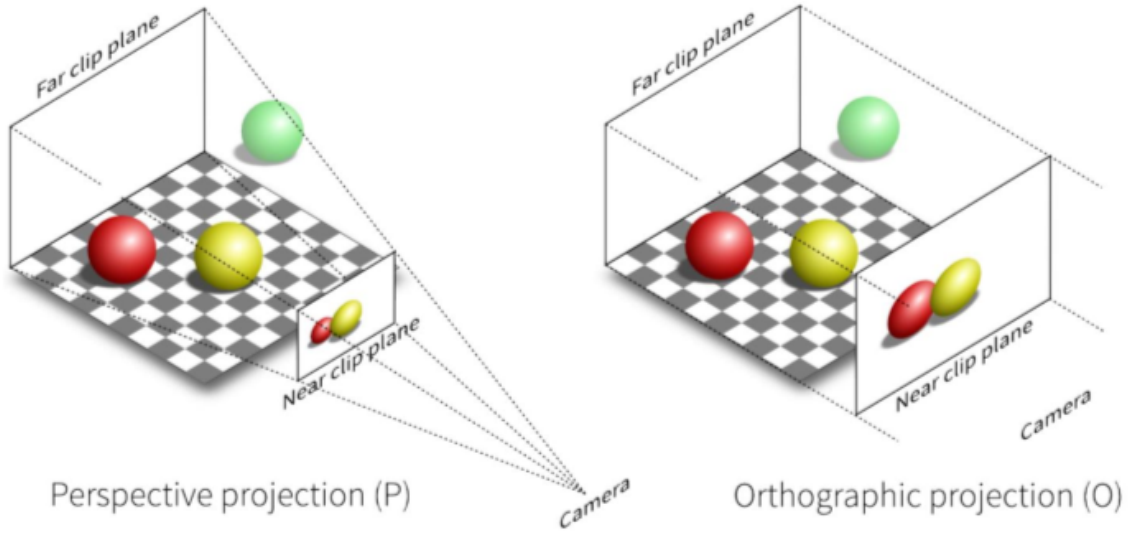
For example, if want the camera to be at $(0, 0, 3)$ in world coordinates, we would define a view matrix V that moves the world 3 units in the negative z direction instead!

$$V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

9.3 Perspective Projection

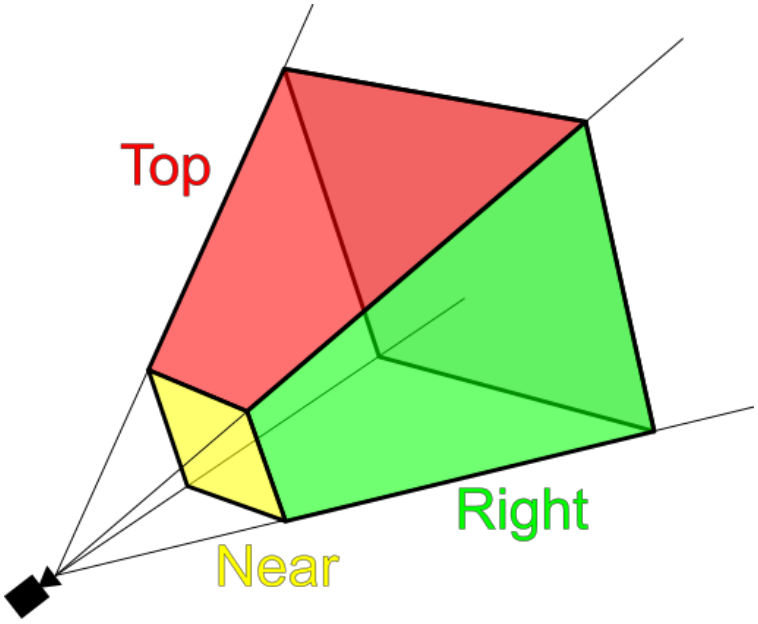
Perspective projection is key to 3D graphics. They create a world that we are used to seeing! The lens in your eyes, the lenses in your cameras, they all use perspective projection. Objects that are closer to you appear larger. **Parallel lines seem to meet at far distances**.

We previously saw orthographic projection. In this projection, the lines of projection were all *parallel*. In perspective projection, the projection lines are not parallel. They instead meet at a particular point.



We previously saw orthogonal projection as a way of defining the "viewing volume" of what gets

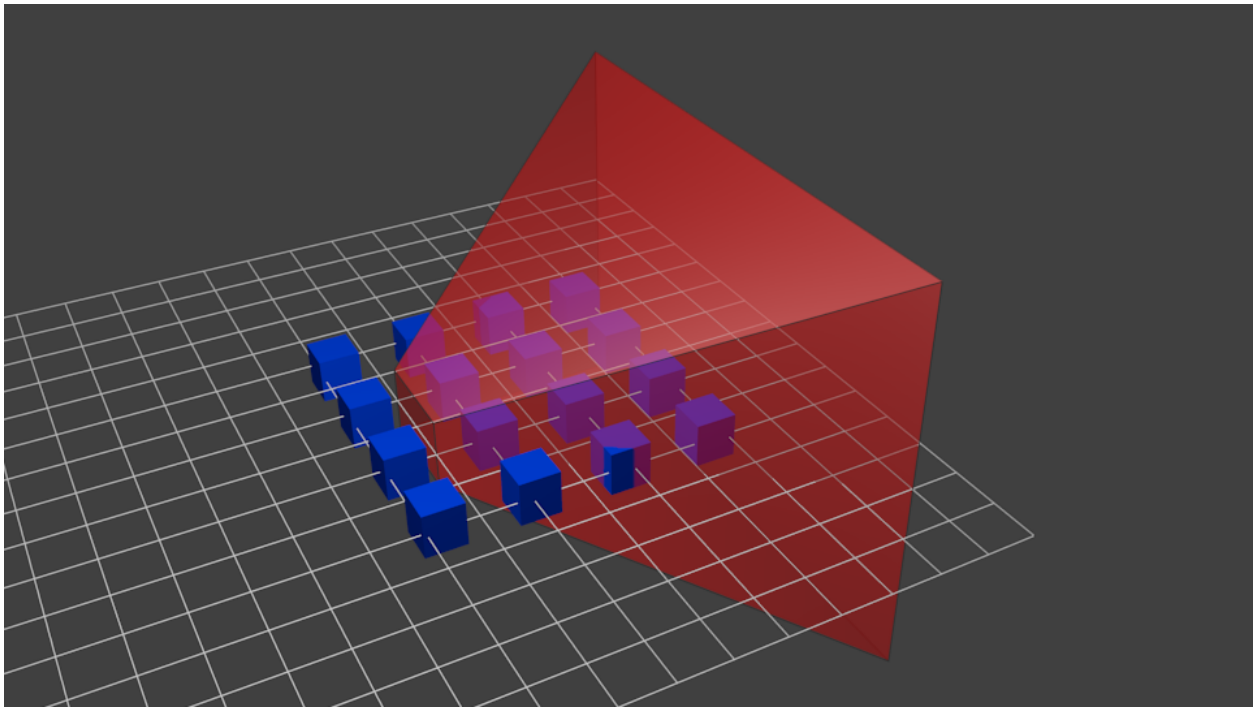
mapped to *from* world coordinates *to* Normalized Device Coordinates. Perspective projection can be thought of similarly, except that the viewing volume is no longer a rectangular prism. Now, it is a **square frustum**.

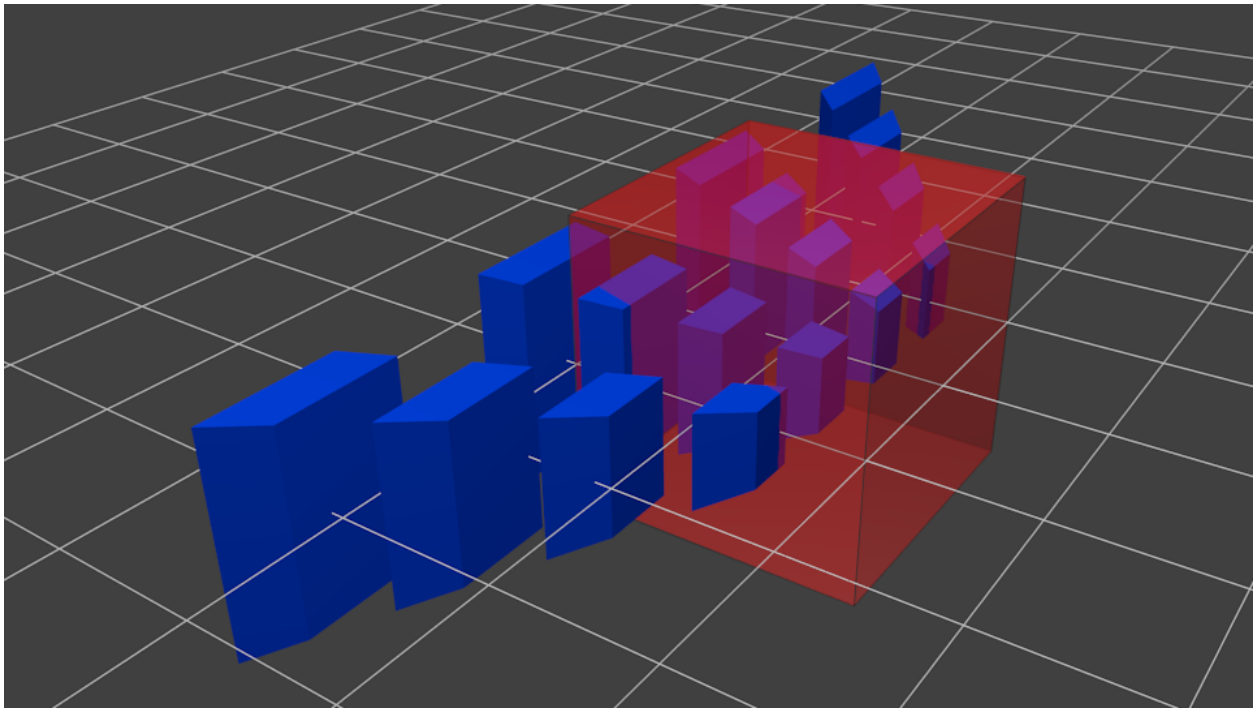


We take this frustum, where the smaller side is nearer to the viewer, and transform it to NDC, forming a cube. The result is that objects closer to the viewer expand, and objects further away shrink.

The matrix which defines this viewing volume is the **projection matrix** P. It transforms from camera space to NDC. (Actually clip space, but we'll return to that).

$$PVMv_{local} = v_{NDC}$$





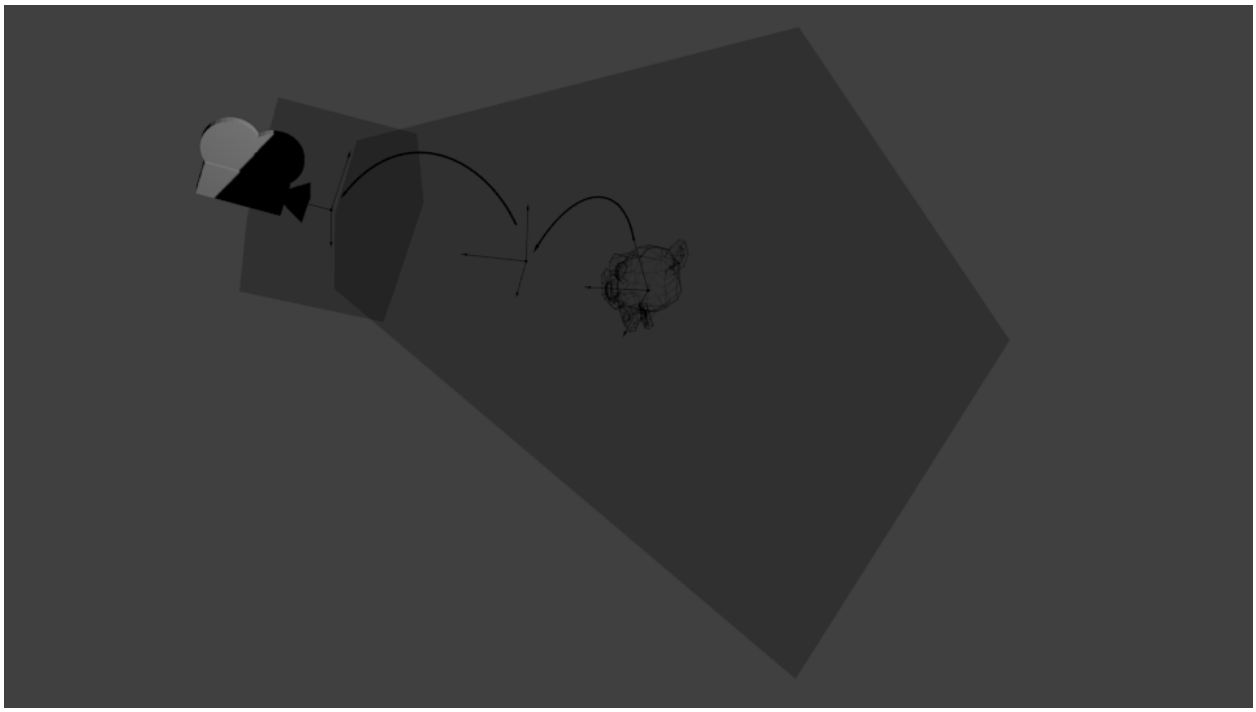
To specify this frustum we require a few parameters:

- **Near clipping plane**: the plane perpendicular to the x-y plane at $z = near$ which defines the front of the frustum.
- **Far clipping plane**: the plane perpendicular to the x-y plane at $z = far$ which defines the back of the frustum.
- **Field of view**: The angle between the lines of projection. Typically this is measured in the vertical direction. 45°is a reasonable number.
- **Aspect ratio**: This defines the square/rectangle shape of the near and far ends of the frustum. Since we have a vertical field of view, this specifies what fraction the vertical view is compared to the horizontal view.

The `glm` (OpenGL math) library does the work for us:

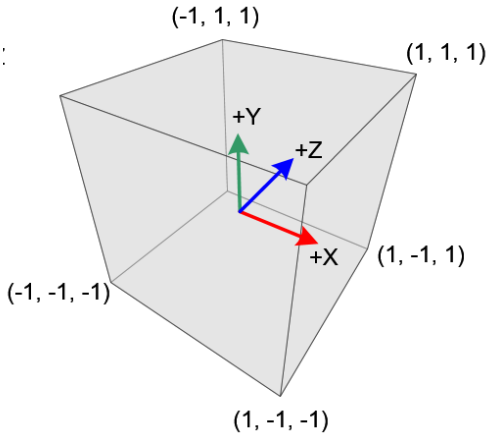
```
glm::mat4 P = glPerspective(fovy, aspect, near, far);
```

Note: we specify the near and far clipping planes to have positive values. That is, they measure the distance of the plane from the camera. However, in reality, that are at **negative z coordinates** since the camera is looking down the negative z axis.



9.4 Time to fix some previous lies

Normalized Device Coordinates



This previous figure shows something interesting. Notice that in NDC the z-axis is now facing the opposite direction! Indeed, **Normalized Device Coordinates is a left-handed system**.

Return to the in-class demo. We had a cube drawn with opposite corners (0,0,0) and (1, 1, -1).

We saw that without perspective projection (i.e. the projection matrix was the identity), we saw the **front** face of the cube, at $z = 0$. I told you this was because the camera sits at the origin and looks down the negative z axis. So, the $z = 0$ face is obviously closer to the camera, and is therefore the face seen, compared to the back side of the cube at $z = -1$.

That was a lie.

In reality, the Normalized Device Coordinates are rendered as follows. The x and y coordinates directly map to window coordinates using the viewport matrix. So, all values of x between [-1,1] get mapped to window coordinates $[x_0, x_0 + w]$ (see the viewport lecture for notations). Similarly, y between [-1,1] get mapped to $[y_0, y_0 + h]$.

What about z? The $z = -1$ plane is the "near" plane and the $z = 1$ plane is the "far plane". So, the objects with $z = -1$ *should* get rendered "in front of" objects with $z > -1$.

However, this is only true if you tell OpenGL to do **depth testing**. Otherwise, OpenGL ignores the z component of NDC entirely. Whatever is drawn *last* is rendered "on top".

So, return to the cube with one face at $z = 0$ and one face at $z = -1$. With a projection matrix being the identity matrix, the cube's face at $z = -1$ is actually *closer to the viewer* than the cube's face at $z = 0$.

Aha! So the camera is not really at the origin and looking down the negative z-axis. If we think of the camera as what is "seen" and what will be drawn, then the camera is at $(0, 0, -1)$ in NDC and looks down the positive z-axis.

Yet, in another point of view, the camera is indeed at the origin and looking down the z-axis. Why? It comes from the perspective frustum and **clip space.**

Clip Space

It is *not* the case that:

$$PVMv_{local} = v_{NDC}$$

In fact, we have:

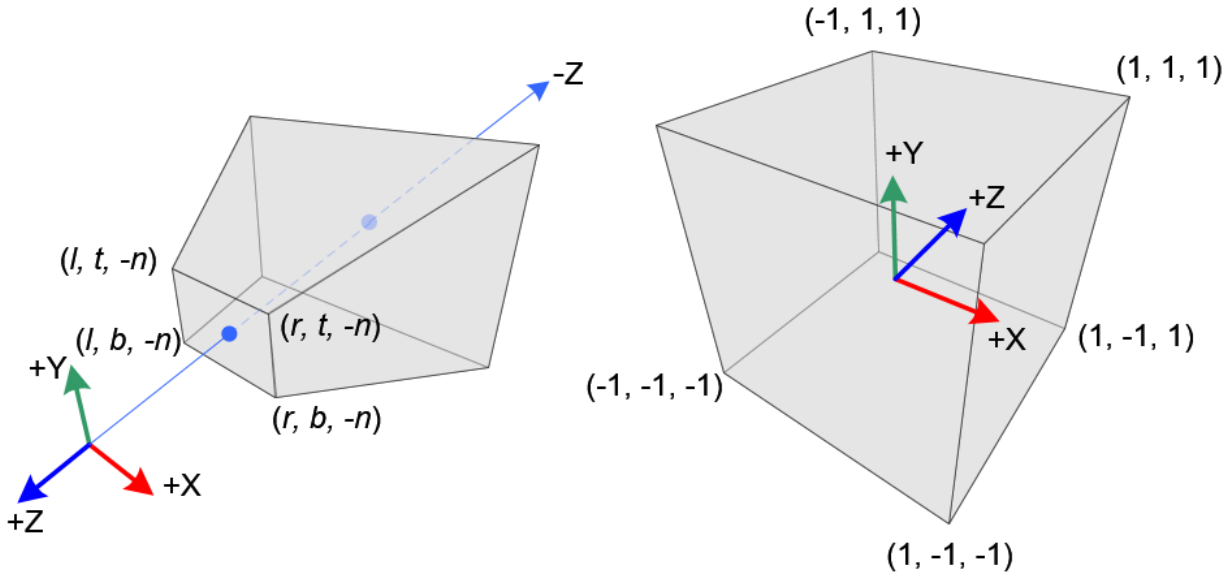
$$PVMv_{local} = v_{clip}$$

What is clip space you ask? Let us look more closely at the projection matrix and what actually happens. For a frustum which is symmetric (it's left and right sides are the same distance from the z-axis; it's top and bottom sides are the same distance from the z-axis), a perspective projection matrix is defined as:

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where the corners of the *near side* of the frustum has corners:

$(-r, t, -n)$, $(r, t, -n)$, $(r, -t, -n)$, $(-r, -t, n)$. In the below diagram, $b = -t$, $l = -r$.



With this view of the projective frustum, the camera is indeed at the origin and looking down the negative z-axis. Just as the diagram on the left of the previous figure shows.

But, we have one automatic/invisible step between going from frustum to NDC. Notice that the perspective projection matrix we showed earlier is the first matrix which **modifies the 4th homogeneous coordinate**.

In homogeneous coordinates, we have 4 coordinates: (x, y, z, w). Much like in two dimensional homogeneous coordinates $w = 1$ encodes a "point" and $w = 0$ encodes a "direction". Thus, a point with $w = 1$ can be translated while a direction cannot.

When we multiply a point $(x, y, z, 1)$ by the perspective projection matrix, we get a point in **clip space**. In particular, notice that $w_{clip} = -z_{camera}$ (check the 4th row of the projection matrix above). Thus, a larger w means *further away from the camera* since, this time around, we are again thinking of the camera being at the origin and looking down the negative z axis.

It is within clip space that **clipping** occurs. That is, we remove any vertices which **would become** outside NDC. The key is **perspective division**.

We calculate NDC from clip space like this:

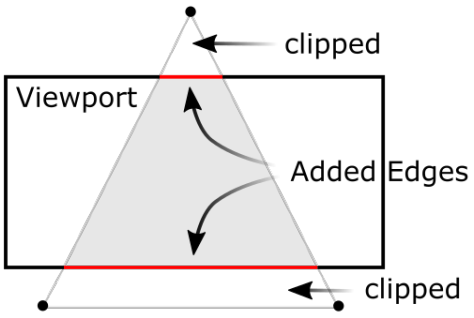
$$\begin{bmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{bmatrix} = \begin{bmatrix} x_{clip} \ / \ w_{clip} \\ y_{clip} \ / \ w_{clip} \\ z_{clip} \ / \ w_{clip} \end{bmatrix}$$

Dividing by w is called the perspective divide. Now, we know NDC is defined to be [-1,1]. And we compute coordinates in NDC by perspective division. Therefore, a coordinate x_{clip} is *inside* the [-1,1] NDC volume if $-w_{clip} < x_{clip} < w_{clip}$.

So, it is easy to test which vertices to keep. We only keep vertices which have all three:

$$\begin{aligned} -w_{clip} < x_{clip} < w_{clip} \\ -w_{clip} < y_{clip} < w_{clip} \\ -w_{clip} < z_{clip} < w_{clip} \end{aligned}$$

When primitives have vertices partly outside and partly inside, OpenGL **clips** the primitive so that its new vertex is directly fits in NDC.



For the curious reader, here is a special detail. It may confuse a non-careful reader. Each vertex belongs in its *own clip space*. What does that mean? Each vertex has its own value of w_{clip}. So the test for whether or not to clip a vertex is dependent on that vertex itself and its particular value of w_{clip}. Every vertex belongs in a "shared" camera space. Every vertex belongs in a "shared" NDC. But, to get from camera space to NDC, each vertex passes through its own clip space and checks against its own w_{clip} to find out if it gets clipped or not.

9.5 The Camera: Part 2

The camera (or what we conceptually think of as the camera) is extremely important to graphics. It's worth revisiting. In one point of view the camera always exists at the origin and look down the negative z-axis. In another point of view, we can actually place the camera in world coordinates as if it was an object or any other model.

The latter point of view is most practical. The only trick is that we have to remember that our **view matrix** is defining how to get from world coordinates to camera coordinates. So, if we want to think of the camera as an object in the world coordinates, it must be that the view matrix is the **inverse matrix** of the camera's world position.

Let's consider the case study of a classic operation: "camera look at". This operation looks to build a view matrix which represents the camera at a particular point in world coordinates and looking towards another point, also specified in world coordinates.

Let's say we want to position the camera so that it is placed at (e_x, e_y, e_z) (e for "eye"). The camera should then be rotated to look at (t_x, t_y, t_z) (t for "target"). Our eventual V matrix needs to this have a translation, to put the camera at (e_x, e_y, e_z) in world coordinates, and then some sort of rotation to make it face the target.

Let's start with translation, that's easy. Again, we want the V matrix to actually encode the inverse of the camera transformation. Whats the inverse of translation? Just multiply by -1.

$$V_{trans} = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now we need to specify the rotation. We could figure out all the elementary rotations needed to get the camera to point as the target. Or, we could consider another view point for transformation matrices. Recall from Lecture 4 that one way of understanding transformation matrices is not how they transform points from one coordinate system to another, but rather how it transforms the whole coordinate system of one space to become the coordinate system of the destination space.

In this case, our destination space is camera space, where the camera is at the origin and looking down the negative z axis. In camera space, this negative z axis has an associated direction

$-\hat{k} = (0,0,-1)$. Therefore, in camera space, the positive z axis is defined as $\hat{k} = (0,0,1)$. Relative to the world coordinate system, this $\hat{k}_{camera}$ has an associated vector cz_{world}, the camera's positive z-axis, specified in world coordinates.

What is that direction? Well we have an eye position and a target position we want to look at. The direction the camera is looking is thus a displacement vector between the target and the correct eye position.

$$\vec{d} = \vec{e} - \vec{t} = \begin{bmatrix} e_x - t_x \\ e_y - t_y \\ e_z - t_z \end{bmatrix}$$

$\vec{e} - \vec{t}$ gives a vector *from* $\vec{t}$ *towards* $\vec{e}$. But... this is exactly what we want. We want the camera to be looking down *negative z*. Therefore, $\vec{e} - \vec{t}$ describes the positive z axis of the camera space, in world coordinates. Of course, we should normalize this vector, so $cz_{world} = \vec{d}/|\vec{d}|$.

So we've now described the z axis of the camera's coordinate system relative to the world coordinates. We do the same for the x and y axes. It follows a simple trick that is a consequence of 3D rotations.

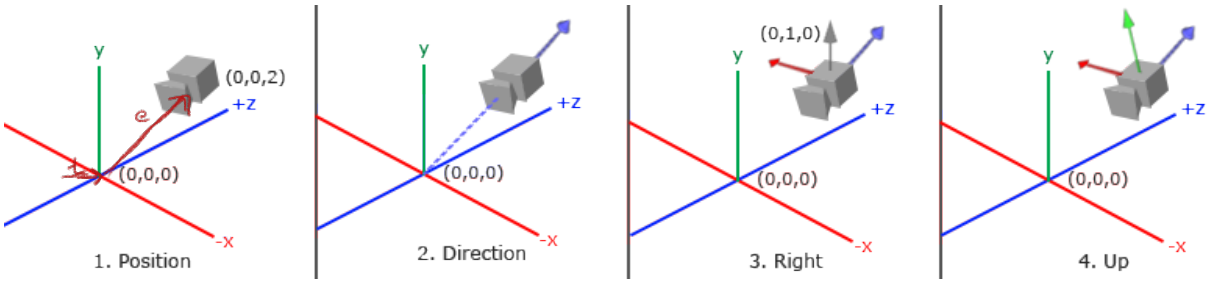
To "look" in a particular direction leaves you still with one degree of freedom. To see why, pick any point on a the wall in the room you are currently in. Look at it. Rotate your head side-to-side while still looking at that point. What happens? Your vision **rotates around that point**. This is exactly the **axis-angle** formulation of 3D rotations. The look direction gives us the *axis*. Now we have to specify the angle.

One way to specify this angle is to specify the proper x and y axes of the camera coordinate system. We do so with a simple trick. Look at your point on the wall again. This time, put the crown of your head facing toward the ceiling. Call this "upwards". Now, to look at your point you may have to move your eyes a little bit up or down in their sockets. That's okay, as long as we keep the crown of your head pointing upwards. If your head is the camera, your eyes are looking in the look direction and your right ear is facing in the direction of the positive x axis. Your ear points out to the right.

Now, we will let our head drift away from the crown pointing directly up. Bring your chin up or down so that you're still looking at your point on the wall, but now your eyes are looking "straight ahead", but maybe your chin is no longer pointing parallel to the floor. That's okay. This is the key observation: your right ear hasn't moved! By tilting your chin up and down, you essentially rotate your head around the x axis (the axis coming out of your right ear). Once you've moved your chin so that the line of sight is "straight out of your head", the crown of your head now doesn't point directly up toward the ceiling, but the crown of your head still points directly "upwards" in the coordinate system relative to your eye balls.

In summary:

1. Specify a position of the camera in world space.
2. Specify a look direction $\hat{d}$ as the normalized $\vec{e} - \vec{t}$.
3. Let the "up" direction be directly up toward the ceiling and then let your right ear point out toward the positive x axis.
4. Rotate around this x axis so that your line of sight returns to be $\hat{d}$ and that the crown of your head is no longer pointing directly upward.



Now, how do we actually do the math to compute steps 3 and 4? Cross products!

Given $\hat{d}$, we compute $\hat{r}$ as the cross product between $\hat{d}$ and (0, 1, 0), that is, the cross product between $\hat{d}$ and "directly up" in world coordinates. Then, we compute $\hat{d} \times \hat{r}$ to get the "up" direction relative to the camera, call it $\hat{up}$.

Phew. That was a lot. Put it all together now:

$$V = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & -e_x \\ \hat{up}_x & \hat{up}_y & \hat{up}_z & -e_y \\ \hat{d}_x & \hat{d}_y & \hat{d}_z & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ta-da! This is the view matrix that (conceptually) positions the camera at location (e_x, e_y, e_z) in world coordinates and looks toward $\vec{t}$.