

Lesson 7: Cameras

Lesson: Introduction to Cameras

[[http://commons.wikimedia.org/wiki/File:Swampy_fisheye_lens_\(crop\).jpg](http://commons.wikimedia.org/wiki/File:Swampy_fisheye_lens_(crop).jpg)]

[http://commons.wikimedia.org/wiki/File:Camera_focal_length_distance.gif twice - animated]

[

<http://commons.wikimedia.org/wiki/File:Elevation-Panth%C3%A9on-Lanterne-Paris-France-1881.jpg>]

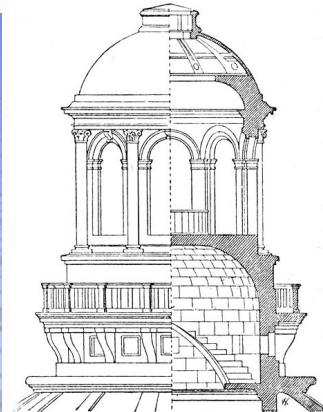
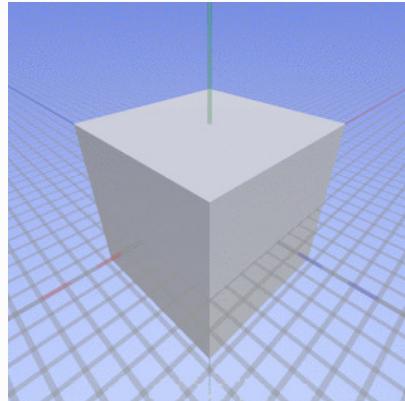
[

fisheye

projections]

perspective

orthographic]



There are all sorts of different ways to view the world. Here are three: a fisheye lens, a perspective view, and an orthographic view. These are all different types of projections. A projection maps locations in the world to locations on the screen.

A fisheye projection is possible to simulate in computer graphics - hey, just about anything can be simulated in computer graphics - but it's rarely used in practice. These other two projections are more popular, for a number of reasons. First, they act similarly to how a camera captures a picture, so we're used to them. Straight lines stay straight, unlike the fisheye lens. All in all, the math is simpler.

A perspective view is what we usually associate with seeing the world. Distant objects get

smaller.

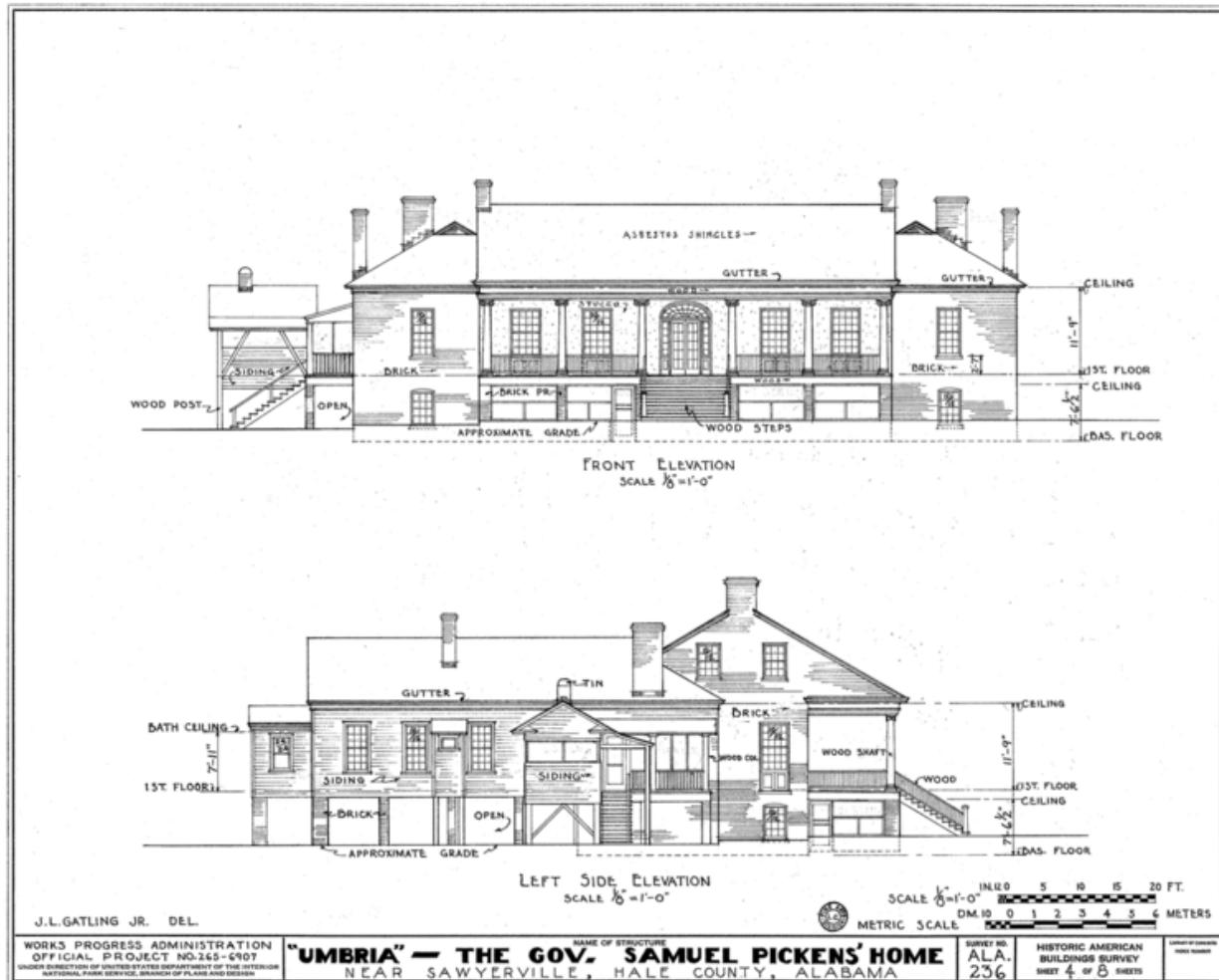
An orthographic view is much more common in design software. In an orthographic view, objects do not get smaller as they get more distant. Parallel lines in the real world stay parallel in this view, as opposed to perspective. In perspective, such lines meet at vanishing points. [first and last we'll say about them.]

[Additional Course Materials:

The [fisheye lens]([http://commons.wikimedia.org/wiki/File:Swampy_fisheye_lens_\(crop\).jpg](http://commons.wikimedia.org/wiki/File:Swampy_fisheye_lens_(crop).jpg)), [perspective](http://commons.wikimedia.org/wiki/File:Camera_focal_length_distance.gif), and [the elevation drawing](<http://commons.wikimedia.org/wiki/File:Elevation-Panth%C3%A9on-Lanterne-Paris-France-1881.jpg>) are from Wikimedia Commons.
]

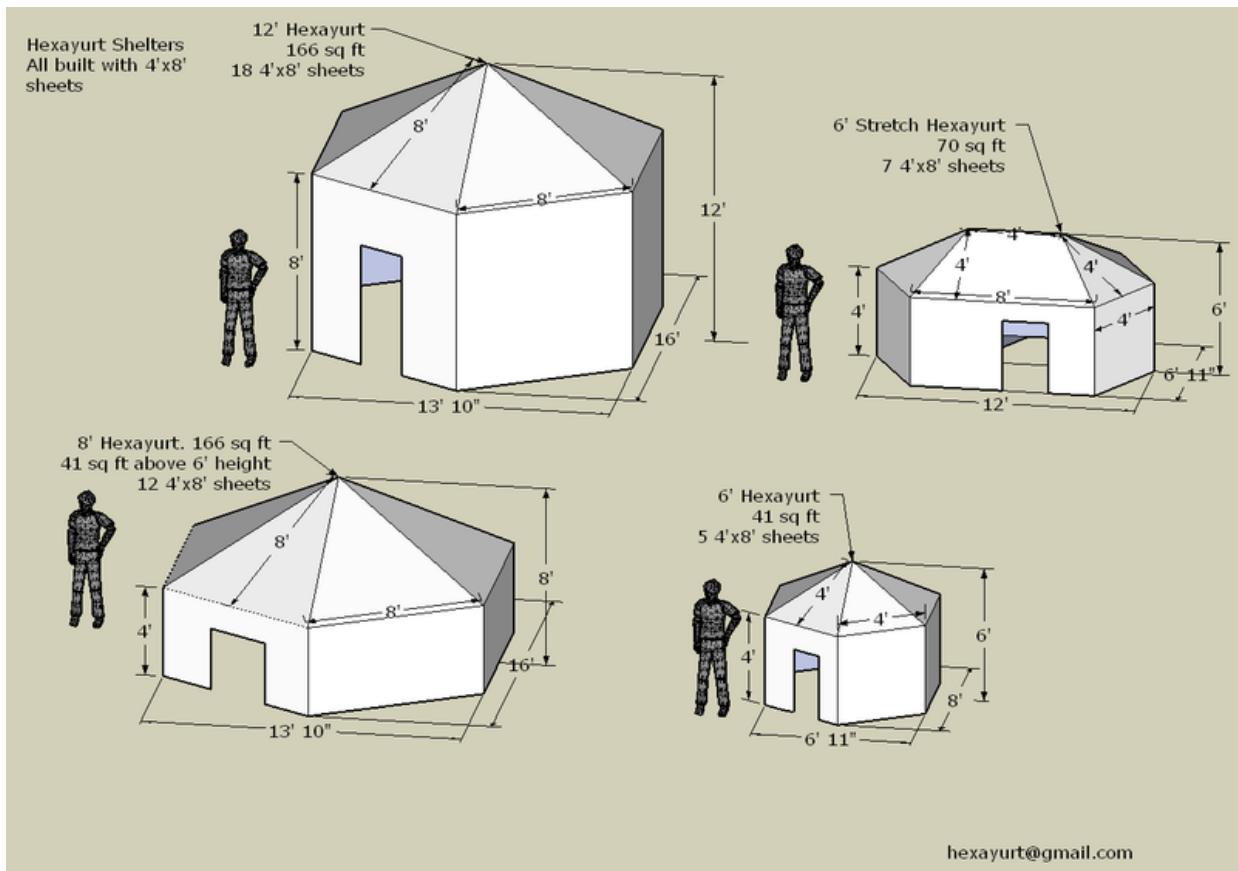
Lesson: Orthographic Camera

[
http://commons.wikimedia.org/wiki/File:Umbria_Plantation_-_Architectural_drawing_of_front_and_east_elevations.png]



This orthographic projection isn't normally how we directly perceive the world, but it's a form we commonly use nonetheless. For example, assembly instructions usually show an orthographic view. Architectural plans showing a front, side, or top view of a building use this sort of projection. You can take measurements off these types of views. Having the distortion caused by perspective projection would most likely hinder our understanding of what we're seeing.

[http://commons.wikimedia.org/wiki/File:All_hexayurts_web_dimensions.png]



hexayurt@gmail.com

An orthographic camera can be used to examine objects from angles other than directly along an axis.

A demo using an orthographic camera follows. I find that I'm so used to seeing with a perspective view that I perceive an optical illusion. If I zoom out by using the slider and look at the whole scene, my visual system tells me that the front edge of the grid is definitely shorter than the back edge. Try it yourself and let me know in the forums what you think.

[Additional Course Materials:

See the [Wikipedia article on graphical projections](http://en.wikipedia.org/wiki/Graphical_projection) for more on this subject, including a wide range of other architectural projections.

Images from Wikimedia Commons: The [Umbria Plantation](http://commons.wikimedia.org/wiki/File:Umbria_Plantation_-_Architectural_drawing_of_front_and_east_elevations.png) and [hexayurt shelter](http://commons.wikimedia.org/wiki/File:All_hexayurts_web_dimensions.png) images are from Wikimedia Commons.

]

Demo: Orthographic Projection

[unit7-orthographics_demo.js]

Lesson: Three.js Orthographic Camera

To define an orthographic camera you do two operations. One is to define a volume in space, the other is to move and orient that volume of space as you wish.

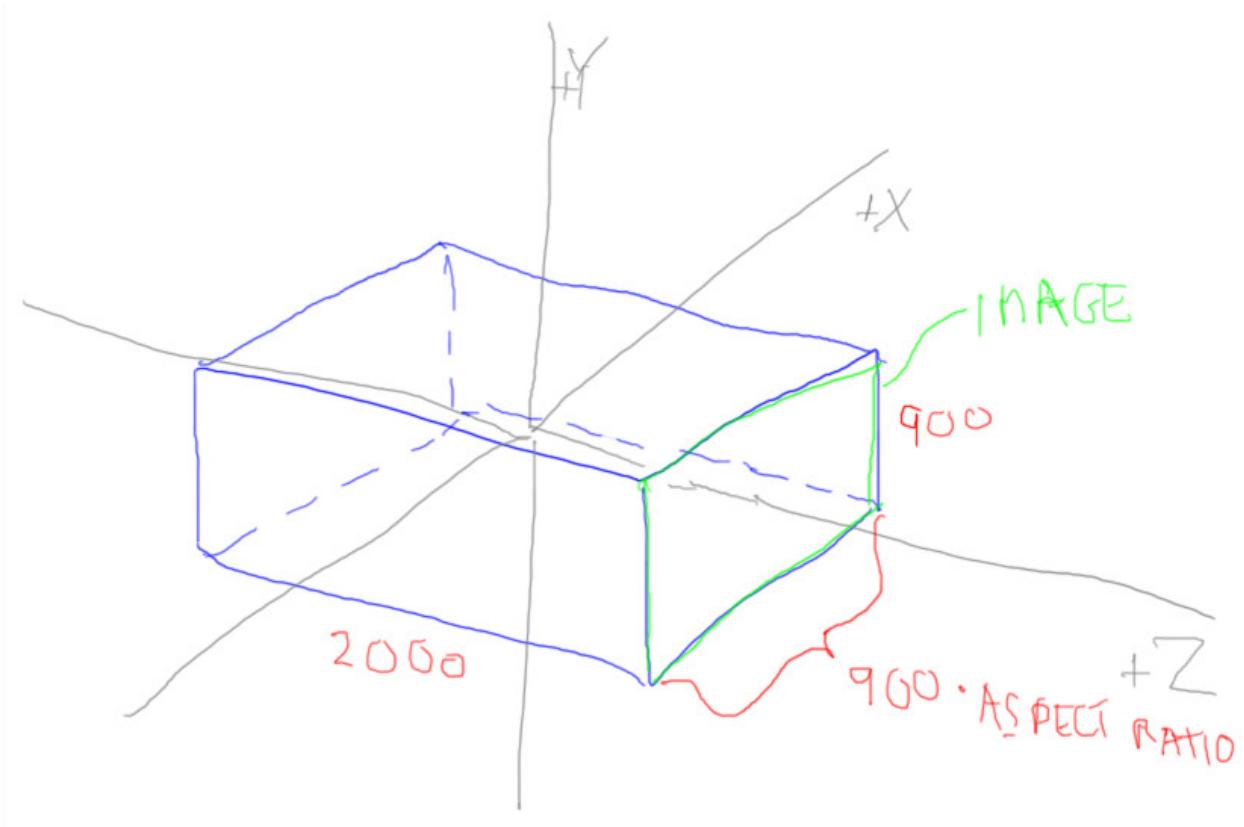
```

viewSize = 900;
aspectRatio = window.innerWidth/window.innerHeight;
// OrthographicCamera( left, right, top, bottom, near, far )
camera = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );

viewSize = 900;
aspectRatio = canvasWidth/canvasHeight;
// OrthographicCamera( left, right, top, bottom, near, far )
camera = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );

```

[draw box showing this view, show viewSize and aspectRatio*viewSize, and near and far. Put the box in a separate layer, as we'll move it later. Makes it nice and canonical. DEFINITELY PUT LABELS IN SEPARATE LAYERS.]



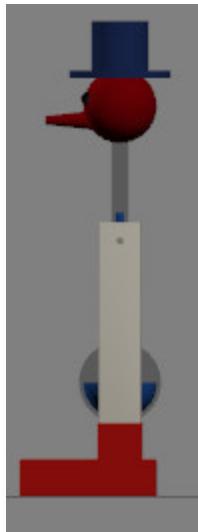
This first piece of code creates the orthographic camera and defines the rectangular box of space it is viewing. This box is defined by the left to right width limits, top to bottom height limits, and front to back limits, all in world-space coordinates. These correspond to the X, Y, and Z axes. The camera can be thought of as being positioned on the +Z end of the box defined, looking down the -Z axis, with up being the +Y direction, as usual.

For convenience, I define a `viewSize`, which is how much vertical space I'd like to fit in the view. The `aspectRatio` is just that, what's called the “**aspect ratio**” of the window. It describes how wide the view is compared to how high it is. For example, if your view was 1000 pixels wide and 500 high, the aspect ratio would be equal to 2. In other words, you'd want your box twice as wide as it is tall. If you don't, you'll get a stretched view of the world, which is probably not what you wanted. What all this code does is define a box in space, in this case it happens to be centered around the origin.

By the way, I've seen some demos just use the canvas width and height *directly* as the camera's size. Don't do this: you're setting the camera's size in your scene in world coordinates, which is most definitely not the same as the width and height of your canvas in pixels.

----- new page -----

If we stopped here, we'd get a side view of the bird.



What we'd now like to do is position and orient this camera in space to make a more interesting view.

```
camera.position.set( -890, 600, -480 );
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(0,310,0);

camera.position.set( -890, 600, -480 );
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(0,310,0);
```

This code does that. The first line moves the box by this amount, it's a translation, the same as for a normal object. The cameraControls code does two things. First, it sets up what sort of mouse controller you want for the camera, if any. I came up with this custom OrbitAndPanControls class, extending the OrbitControls class I found in three.js. This control keeps the camera oriented so that the +Y axis is always up. For an more unconstrained but trickier-to-use control, the TrackballControls class could be specified instead.

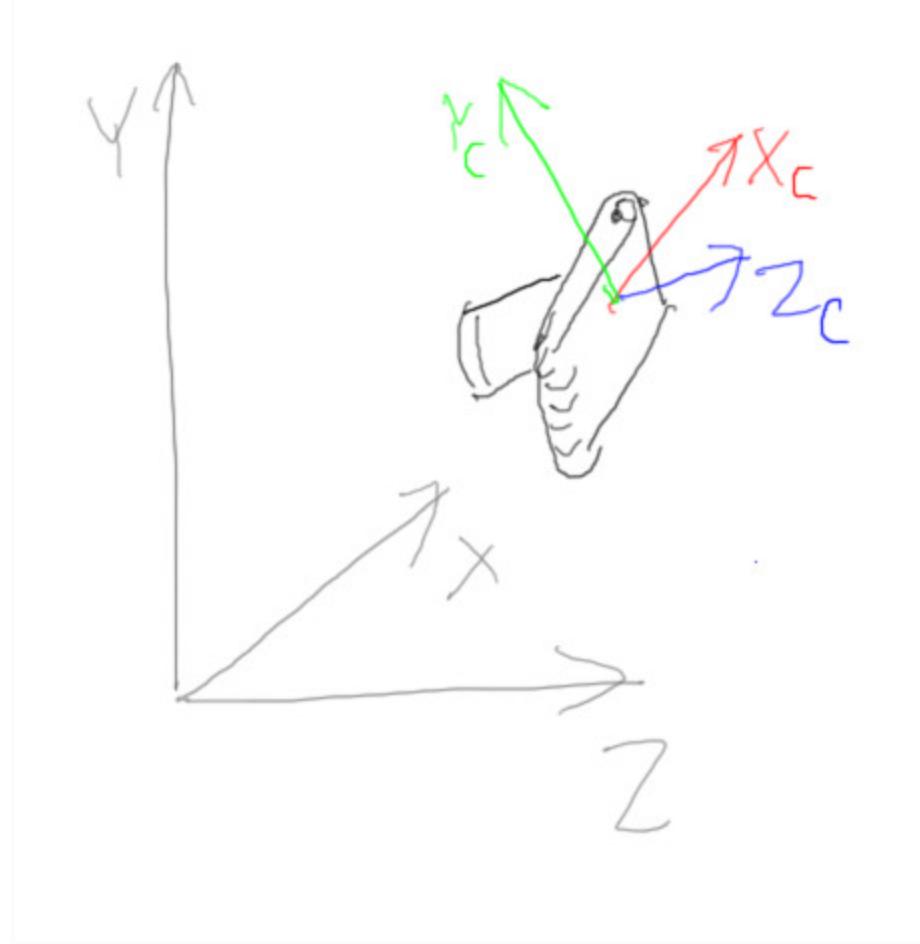
In either case, the next line sets the target for the camera. The target is the location being looked at. This defines a new Z axis direction for the camera. The +Y axis is kept pointing up as best as possible - we'll talk about that later, when we look at the matrices forming this camera.

Once we've positioned this box in space, it defines what part of the scene we want to render. Whatever happens to be inside this box is what we'll see on the screen. The z-buffer is applied as usual so that objects that are closer to the camera cover up what's behind them.

This is just an overview of the basic idea: the camera defines a chunk of space defining its limits, and this chunk then gets positioned and oriented in space to make an image. I'll go over this process in detail in the following lessons.

Lesson: LookAt

[draw moving the camera to a location first, then defining the box it looks at. Leave room to right for cross product later! Draw camera itself in a separate layer, so I can reuse. ***NOTE: in real drawing, leave off Xc and Yc! These come later***]



Let's look a bit more in depth at how this box in space actually gets positioned and defined internally. First, we define our camera object as looking down the world's -Z axis and oriented so that the top of the camera is facing along the +Y axis. The camera starts at the origin, just like other objects.

The first step is to move the camera to where we want to make an image and also tilt and orient it as desired. This is similar to how we have been orienting objects all along: we define some sort of rotation and translation to get the camera into place. We ignore scale - we're just trying to set up the camera in space.

Giving the camera a position is simple enough. To compute the rotation needed to orient our camera we use what's called a "lookat" system. This is pretty much what it sounds like: you define a target location for what direction you want the camera to look. This defines the -Z axis for the camera's frame of reference, in other words, how it's pointing in the world.

[bring toy camera into view and show it being oriented. Note how it could spin on its axis.]

However, we could spin the camera along this -Z axis defined for it - we don't know which way is up if we're given just a position and a target.

[draw Y axis for camera. Draw camera and all in separate layer. ***NOTE: in real drawing, leave off Xc and Yc! These come later***]



When defining the camera's location like this, we also pass in an "up" vector of some sort. This vector gives us some guidance as to which way to orient the top of the camera.

Typically in three.js we'll pass in a vector such as $(0,1,0)$, the $+Y$ axis, as the up vector. Given our view vector and our up vector, we'd like to form a matrix that defines a whole frame of reference for our camera. We know where we want the Z axis to look along, but we also want to know where the camera's Y and X axis end up. By specifying a general up direction, we do this.

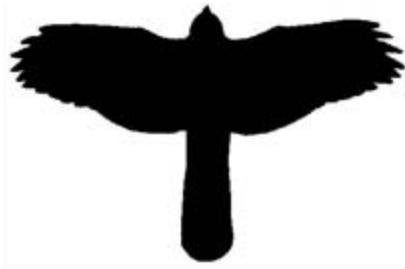
[draw to side: show cross product of "up" and $+Z$, giving X_c , (draw X_c), then Z and X , giving Y_c .]

For example, say we have a camera pointing in some direction, looking along its negative Z axis. If we give an up direction, we can find the direction pointing to the right of the camera, which is the X axis, by taking the cross product of this Z axis and our up vector. Now we have two axes locked down, camera Z and camera X . We can find the true up direction, the camera's Y axis, by then taking the cross product of the X and Z axes. [right handed rule with hand] Remember that the cross product of two vectors will give us a vector that is perpendicular to both of them. This may all sound a bit odd and like a lot of work, but is extremely common - it's probably the place

where I use the cross product the most.

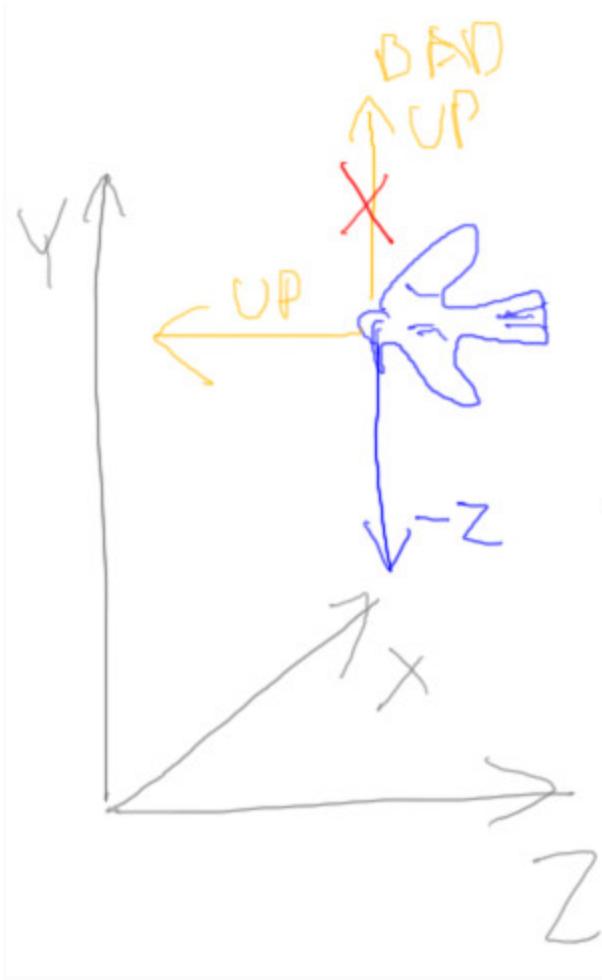
The whole point here is that, given a view direction and an up vector as a “hint”, we can get a frame of reference for the camera.

[Make a hawk model out of an outline.]



We'll often use the world's +Y axis as our up vector “hint”, but it all depends: say we're a hawk soaring over some terrain. We're looking straight down, which would be looking down the Y axis of our world.

[draw hawk looking down. Show its Zc lookat axis, and XYZ axes of world]



If we used the world's Y axis as our hint, it doesn't make much sense: we've defined the -Z axis the camera's looking along and the Y axis for the camera's up direction as vectors in exactly opposite directions. This is not very useful! The cross product is the vector 0,0,0. In this case our up vector is more like the direction pointing out the top of our hawk's head, the direction it is flying. Now that the up vector is not parallel to the lookat direction we can form a good frame of reference for the hawk.

Exercise: Three.js Lookat

`CameraControls.target`

`-2800, 360, -1600`

There's in fact a lookat method on the camera itself, but most of the time we use the

CameraControls class to let us move the camera around with the mouse.

If you run the exercise code, you'll see a grid and not much else: the drinking bird has left the view. Why did he leave? Where did he go?

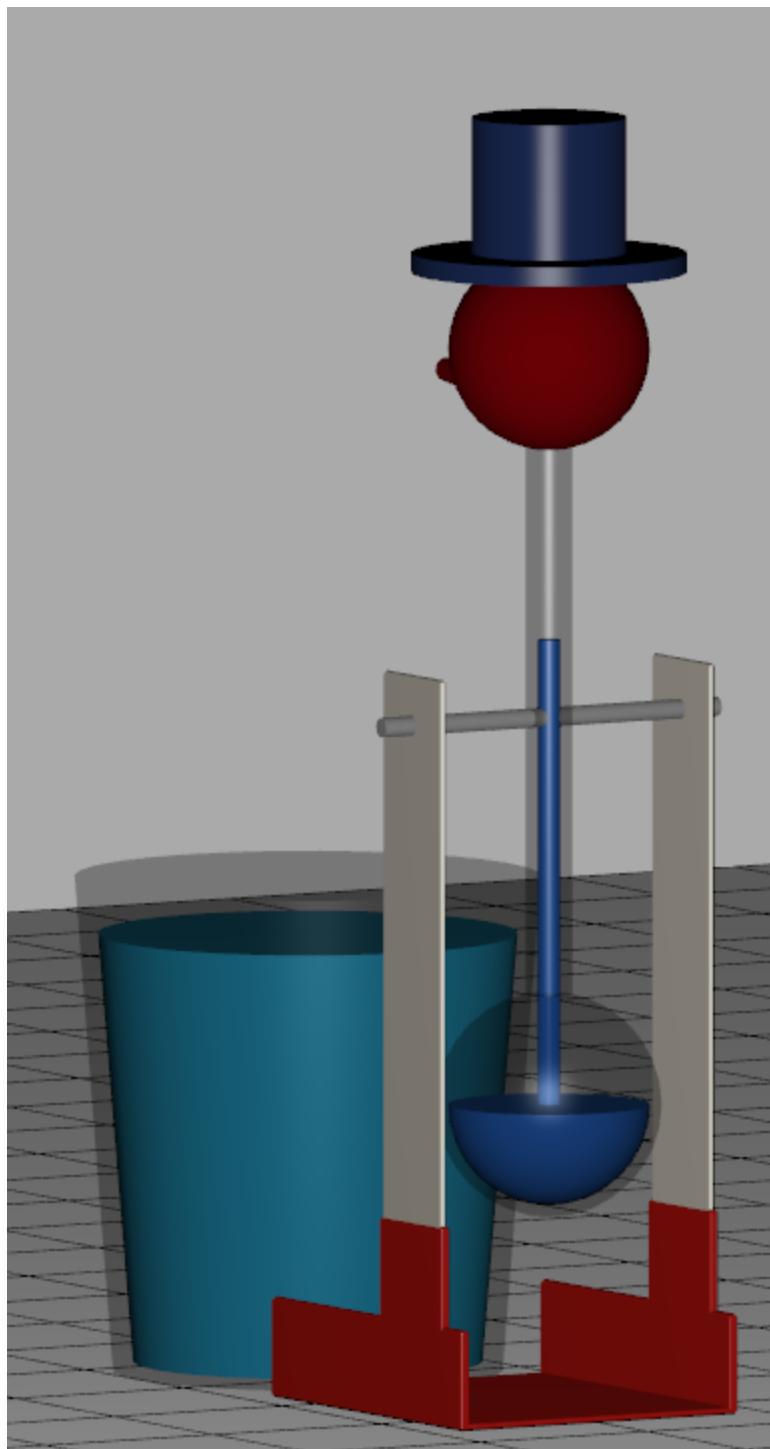
We've had word that he's been sighted at these coordinates. Your job is to properly set the target to see what the bird's doing. You don't need to modify the camera's position or size itself.

Answer

```
// CONTROLS
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(-2800, 360, -1600);

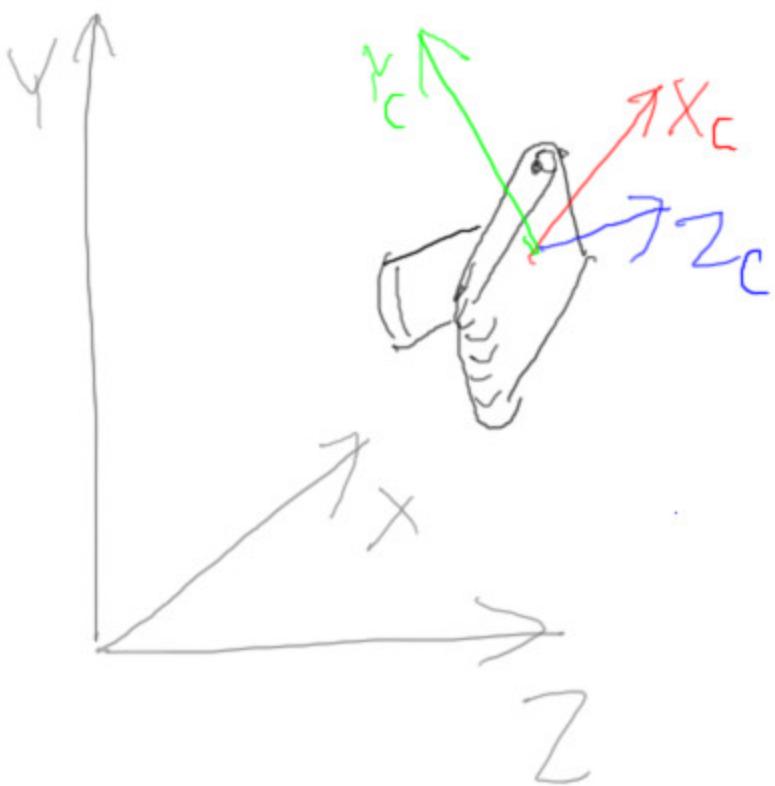
// CONTROLS
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(-2800, 360, -1600);
```

The solution is to set the camera control's target position to the proper coordinates. Once this is done, the bird's position is revealed. His quest for water is at an end!



Lesson: View Transform

[show camera in space. We want “space in camera” - show re-oriented rotated view]





We've just shown how to position and orient a camera in world space. If we apply this matrix to some camera object, the camera then gets placed in the world. However, we want exactly the opposite of this. We don't want to position the camera with respect to the world, we want to position the world with respect to the camera.

Think about what's happening: we have some objects in a scene. They've perhaps had a number of modeling transforms applied to them. The objects are now all positioned in world space. What we want to do now is transform the world to the camera's frame of reference.

[get out camera and teapot model. Move camera with respect to teapot, then teapot with respect to camera. Draw on screen 2D coordinate axes, X and Y. Camera at bottom pointing up, teapot up and to right. Start camera at origin.]

Here's a camera [flash flash] and a model. Say we slide the camera 5 units along the world's +X

axis. From the camera's point of view, if we subtracted 5 from all of the world's coordinates everything would be nicely oriented for the camera. The long and short here is that we use the inverse of the matrix that would position the camera in the world. Inverting basically says, "no, it's all about me, the camera: reorient everything with respect to me me me." This inverse matrix is called the view matrix.

[$C^T M O$]
[$V M O$]

For this reason this view matrix is sometimes written as the inverse of the camera matrix, the matrix that would move the camera to its place in world space.

Another way to think of this view transform is simply as the very last modeling matrix in the chain. The M matrix here represents all the transforms needed to get an object into world space. Now one last transform, the view matrix V, takes the objects in world space and moves them into the camera's frame of reference.

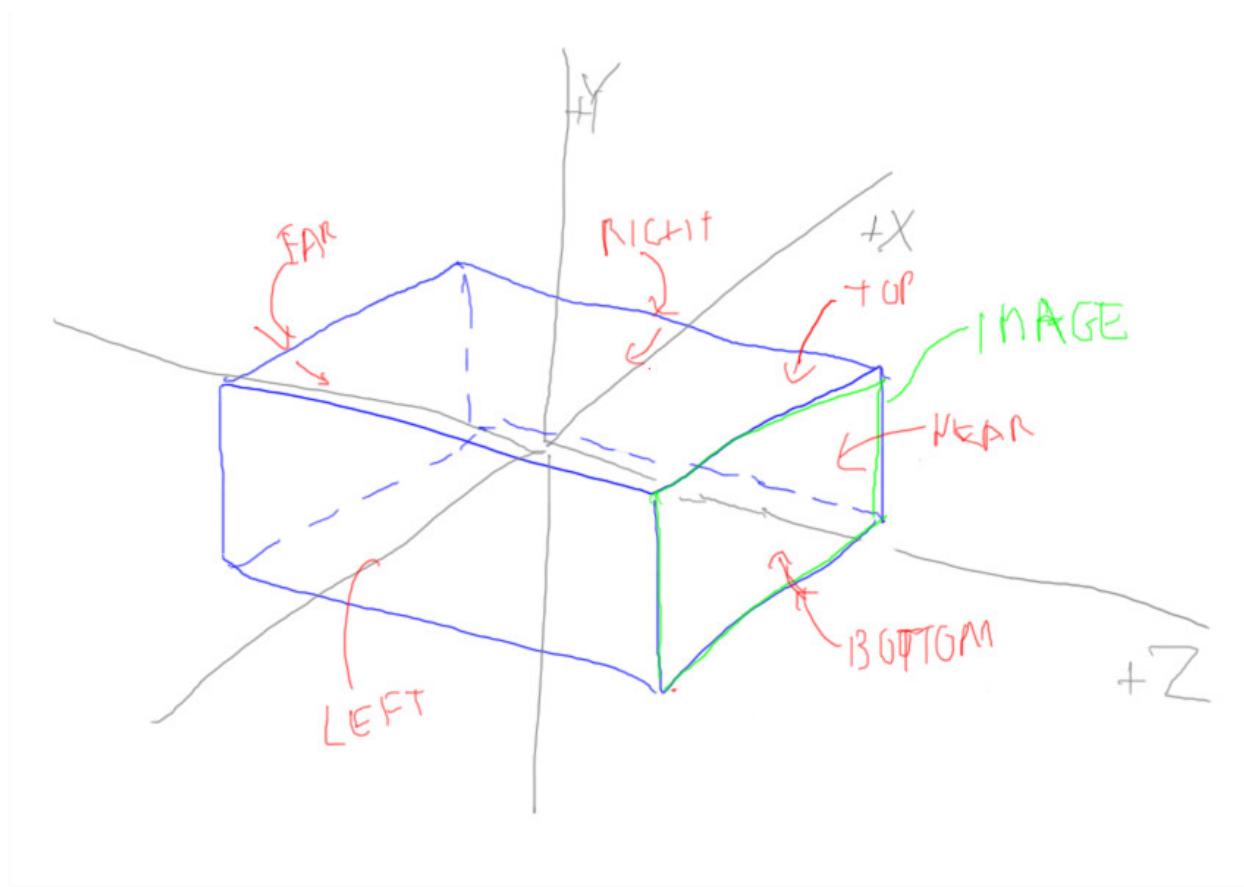
If you deal with WebGL or OpenGL directly, you will see these two matrices put together as a single matrix, called the **model-view matrix**. This matrix is applied to an object, immediately taking it from its own model space to view space.

Lesson: Normalized Device Coordinates

```
OrthographicCamera( left, right, top, bottom, near, far );
```

```
OrthographicCamera( left, right, top, bottom, near, far );
```

[Put a camera at the origin, looking down -Z. Copy it from a layer. Move the whole box to the left - it is shown incorrectly here, it should be with the near face centered at the origin.]



We now have all our coordinates in the camera's frame of reference. What we now want to do is define a box in this space to contain everything we're going to render. It's sort of like defining the size of the stage.

As noted before, the orthographic camera's definition is to give a left, right, top, bottom, near, far set of values to define this box. These six numbers define the corners of a rectangular box in the view space of your camera. The first time I explained it, we made the box and then oriented it in space. What's happening as far as the matrices go is the reverse: we first orient the world to the camera's view of things, then define our box with respect to the camera.

----- new page -----

left = -right
bottom = -top
near = 0
[this gets added later: far = a positive value]

In practical terms, we usually make the left and right be opposites, along with the top and bottom. The near value is often 0, though can be any number. The idea is that the view matrix moved everything in the world to be placed relative to the lens of our camera. The box defined by the

projection matrix is then in front of the camera and symmetric around its view axis.

[P (V M) O] **[projection model-view]**

This box, called the “view volume”, in fact performs a projection. We’re not just defining a box, we’re saying take anything in this box and project it onto the +Z side of the box and make an image.

You may have been wondering: why are we looking down the negative Z axis? This is done because it makes for a right-handed coordinate system for the camera: the Y direction is up, X is to the right, giving a Cartesian Coordinate system for the image formed. To keep things right-handed, the Z axis must be pointing towards the viewer, meaning we have to look down the -Z axis.

[add line about positive far value]

That said, three.js thinks about the orthographic camera as going in a positive direction, so the far value is specified as a distance along the -Z axis.

Our chain of matrices adds this projection to the front, in other words the projection is applied next. When applied, a coordinate will get a new value in this projection’s frame of reference. This frame uses what are called **normalized device coordinates**, or **NDC** for short. These NDC coordinates are in the range **-1 to 1 for X, Y, and Z**. We’ll talk more about this transform in a minute, but the point is that whatever is inside the box will have coordinates transformed to this new range.

[end recording of 3/27 - I recorded the next few, but didn’t turn on the overhead camera!]

Question: Orthographic Matrix Type

[start part 8, recorded 3/28]

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here’s the orthographic projection in all its glory. It really doesn’t look like much. It turns out the orthographic projection matrix is pretty simple and uses nothing you don’t already know. That’s

the question to you.

Does this matrix perform:

- scaling?***
- rotation?***
- translation?***
- projection?***

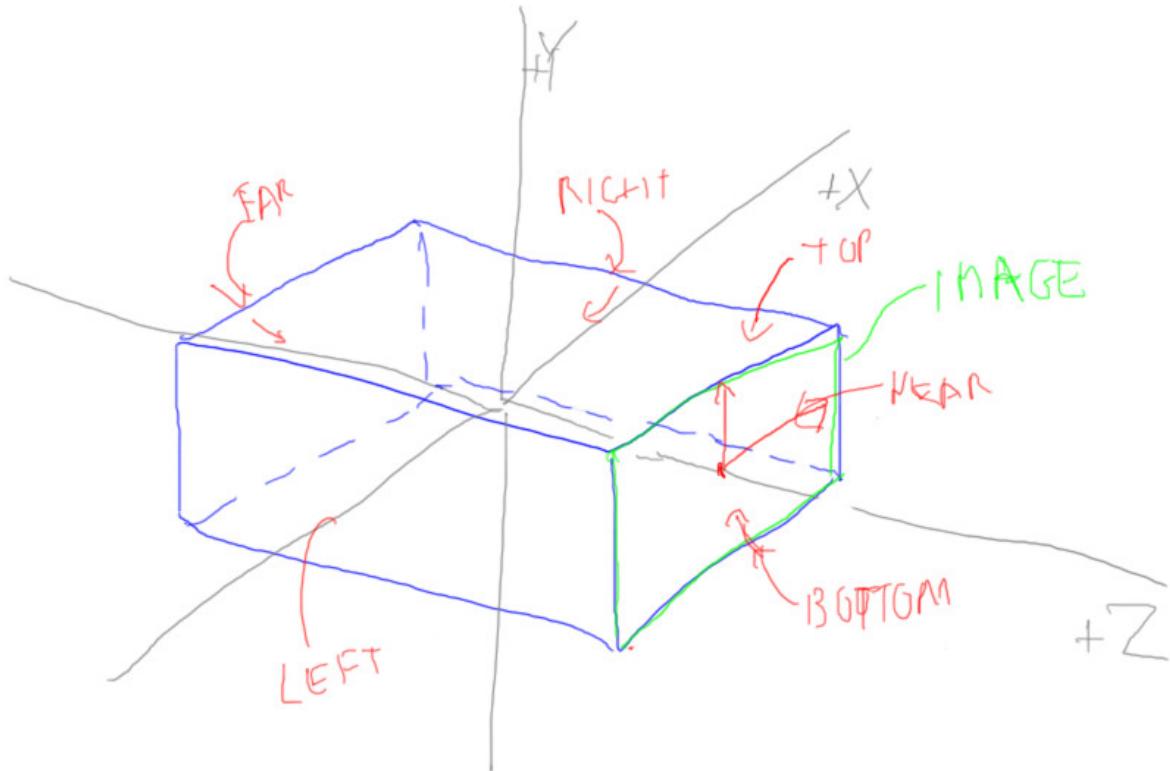
Check all that apply.

My hint to you is to see the matrix zones lesson. By the way, I want non-trivial answers here. A pure rotation matrix has a translation of 0,0,0. That doesn't count.

Answer

The answer is that scale and translation transforms are being done. There are two ways to solve this problem. One is to look at the matrix itself and see that there is some sort of scaling and translation happening. The other is to realize what the transform is doing: it is defining a rectangular box in space by making it a different size along each axis and by moving its origin elsewhere.

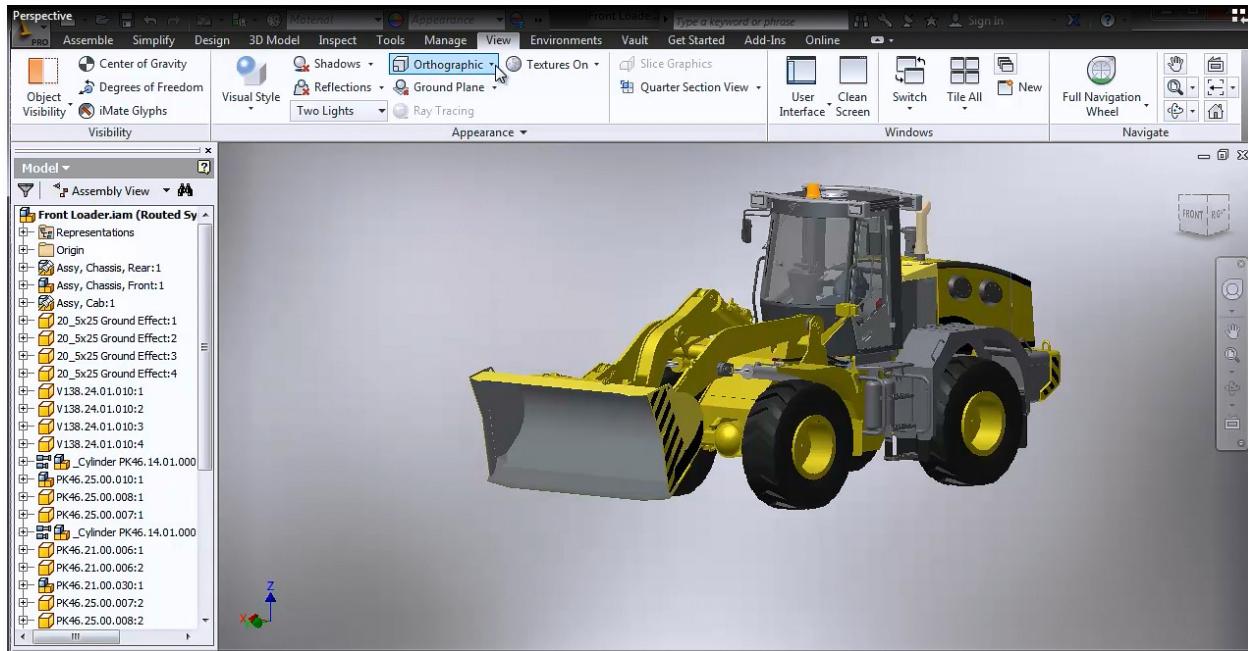
[put image of our orthographic box back on the screen to explain this next part, drawing on image how it's centered.]



Notice that if left is equal to the negative of the right, which is the common case, the X translation value will be zero. Similarly, if the bottom limit is equal to the negative of the top, the Y translation value is also zero. This means the box is centered along the -Z view axis and so doesn't need to be translated in these directions.

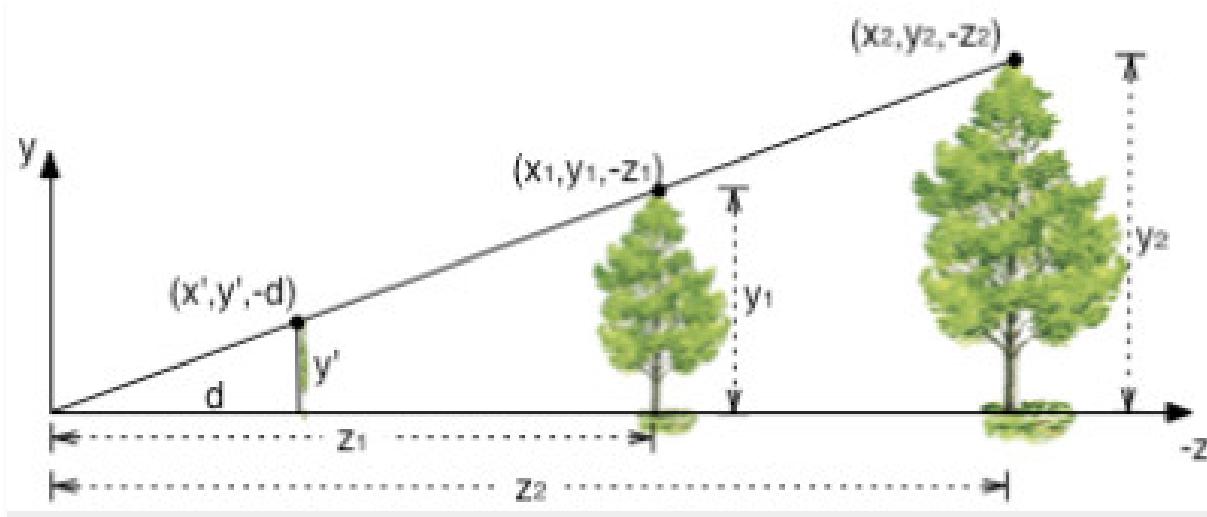
Lesson: Perspective Camera

[video clip: Unit6_Perspective, Perspective.mp4, just play it over my words. Shows transition from orthographic to perspective.]



The perspective camera is where things get interesting. This camera is more like real life, with objects in the distance being smaller. That said, it uses the same sort of pipeline as before. Internally you define a view matrix exactly the same as before. However, then the projection matrix is formed differently.

[draw similar triangles view of stuff:]



In a perspective view of the world, objects get smaller in the image as they get further away from the viewer. Another way to say it is that if something is farther away, it needs to be larger in world space if it wants to appear the same size on the screen. This gives us the idea of similar triangles: y_2 divided by z_2 will be equal to y_1 divided by z_1 .

So perspective projection is going to involve division; but when we multiply a vector by a matrix it's just some dot products, a series of multiplies and additions.

[**(X, Y, Z, 1)**]

This is where the fourth coordinate for our points comes into play. Up to now, this fourth value has always been equal to 1. We run a point through any set of modeling transforms and the fourth coordinate value of 1 is left untouched. This all changes with perspective projection.

[Additional Course Materials:

One clever form of illusion is based on what's called forced perspective. See [this video](<https://www.youtube.com/watch?v=tBNHPk-Lnkk>), for example. People have played tricks with perspective for hundreds of years, such as [this drawing](http://commons.wikimedia.org/wiki/File:William_Hogarth_-_Absurd_perspectives.png) from the 18th century.

]

Lesson: Three.js Perspective Camera

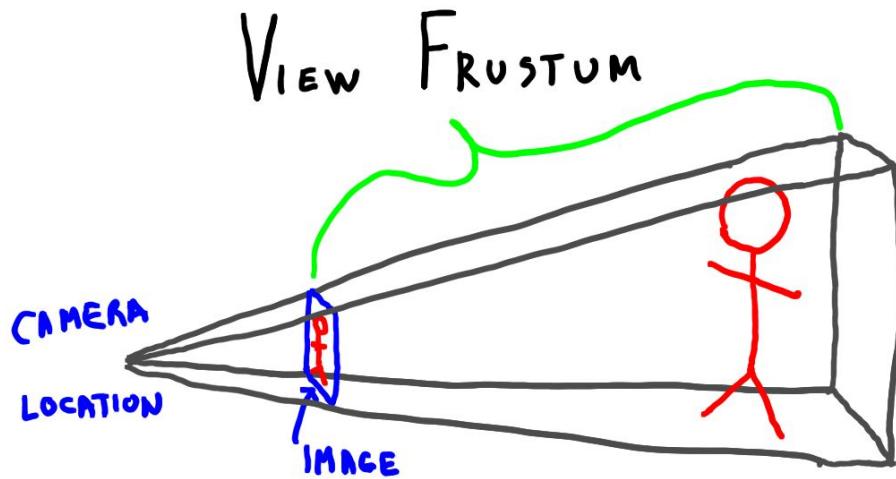
Before explaining how projection works, let's talk about how the perspective camera is set in three.js.

```
// PerspectiveCamera( angle, aspectRatio, near, far )
camera = new THREE.PerspectiveCamera( 30, aspectRatio, 1, 10000 );
camera.position.set( -170, 170, 40 );
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(0,50,0);

// PerspectiveCamera( angle, aspectRatio, near, far )
camera = new THREE.PerspectiveCamera( 30, aspectRatio, 1, 10000 );
camera.position.set( -170, 170, 40 );
cameraControls = new THREE.OrbitAndPanControls(camera, renderer.domElement);
cameraControls.target.set(0,50,0);
```

It's similar to the Orthographic Camera. In fact, the creation call has fewer parameters. We know what the last three are: the aspect ratio of width divided by height, followed by the near and far planes of the view volume we want to define.

[view frustum: add near and far and central axis and FOV]



Remember the view frustum? It's back, at last. The near and far distances are measured from the tip of the pyramid, where the camera is placed, down a central axis. Back when I was a young man, we called "near" and "far" by the names "hither" and "yon", which I think is more poetic. You'll still occasionally see the terms "hither and yon" used in products, so keep an eye out.

The first argument for the perspective camera is the field of view. This is the angle between the top and bottom planes of the view pyramid. Notice that in three.js this number is specified in degrees, unlike many other angles, which use radians.

The field of view along with the aspect ratio fully describe the locations of the four sides of the pyramid. For the orthographic camera we defined the location of every side of our view volume box. Here the view is assumed to be symmetric around the center axis. In other words, instead of both a top and bottom value, we need just a single angle describing both; same thing with right and left, the aspect ratio along with the angle is enough. This is three.js's way of doing things, and 99% of the time it's what you want. That said, it's entirely possible to specify each side of the pyramid, and WebGL itself has a frustum call that does just this.

----- new page -----

If you change values on the camera itself, such as the field of view, near, or far planes, in three.js

you need to call:

```
camera.updateProjectionMatrix();  
camera.updateProjectionMatrix();
```

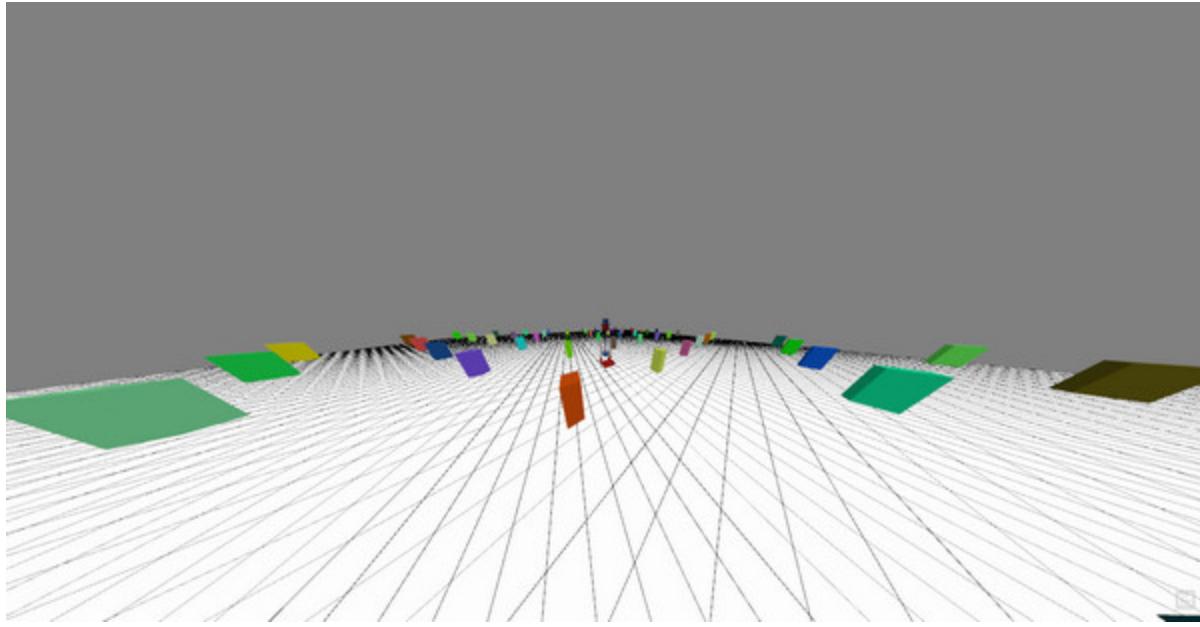
in order to have these changes take effect. For most demos these camera parameters are usually set once on creation and rarely touched, so three.js doesn't spend any time checking them each frame. If you change these values while running your program, calling updateProjectionMatrix has three.js evaluate these parameters and form a new projection matrix.

Exercise: FOV Slider

```
[ use dat.gui library  
“field of view”  
range: 1 to 179 degrees, start at 40 degrees  
]
```

Add a slider that controls the field of view parameter for the perspective camera. Use the dat.gui library to set up this slider. Give the slider the name “field of view”. This field of view slider should have a range of 1 to 179 and start at 40 degrees.

[unit7-fov_solution.js]



When you're done, the program should look like this. As you change the field of view, you should notice things looking considerably different. Once you have the code working, I recommend you try moving around the scene. Compare changing the field of view with what happens when you use the mousewheel or middle mouse button to move back and forth.

[Additional Course Information:

Check previous demo code or the [dat.gui documentation](<http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage>) for how to create a slider.

]

Answer

```
effectController = {  
    fov: 40  
};  
var gui = new dat.GUI();  
var element = gui.add( effectController, "fov", 1.0, 179.0 );  
element.name("field of view");
```

```

effectController = {
  fov: 40
};
var gui = new dat.GUI();
var element = gui.add( effectController, "fov", 1.0, 179.0 );
element.name("field of view");

```

There are two parts. We need to write some piece of code like this to set up the slider itself. This code is run at initialization.

```

function render() {
  var delta = clock.getDelta();
  cameraControls.update(delta);

  camera.fov = effectController.fov;
  camera.updateProjectionMatrix();
  renderer.render(scene, camera);
}

function render() {
  var delta = clock.getDelta();
  cameraControls.update(delta);

  camera.fov = effectController.fov;
  camera.updateProjectionMatrix();
  renderer.render(scene, camera);
}

```

In the render method we use the effectController's fov variable to set the camera's field of view, which is also in degrees. Once the camera parameters are set, we call updateProjectionMatrix to make the camera changes take effect.

We'll talk much more about field of view and exactly what it means, though I think you have a pretty good idea after doing this exercise.

Lesson: Perspective Matrix

Here's the perspective matrix formed from the three.js parameters:

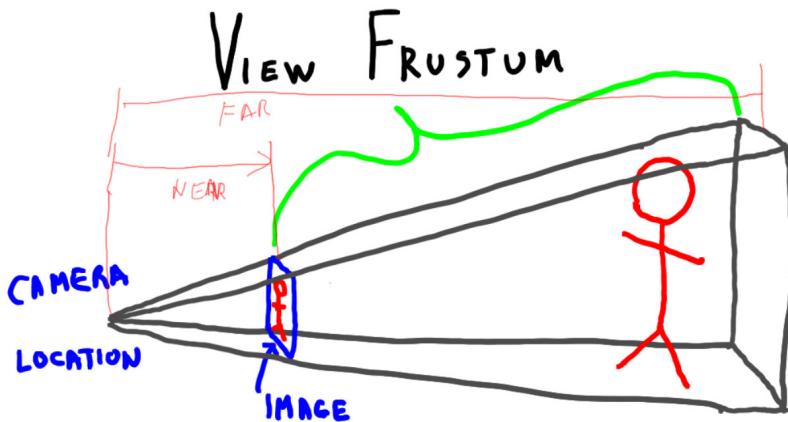
[redraw, of course. In WebGL these numbers get fed in as negative values along the axis, super-confusing.]

$$\begin{bmatrix} \frac{1}{\text{aspect} \cdot \tan(\text{FOV} / 2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\text{FOV} / 2)} & 0 & 0 \\ 0 & 0 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} & \frac{2 \cdot \text{near} \cdot \text{far}}{\text{near} - \text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The numbers in the upper left 3x3 are scale factors, similar to the orthographic projection, though the Z scale is a little different. There's also a translation factor for Z. However, the big differences here are that there's a -1 in the last row, and the lower right corner now has a 0.

I'm not going to derive this projection formula here; most good 3D graphics texts run through this process. Also beware that for some formulations the near and far values are negative, since you're traveling down the negative Z axis. The one headache of keeping things in a right-handed system is this whole +Z points at the viewer problem.

[draw a little frustum and note near and far distances are positive.]



I'm using positive near and far values in this formula, because that's how three.js specifies things, and thank heavens for that. In fact, these two value *must* be positive numbers for the

perspective transform - you can see things would get weird if the near value, for example, was somewhere behind the camera.

For orthographic projection we can use whatever numbers we like for near and far, even negative values, since we're really just selecting a box in space. For perspective we're doing something more elaborate with transforming space.

---- new page -----

The interesting part is how this matrix works with coordinates. Let's take a nice simple case:

field of view 90 degrees

aspect ratio is 1

near is 1

far is 11

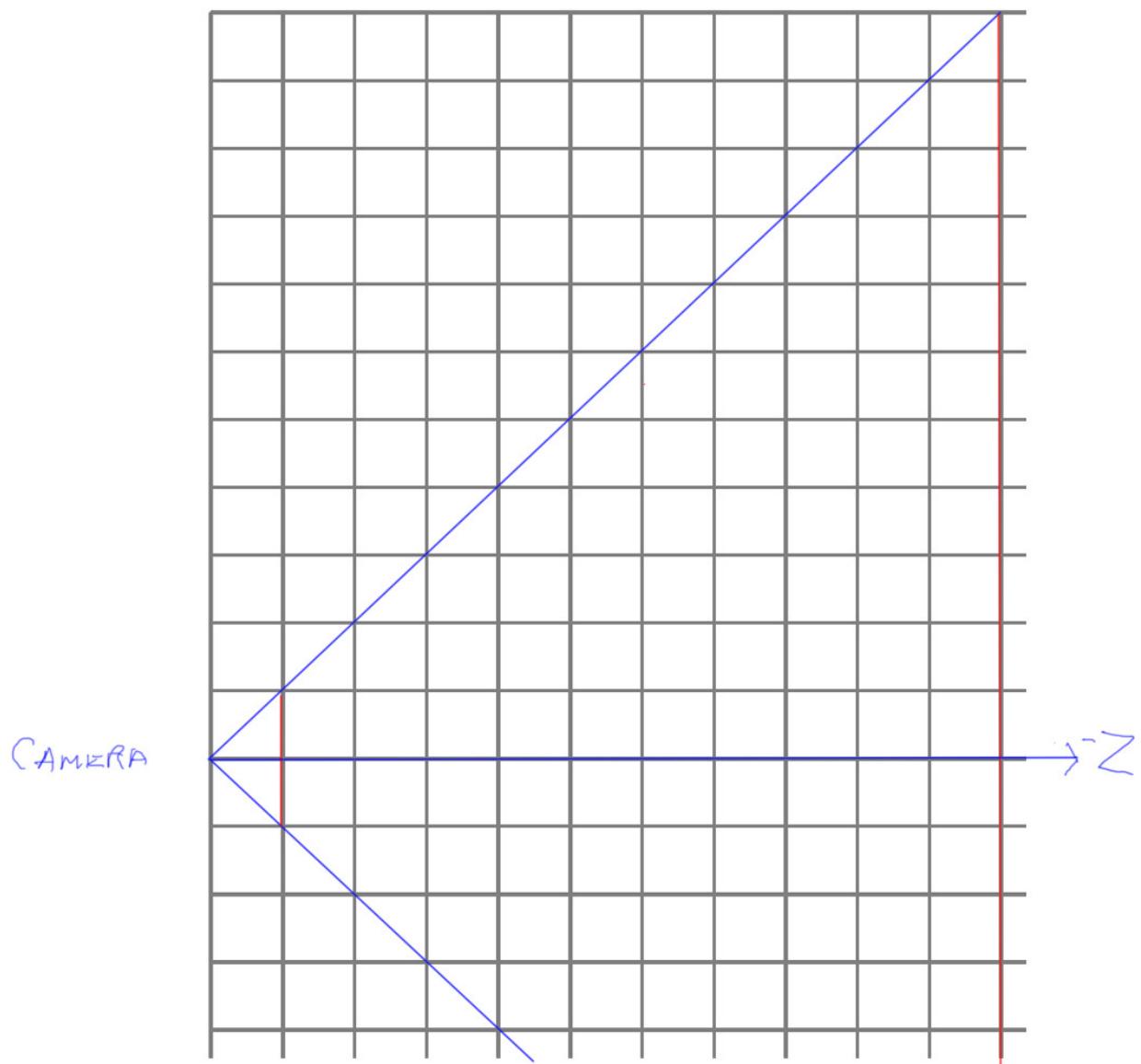
This gives us a matrix

[paper <http://mathbits.com/MathBits/StudentResources/GraphPaper/halfpageOne.pdf>]

[draw the situation, show Y up and -Z of course, MAKE SURE IT'S NEGATIVE-Z, and then add test points. Make sure to add every test point in a separate layer, and make screencaps of each point and with no points.

Note: I forgot the +Y axis here - add it. Label FRUSTUM.

]


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \\ \end{bmatrix}$$

]

Let's use this matrix on some test points and see what we get.

Exercise: Perspective Coordinate Transform

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

]

Multiply this matrix by three points and give the results

points are

(0,0,-1) -> gives (____,____,____,____)

(0,11,-11) -> gives (____,____,____,____)

(0,4,-6) -> gives (____,____,____,____)

I want all four coordinates. Don't do anything beyond multiplying the points times the matrix. Don't forget that points have a fourth coordinate equal to 1.

[Additional Course Materials:

Did you know that you can type mathematical expressions into [Google's search box](<https://www.google.com/>) and it will compute the answer for you? Very handy.

]

Answer

[Careful with the minus signs!]

(0,0,-1) -> gives (0, 0, -1, 1)

(0,11,-11) -> gives (0, 11, 11, 11)

(0,4,-6) -> gives (0, 4, 5, 6)

Here are the answers. Instead of values of 1 in the last location, we have all sorts of interesting numbers.

Lesson: Homogeneous Coordinates

[copy from above]

$(0,0,-1) \rightarrow \text{gives } (0, 0, -1, 1)$
 $(0,11,-11) \rightarrow \text{gives } (0, 11, 11, 11)$
 $(0,4,-6) \rightarrow \text{gives } (0, 4, 5, 6)$

The four values are called **X, Y, Z, and W**. These are called “homogeneous coordinates”, and they’re used for projection.

What we do next with these coordinate values is divide each value by the W of the coordinate. This is called the “**perspective divide**” or “**homogeneous divide**”.

[underline or otherwise mark the w’s]

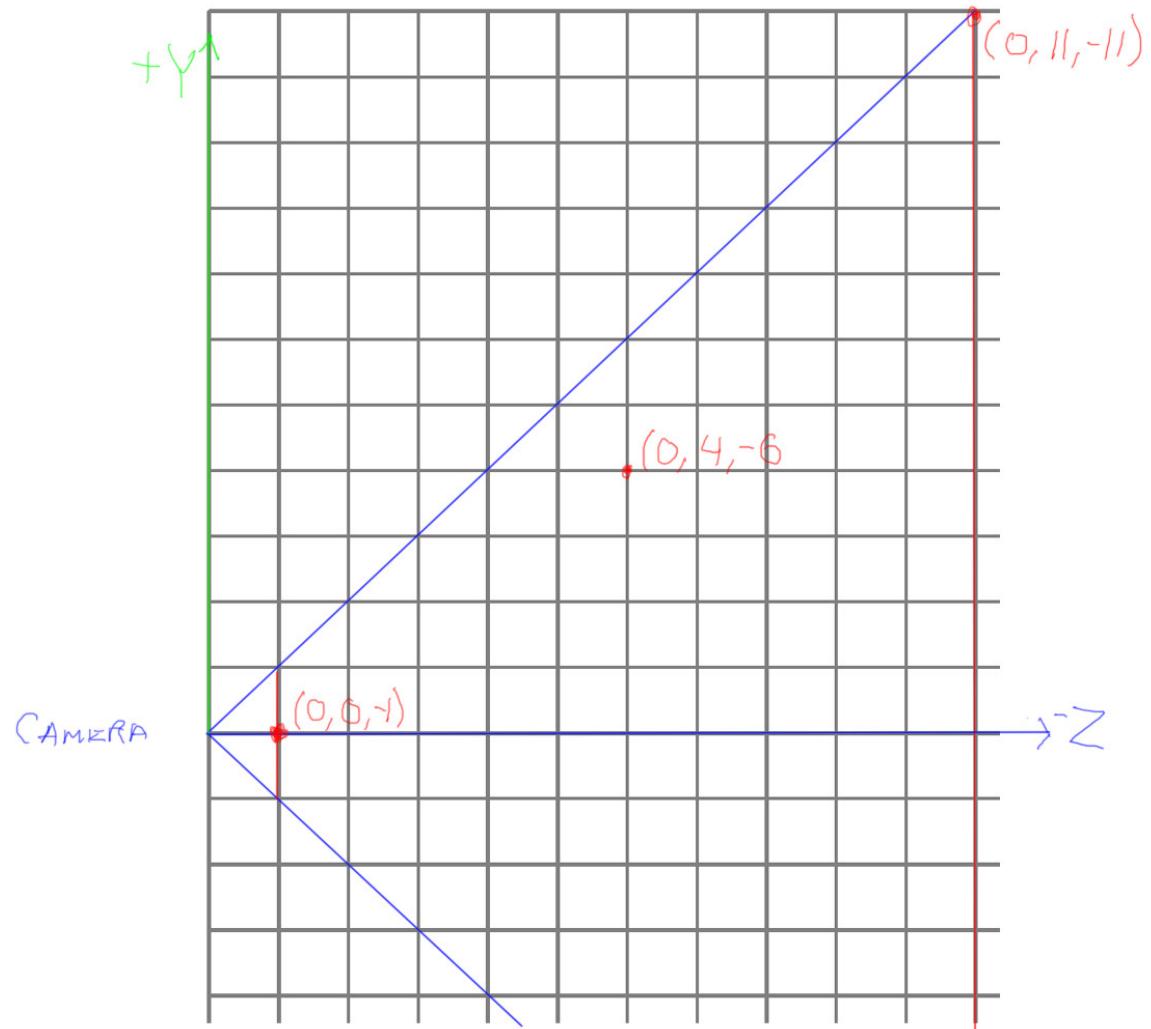
$(0,0,-1) \rightarrow \text{gives } (0, 0, -1, 1) \rightarrow (0, 0, -1)$
 $(0,11,-11) \rightarrow \text{gives } (0, 11, 11, 11) \rightarrow (0, 1, 1)$
 $(0,4,-6) \rightarrow \text{gives } (0, 4, 5, 6) \rightarrow (0, 0.67, 0.83)$

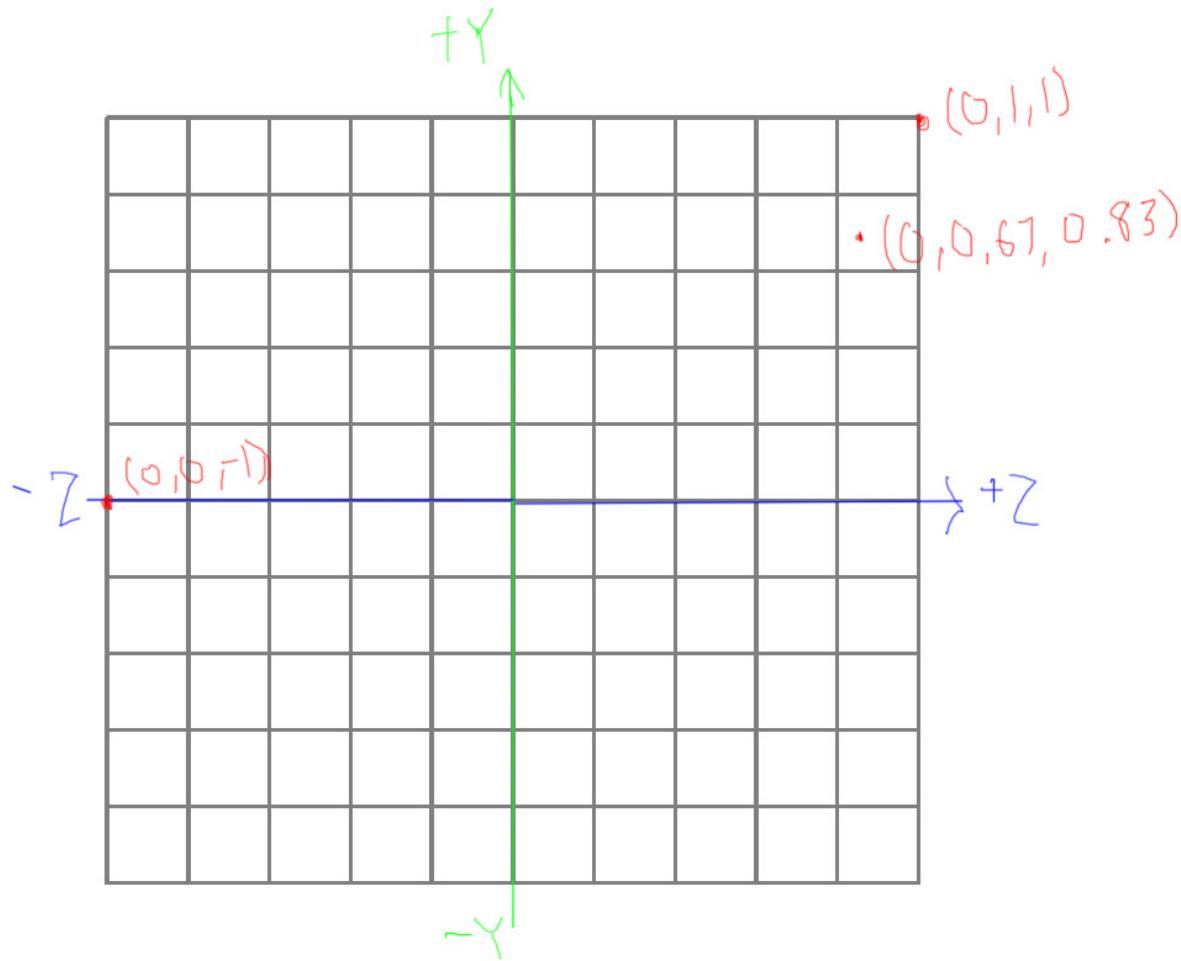
So, for our three test points, we had a value such as 0,0,-1,1, dividing by 1 is simple enough, that gives us 0,0,-1. We don’t need to bother writing out the W value in the result, since W divided by W will always equal 1.

For our next point, W is 11. Dividing all the coordinates by 11 gives 0,1,1.

Our last point is a little more interesting. Dividing through by W gives us 0, 0.67, 0.83.

[Show **frustum** and then show **NDC** points]

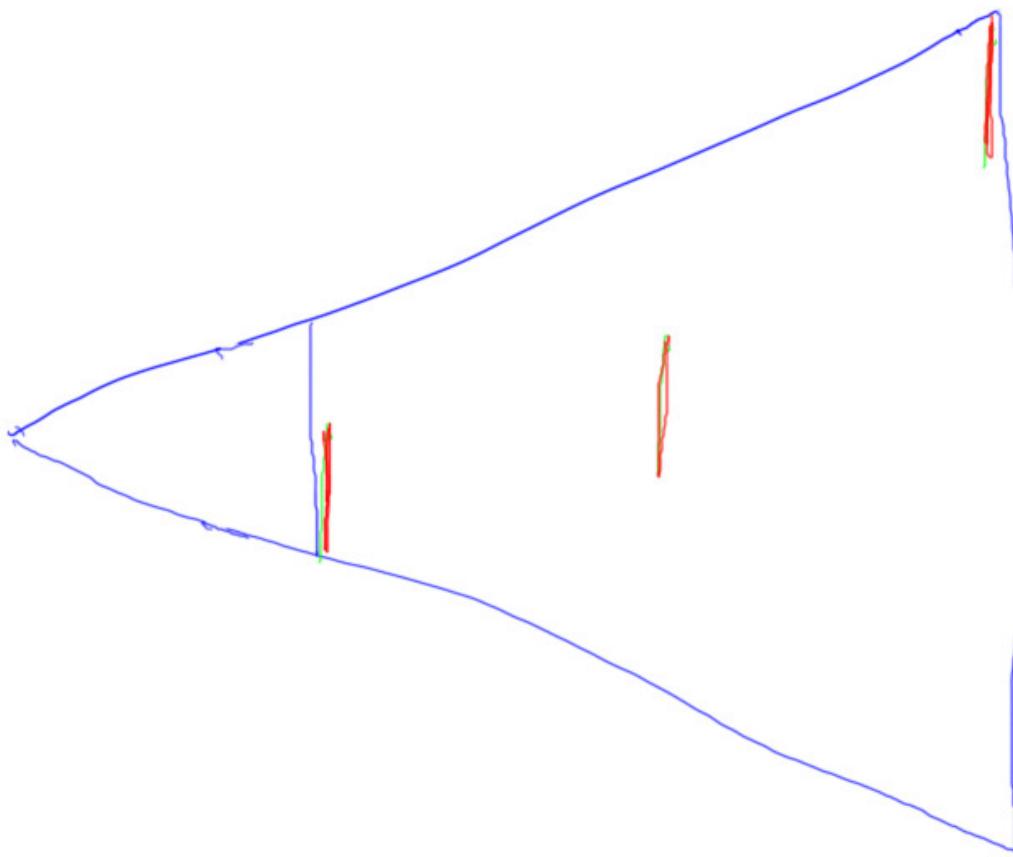




Here are plots of the original points in view space and their corresponding new locations. Notice that the -Z axis is pointing to the right for the frustum, and the resulting axis is +Z to the right.

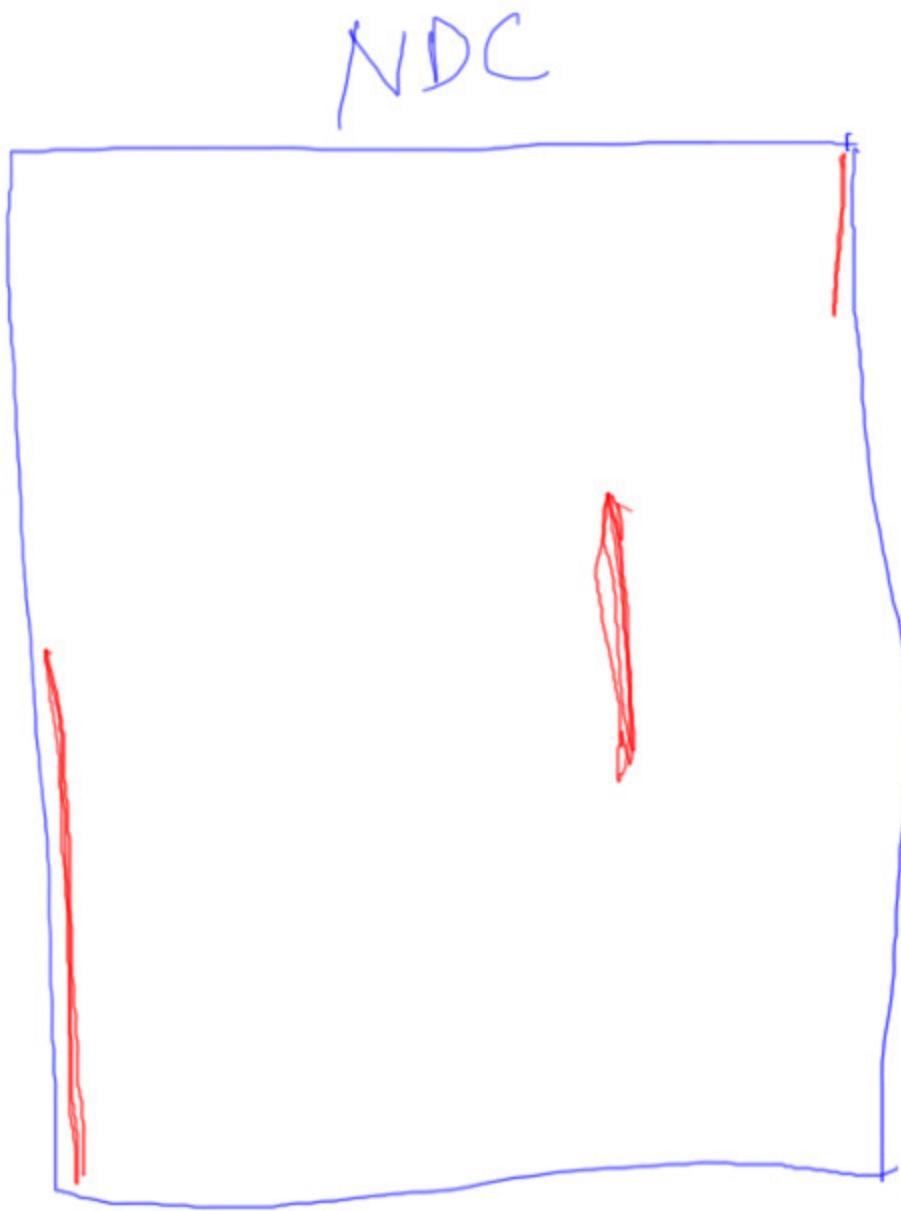
Look at what's happened with these points and where they are transformed. They started inside or on the edge of our frustum. After the projection matrix is applied and division by W is performed, the resulting points are in normalized device coordinates. Anything in the range -1 to 1 for X, Y, and Z is in the visible view volume.

[Show frustum with 3 objects, no scale on frustum, at near plane $\frac{1}{3}$, $\frac{2}{3}$, $\frac{3}{3}$, all with same height. Show NDC version, which will have depths of -1, 0.5, 1 on a scale of -1 to 1. Make heights of objects proportional: stretch closer one, etc. Also, make frustum left and right distance equal to top and bottom, so NDC box overlaps easily. ALSO: SHOW DISTANCES of objects, 1, 2, 3]



Let's take another example, to show what happens to 3 objects that are the same size in world space, but at different distances. This second object in the scene is at twice the distance of the first, the third is three times as far.

[FADE IN, overlap with frustum]



When we transform to Normalized Device Coordinates, the relative area of coverage of the near plane stays the same. That is, the close object was half as high as the screen in our frustum view, and transforms to half the height in NDC space. The second object is farther away, and shows smaller. The third object, on the back of the frustum, is much smaller than the others in normalized device coordinates.

You might have noticed an interesting thing has happened to the depth of the second object. We'll talk more about that in a bit, as it's important.

[Additional Course Materials:

For more on homogeneous coordinates see

[Wikipedia](http://en.wikipedia.org/wiki/Homogeneous_coordinates).

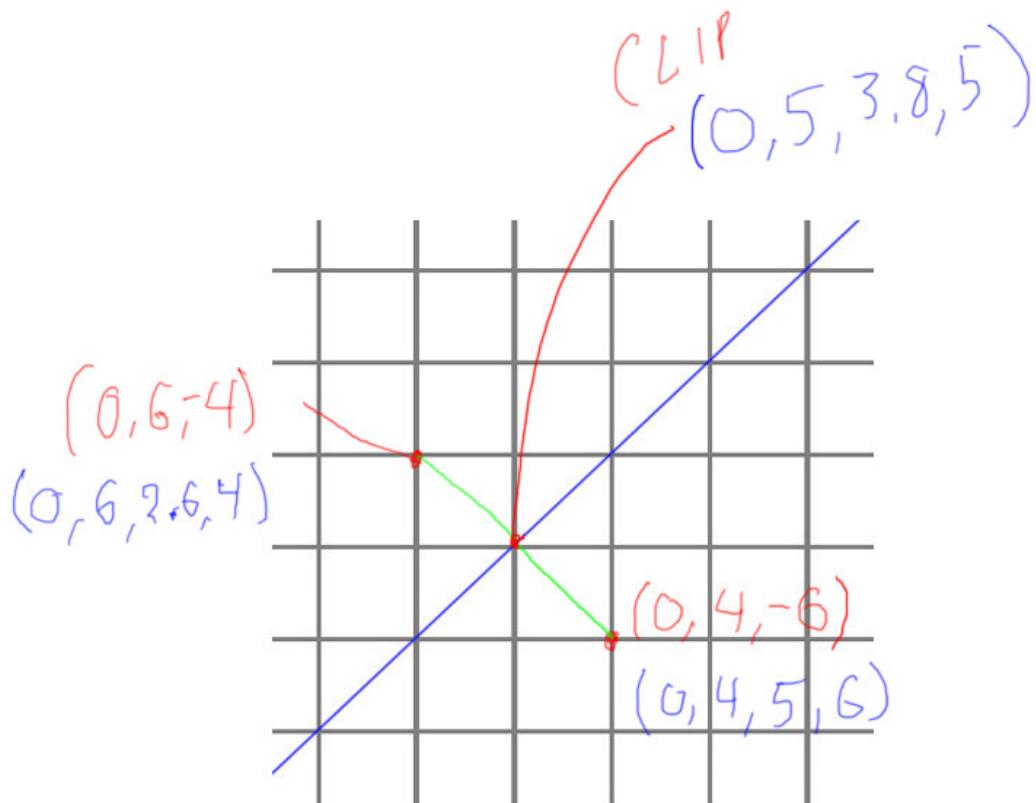
]

[end of recording part 8, 3/28]

Lesson: Clipping

[show our test frustum yet again, but zoom in on this section]

[part 9 recorded 3/28]



I left out a step that happens after projection and before division by W: clipping and culling.

Say we have two points

$(0, 4, -6) \rightarrow \text{gives } (0, 4, 5, 6)$

and

$$(0, 6, -4) \rightarrow (0, 6, 2.6, 4)$$

and they form a line segment. These two points and this line segment are shown on this zoomed-in part of our frustum.

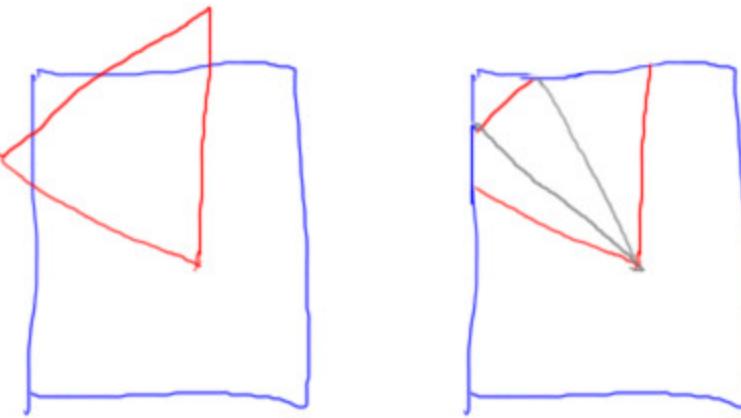
This second point is outside of the frustum. We want to have all coordinates inside our final view volume in NDC space so that we can render them. Clipping is done for line segments and triangle edges that poke outside of the frustum. An edge can be clipped by any number of the faces of the frustum.

What happens here is that all the coordinates between the two points get linearly interpolated. For our example, the point on the frustum face is halfway between our two points. The interpolated point is then:

$$(0, 5, 3.8, 5)$$

We then divide this point by W, as usual, to get a point in Normalized Device Coordinates.

[Draw a triangle and show it chopped by two edges of the screen.]



When a triangle is clipped by one or more faces of the frustum, it can form a polygon with more than 3 sides. This polygon is triangulated by the GPU and each separate triangle rasterized. You as a user don't really have to know or care much about this clipping process - it happens automatically. It's worth knowing about mostly if you need to do similar testing operations on the CPU-side.

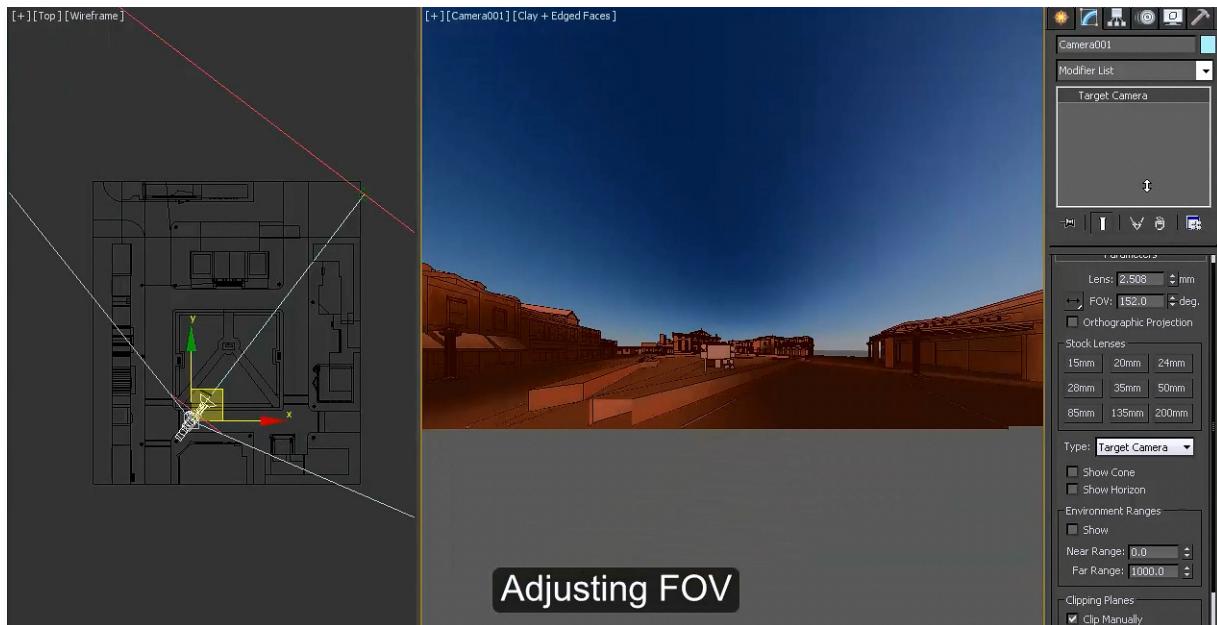
In computer graphics we often make a big deal about how we store 3D points and vectors as

homogeneous coordinates, with the fourth element W. In reality, for almost every operation the W value is either 0, meaning a vector, or 1, meaning a point. It's only after projection and during clipping that the W value is anything but 1. Once clipping is done and we're using normalized device coordinates, we're done with homogeneous coordinates.

However, these homogeneous coordinates are important in that they're what the vertex shader produces. When the coordinate is transformed by the projection matrix, but before division is performed, the coordinates are called “**clip coordinates**”. The vertex shader can produce other intermediate results, such as computing the location once the model and view matrices are applied. It is required that the vertex shader produces a position on the screen for the vertex. This position is a homogeneous coordinate; the rasterizer then takes this position and performs clipping.

Lesson: Field of View

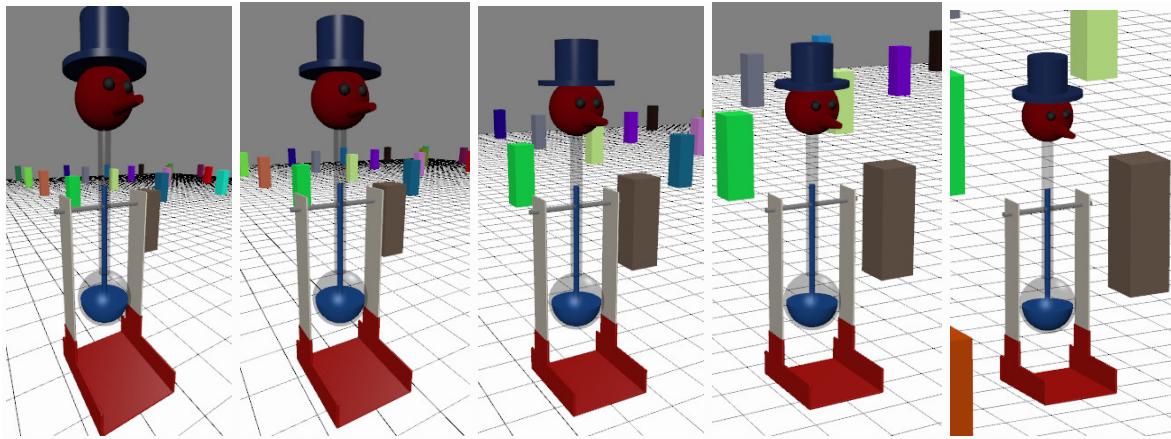
[video Unit6_FieldOfView, “5.2 Camera Control 3.mp4” - starts at 0:25 go to 0:35 - short clip, please either make it fit or just repeat it for the time I talk at first]



Let's talk a bit about perspective camera control. The parameter with the most obvious effect is the field of view. As mentioned before, this value is the angle from the bottom of the view frustum to the top. It also adjusts the side to side angle, factoring in the aspect ratio.

---- [new page] -----

[Show various fields of view]



[zoom lens is optional.]



The field of view parameter acts something like a zoom lens. As you set this angle smaller and smaller, whatever is in the middle of the screen grows larger and larger.

For these images what I've done is have the field of view match the camera move, more or less. In other words, as I zoom in on the model, I also move farther away, so that the model takes about the same amount of room on the screen. As we zoom out, the perspective distortion gets less and less.

At the limit, that is, as we attempt to go infinitely far away and have the field of view approach an angle of zero degrees, the view becomes an orthographic projection. If you're practically infinitely far away, everything you're looking at is at essentially the same distance as far as relative size goes, which is how the orthographic projection treats the scene.

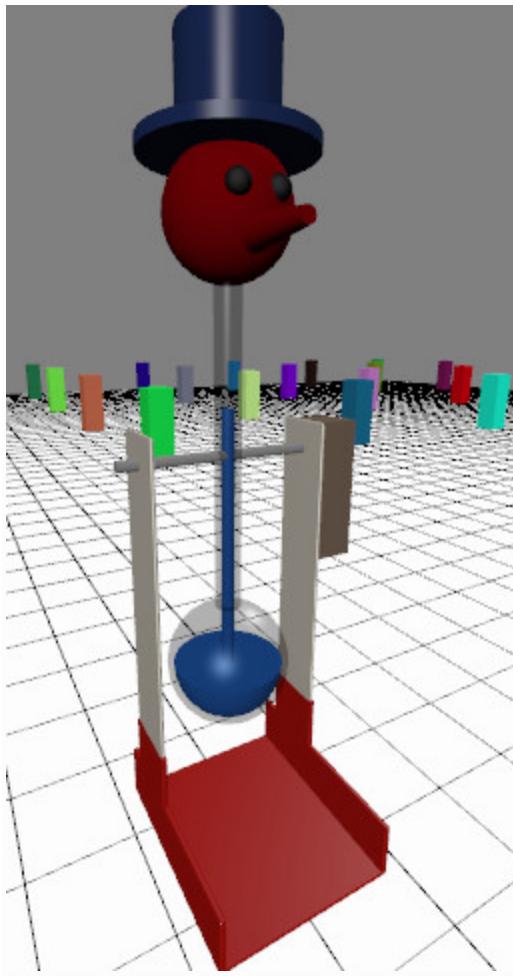
This effect of having the field of view and the camera movement offset each other has become a go-to effect in cinematography. If you want to express that something weird or dreamlike is happening in the scene, you zoom in while moving backwards at the same time, or vice versa. Give this demo a try yourself and see how it looks.

Demo: Dolly and Zoom

[Dolly out and zoom in to keep target stable width: unit6-dolly_zoom_demo.js]

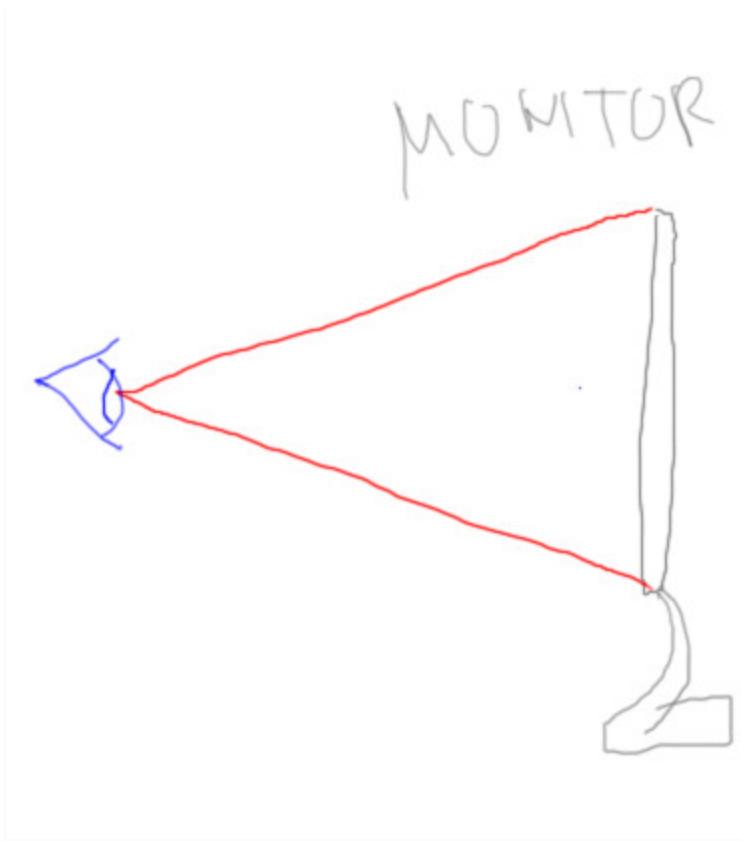
[Additional Course Materials:
Here's [more about the dolly-zoom
technique](<http://tv tropes.org/pmwiki.php/Main/VertigoEffect>) and its use in films.
]

Lesson: True Field of View

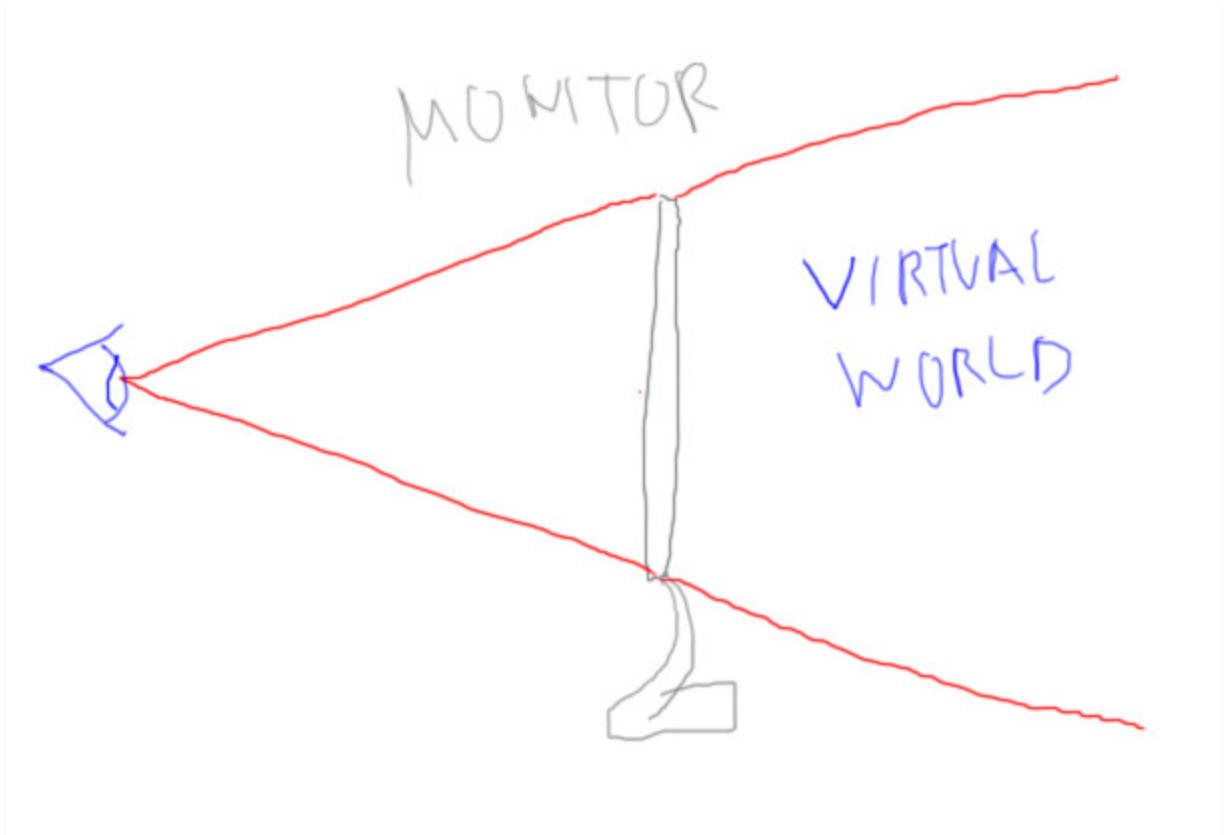


[draw field of view for screen to right of these images, and show what is happening: real window, virtual window]

We'd say the view of the bird is pretty distorted. Really, the problem is that you're not sitting close enough to the screen. It's sort of like being overweight. I could instead say that I'm not dense enough - if I was denser then I'd look thinner.

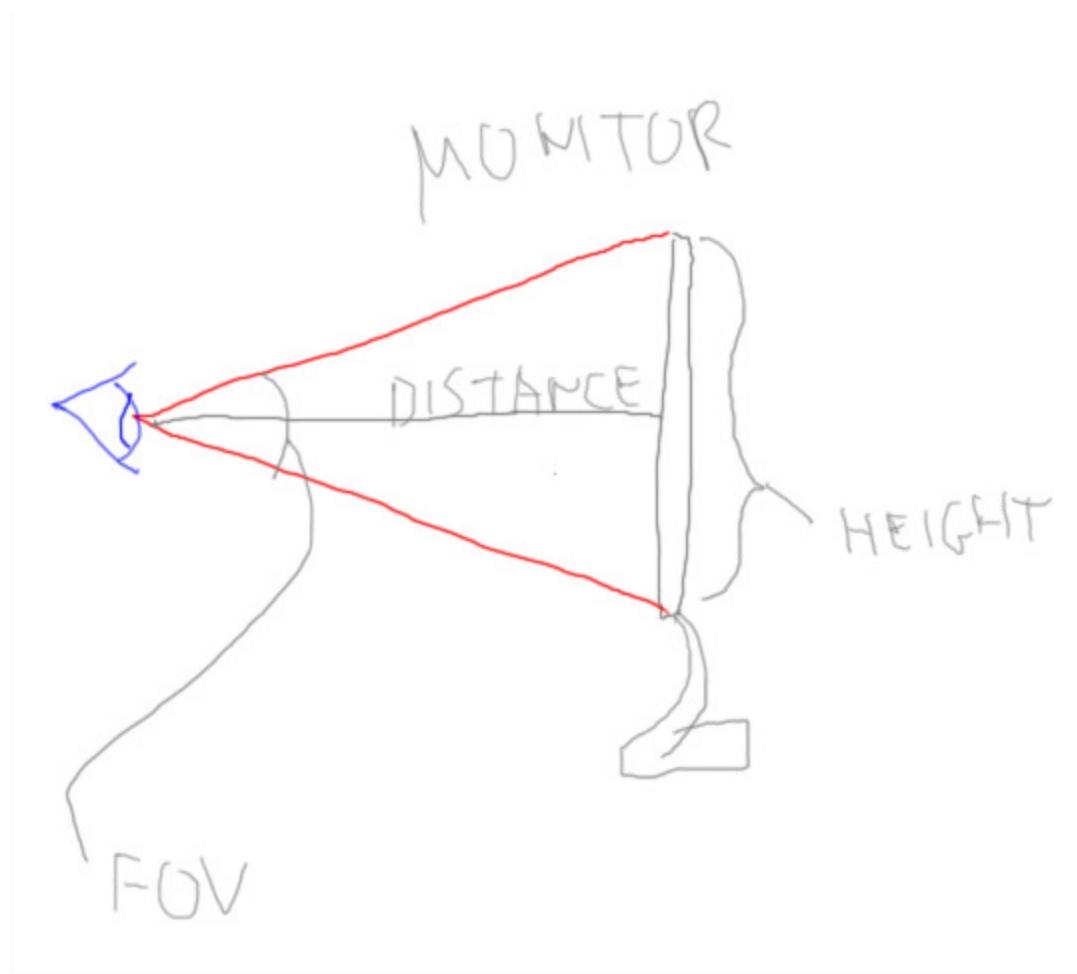


Think of what the field of view represents. When you look at your monitor, tablet, or mobile device, you're a given distance away from it. The screen itself is a certain height. This forms a real-world field of view angle.



A window on your screen is your window into a virtual world. Say this window in the real world has a 20 degree field of view for your real-world eye position. If your virtual world also has this same field of view, you won't perceive any distortion. Objects towards the edge of the screen might look distorted if you were to back away from the screen, but at your current eye position the field of view is perfect: the image is warped a bit in projection, but your eye is seeing the screen at a tilt that compensates for it.

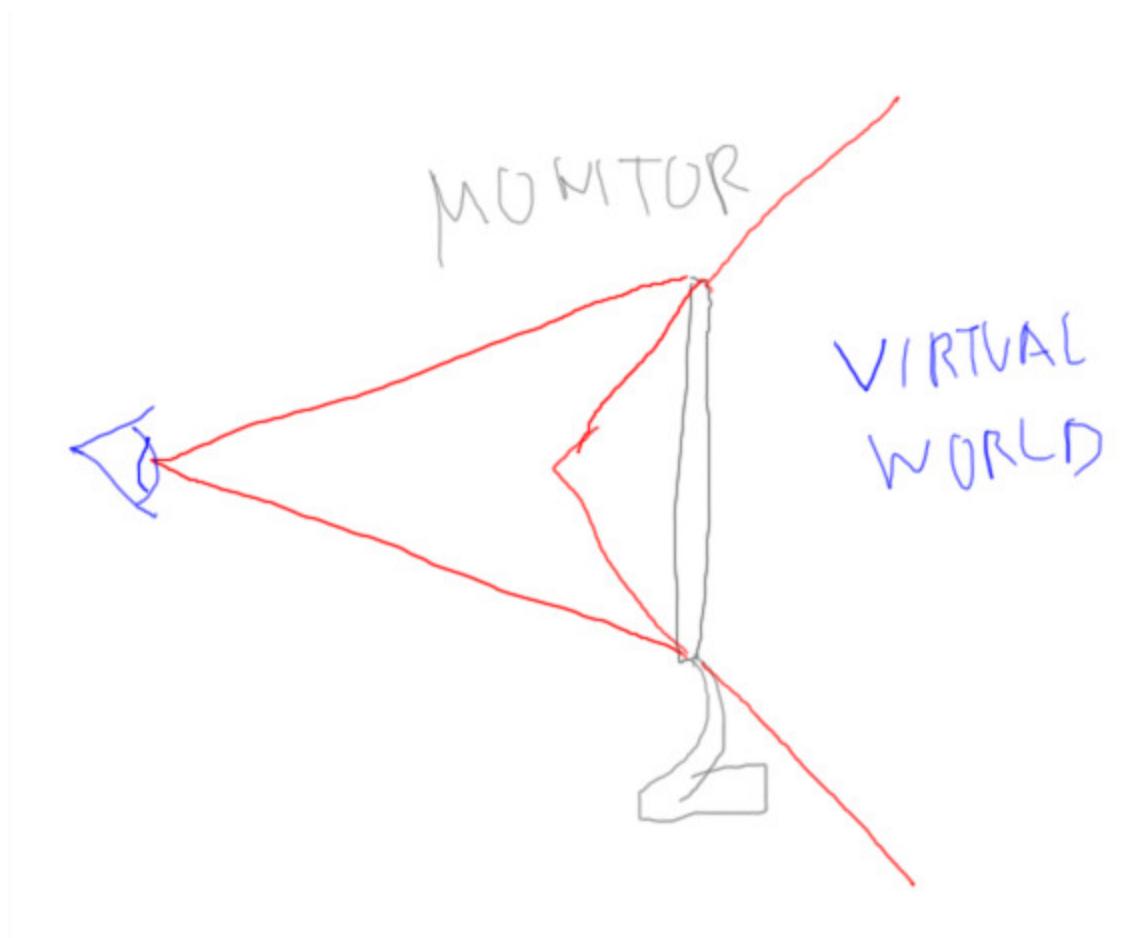
[show trig formula: tangent of (FOV/2) is the monitor height, show on screen]



The formula for window height for a given field of view and distance from the screen is

$$\text{height} = 2 * \tan(\text{fov}/2) * \text{distance}$$

As the field of view increases, the tangent value increases and so the height of the monitor increases.



What tends to happen in videogames is that the designer crank up the virtual world's field of view so that players can see more of what's happening around them. This can certainly lead to a more playable game, but this is also where the distortion comes from, that the virtual field of view doesn't match the real-world field of view. If you moved your point of view so that you were at the apex the virtual world's frustum (gesture) the virtual world would appear undistorted. Of course, it might actually all be blurry at this distance or you might get a massive headache, but that's the theory.

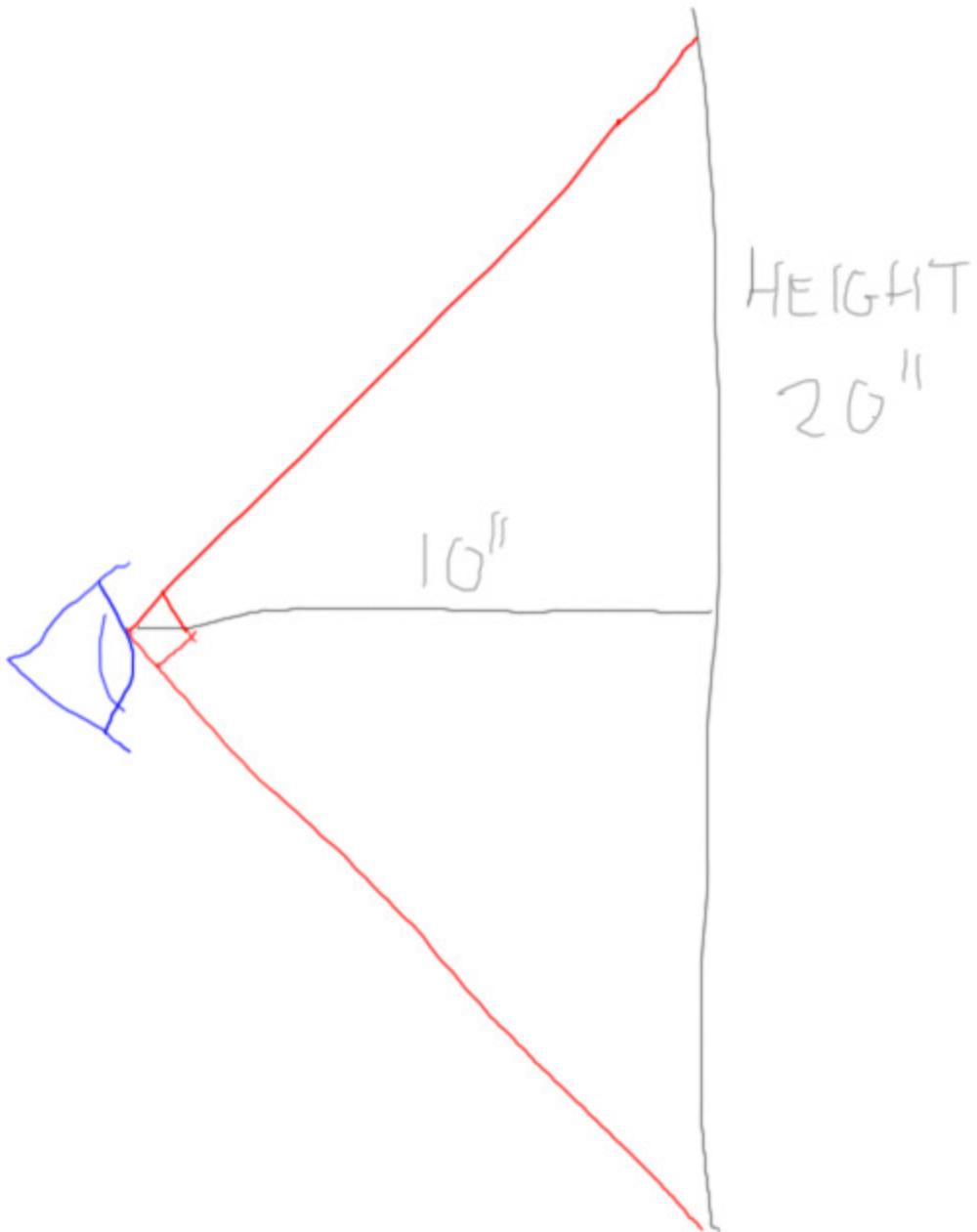
Question: Monitor Field of View

[**90 degrees FOV, 20 inches high**]

Say you want to have a graphics program displaying in the monitor truly represent a 90 degree field of view, and your monitor is 20 inches high. How far away from the screen should your eyes be?

Answer

[Draw monitor configuration, show triangles]

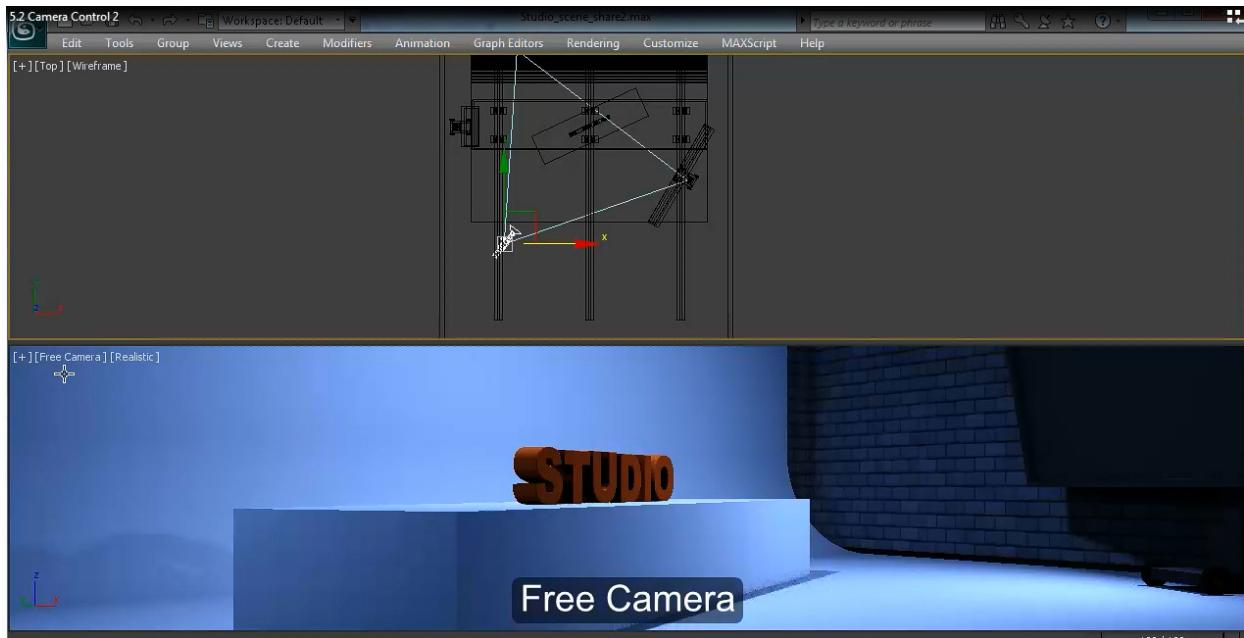


We don't really need any trigonometry for this one. Here's the eye and the monitor's height. At 90 degrees, these are right triangles and each pair of two sides are the same lengths. We can see

that the height is twice that of the distance the eye must be from the monitor. The answer is 10 inches, which is darn close to the eye.

Lesson: Target

[video Unit6_FieldOfView, “5.2 Camera Control 3.mp4” - starts at 0:00 go to 0:23]



There are many ways to move a camera through a scene. We can walk or fly, we can look one direction while moving another, and so on. Here we're moving a bit sideways and a bit forward while looking forward.

I usually think of the camera being in one of two modes: viewer-centric or model-centric. When things are viewer-centric, the viewer is moving around through the world.

[transition at this point to the “target” view, where we’re focused on the desk.]

When model-centric, a particular object in a world is being studied and the viewer’s goal is to see this object from different angles.

In this mode we keep the camera focused on a location as we move. The place where we look is called the “target” in three.js and in other systems. Note that this target location is not at all necessary in making the various matrices needed for viewing the scene. The target is much more related to user intent.

[---- new page ---]

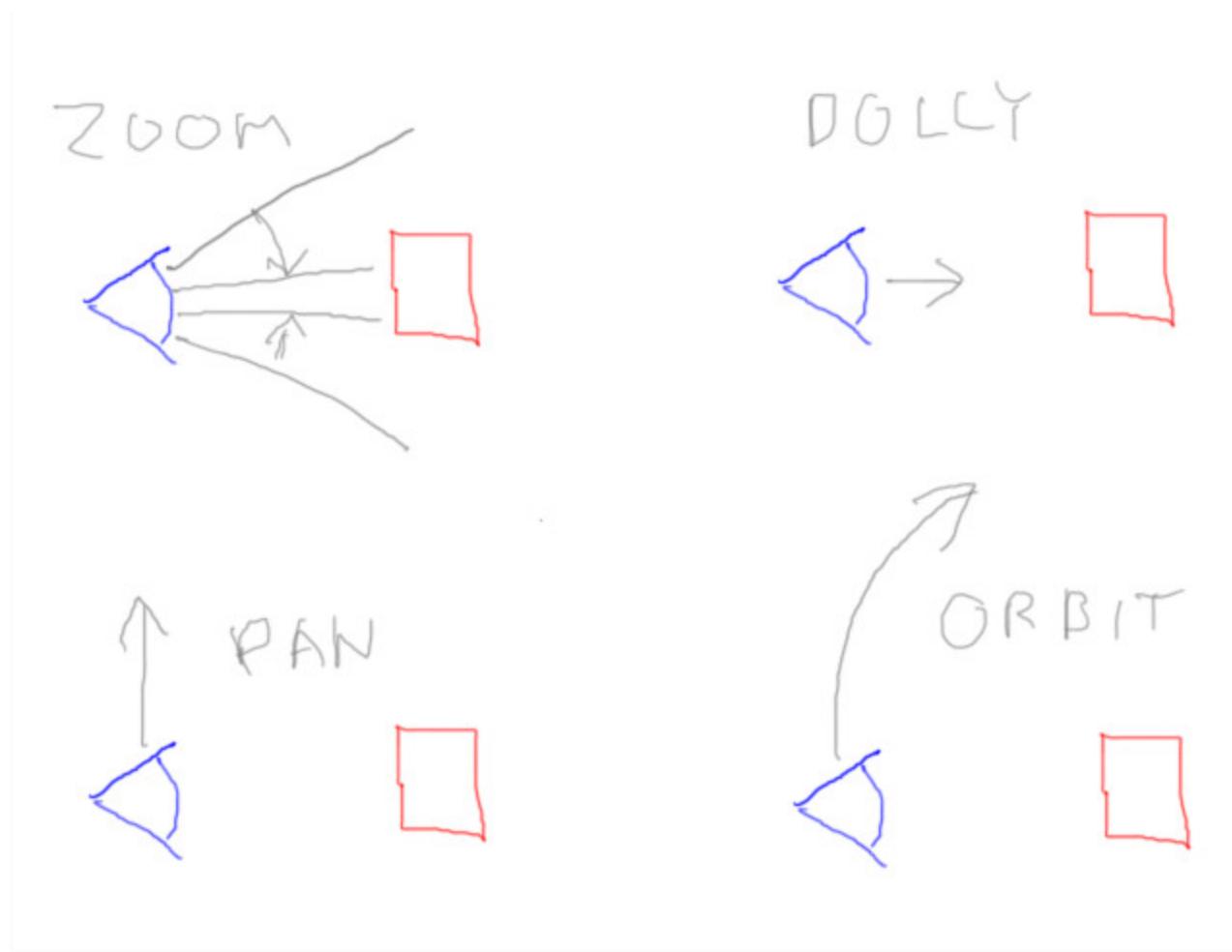
```
cameraControls.target.set(0,300,0);  
cameraControls.target.set(0,300,0);
```

Because the target is such a useful concept, many (but not all) camera control systems support the idea of a target in some way.

To set the target we use the cameraControls' target parameter. We've set this parameter before: it provides the point that our camera is looking at, so setting the view matrix. By setting the target in the cameraControls, you not only set camera's view transform, you keep the camera pointed at this location when you orbit. If the object is moving, you can update the target each frame and the camera will stay trained on it.

Lesson: Dolly, Pan, Orbit

[show camera moves: zoom, dolly, pan, orbit]



Let's define some terms for various camera moves. Here's a top-down drawing of a viewer and an object.

When you adjust the field of view, that's zoom (or bizarre quasi-fish-eye, if you widen the field of view too much). When you move directly towards or away from the object you're looking at, that's dollying. IT'S NOT ZOOMING. "I have a zoom on my camera" does not mean "If I want to make something larger, I run towards it and yell 'zoom!'" It's so easy in computer graphics to either zoom in or dolly in that they're sometimes not that easy to tell apart. However, they're certainly different in their effect on the camera's field of view and location. The one giveaway that you're dollying and not zooming is if new faces become visible or disappear. Zooming changes the size of the image seen, but does not change visibility.

All that said, I'll sometimes say "zoom in" when the controls are actually dolly controls, just because most people don't know what "dolly in" means. The camera controls for most of the programs in this course have the mousewheel and middle mouse button set to dolly the view in and out.

The other common camera moves are panning and orbiting. Panning means moving left or right as you keep the camera pointed forward. For most programs we're using, the right mouse button pans. Orbiting means circling around the **target** location, similar to how a satellite orbits the earth. For those of you into first person shooters, it's circle strafing a target. The target location is often placed at the center of whatever object is to be examined. In most of our programs, left-mouse orbits around a target location.

Incidentally, all of these mouse moves are what we just happen to use, there are no ironclad standards among programs of which mouse button does what.

[Additional Course Materials:

[Panning]([http://en.wikipedia.org/wiki/Panning_\(camera\)](http://en.wikipedia.org/wiki/Panning_(camera))) means something a bit different for cinematography.

Circle strafing is explained [here]([http://en.wikipedia.org/wiki/Strafing_\(gaming\)#Circle_sStrafing](http://en.wikipedia.org/wiki/Strafing_(gaming)#Circle_sStrafing)). Wikipedia has just about everything.

]

Question: Camera Changes

Say you're controlling a camera by using a position and target system.

For each of these camera moves, what must change, the target or the position? Check all (or none) that apply. Select an attribute only if it is changing a fair bit, say more than a foot, for a given attribute. For example, if I said to move your camera so that it was pointing straight up, the target would have changed considerably, while the position would have changed just a small bit.

Target Position

- Orbit to the right*
- Turn camera to the left*
- Zoom in*
- Pan to the right*

Answer

Target Position

- Orbit to the right*
- Turn camera to the left*
- Zoom in*

[X] [X] Pan to the right

Orbiting keeps the target fixed and moves the position.

Turning the camera keeps the position fixed and moves the target.

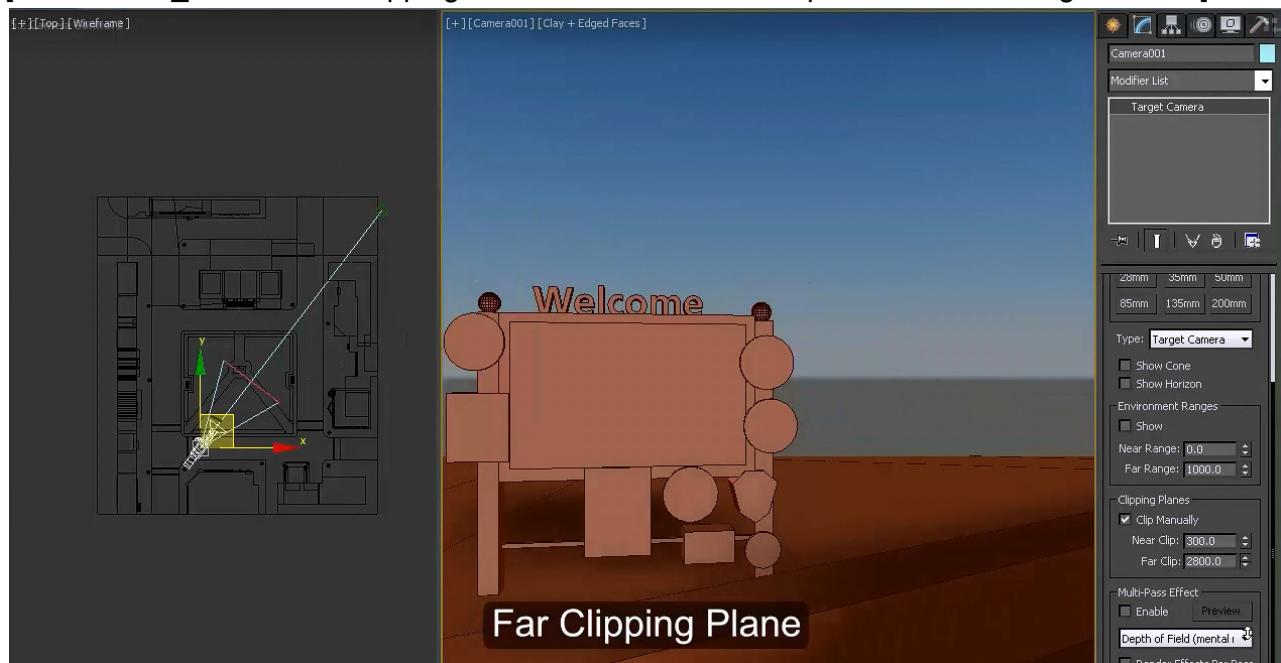
Zoom changes the field of view and nothing else.

When you pan, you are moving the camera perpendicular to the way it is facing, so both the target and the position are changing. In fact, controlling where the target location is can be a somewhat frustrating part of using various graphics modelers. If you pan and then orbit, you're now orbiting around some different target location, which may not have been what was intended.

Note that I didn't ask about the effect of dollying on the target and position. Dolly certainly changes the position, but it's up to the program whether to change the target in this case. If the control also moves the target, it's like you're walking forward and don't particularly care where the target is located. If the target doesn't change, it's like you're walking towards an object and perhaps slow down as you approach it.

Lesson: Near and Far Clipping

[video Unit6_NearAndFarClipping, “5.2 Camera Control 3.mp4” - starts at 0:39 go to 0:53]



Here you can see the effect of moving the near and far clipping planes through the scene. Seeing this kind of clipping is usually a bug, not a feature. An algorithm such as ray tracing doesn't have this sort of problem, as the mechanism there is to shoot rays from the eye.

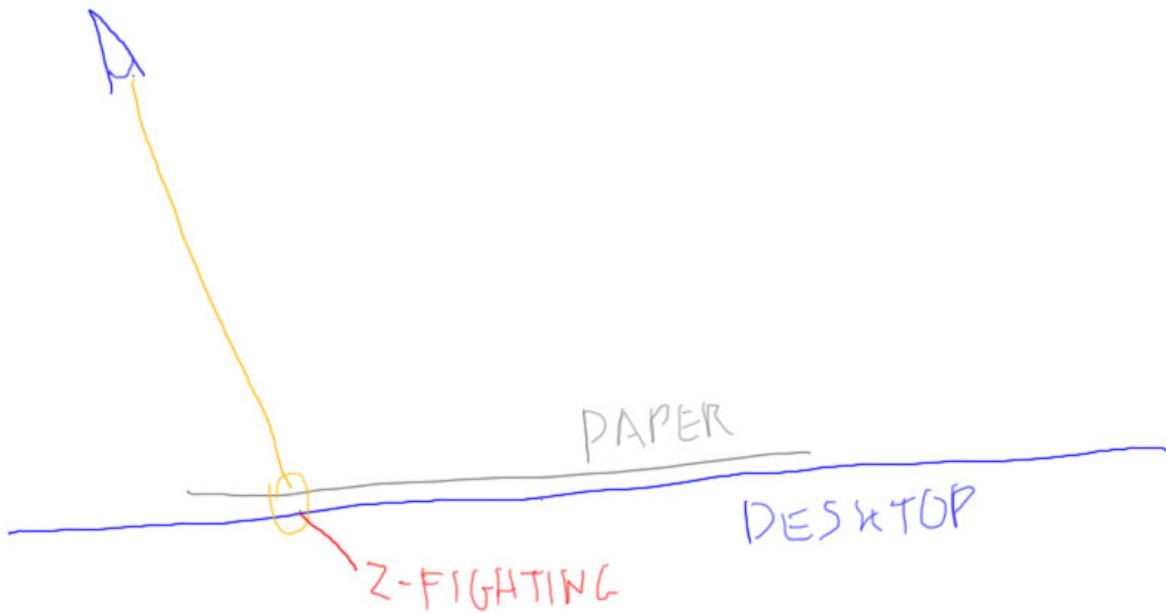
These two values are necessary for rasterization to work sensibly, due to its use of the projection matrix. Well, at the minimum you need the "near" distance; it's possible to form a projection matrix that gives a frustum with no far limit to the pyramid.

[z-buffer bytes{
[24 bits z-buffer | 8 bits stencil buffer]

draw a sheet of paper and a desktop now on a separate layer, lay out page
]

The key thing about these two values is that you want to set them to be as close together as you can, while still avoiding clipping - the near plane is particular important to move as far as possible away from the camera. The near and far values determine how the z-buffer value is computed.

Internally the z-buffer value typically gets stored as an integer value with some number of bits. For example, 24 bits is common, with 8 bits for what's called the stencil buffer, a separate buffer that I'm not going to talk about in this course but that can be used for clipping and other effects. The z-buffer has a lot of bits, but not an infinite number of them.

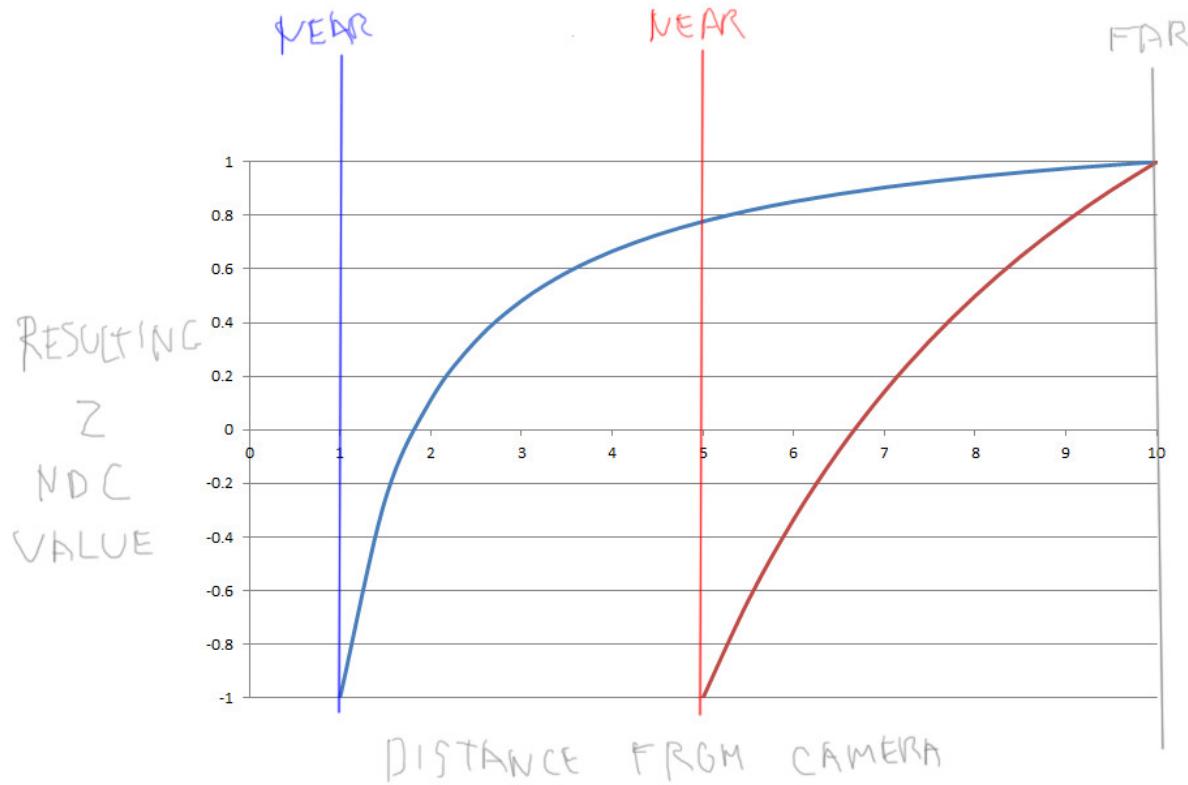


For example, if you're rendering a sheet of paper on top of a desk, you can easily get z-fighting, even if you've modeled everything correctly and the sheet is slightly above the desk. At some

pixels the z-value of the paper and the desktop will have the same z value and the desktop can then bleed on through.

[draw near and far planes at 1,5 (in blue, red) as tick marks on horiz. axis and both colors at 10. Label axes: **Y axis is NDC Z value, X axis is distance from camera.**]

[try to leave a bit of room to right for train tracks photo]



The z-depth range of values is spread between the near and far distances. It's clear that having these two distances close together directly benefits precision.

However, with the perspective transform in particular, you want to move the near plane as far from the eye as possible. Here's an example. Say we have our near plane at a distance of 1 unit away from the camera and the far plane 10 units away. The NDC z-depth does not vary linearly but instead forms a hyperbolic curve. For example, say we have an object at 7 units away. The NDC Z value is about 0.9 when the near distance is 1 unit. In other words, the z-depth of more distant objects are relatively higher. These objects that are farther away have to share a small

range of z-depth values and so are more likely to exhibit z-fighting.

[may sneak in a photo of train tracks, if there's room.

<http://commons.wikimedia.org/wiki/File:%E3%83%88%E3%83%AD%E3%83%83%E3%82%B3%E7%8E%8B%E5%9B%BD%E7%BE%8E%E6%B7%B1%E6%B2%BF%E7%B7%9A%E9%A2%A8%E6%99%AF%E7%99%BD%E6%A8%BAP6260564.jpg>

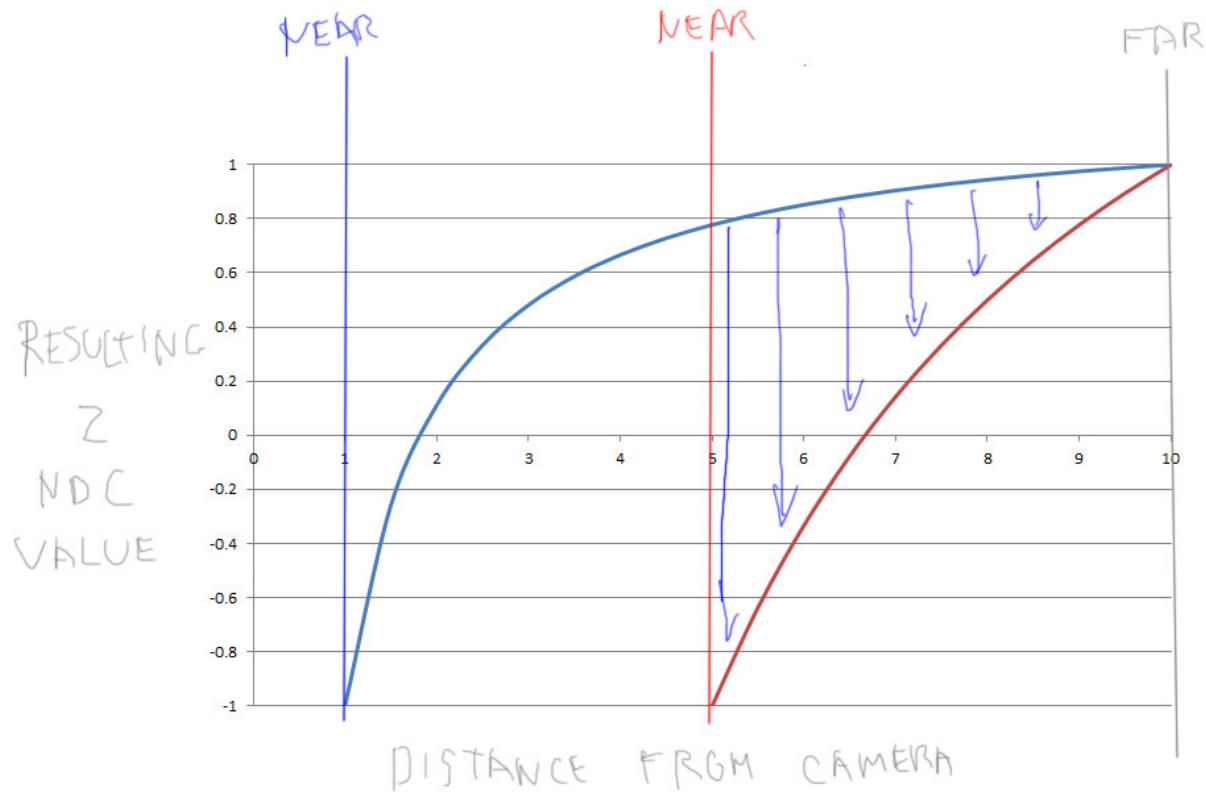
]



The reason the z-depth values vary in this non-linear way has to do with interpolation. We want straight lines to stay straight when using perspective projection. I won't prove it to you here, but think of train tracks disappearing into the distance. Near the camera the railroad ties are visually far apart; as you look towards the horizon the tracks get closer and closer together. The distance

between the tracks is the same, of course, and the tracks stay straight, but the distance between them on the image changes. The W value in computer graphics is interpolated linearly but when used for division gives us this differing rate of change.

[Now mark on graph where the blue line is at distance 5, and draw arrows from the blue line from 5 to 10 downwards to the red line.]



To get back to setting the near and far planes, say we're able to safely move the near plane to a distance of 5 units and not cause clipping. We're effectively taking this piece of our original graph and stretching it to our new range. First, we get the simple benefit of having a smaller range between the near and far. We also get a more nearly linear graph. The more you increase the near plane relative to the far plane, the slower the NDC z-depth actually goes to 1.0. [point to curves]

The long and short is that moving the near plane away from the camera has a large benefit, much larger than moving the far plane in by a similar distance. Of course, this all begs the question: how do we know where to set these planes? The far distance is usually relatively easy to compute: either know in advance or perform some rough computation to determine the distance to the farthest object in the scene. The near clipping plane is trickier. You usually have to have some rules, such as not letting your camera get too close to the walls, or use some rule of thumb such as that the near plane will be one thousandth the distance of that of the far

plane. Some more elaborate systems will set the near plane very close and perform a separate quick render of nearby objects to determine a good distance for the setting for the scene. There's no single, perfect answer.

[Additional Course Materials:

You can see the effect on precision with this [z-depth calculator](http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html). There's more about optimizing your z-depth range [in this article post](<http://outerra.blogspot.com/2012/11/maximizing-depth-buffer-range-and.html>).

For more on the stencil buffer, [this page](http://en.wikibooks.org/wiki/OpenGL_Programming/Stencil_buffer) is a start. Really, a book such as [“the Red Book”](<http://www.amazon.com/OpenGL-Programming-Guide-Official-Learning/dp/0321552628?tag=realtimerenderin>) or [McReynolds & Blythe’s](<http://www.amazon.com/exec/obidos/tg/detail/-/1558606599?tag=realtimerenderin>) are good for getting a serious education about its uses.

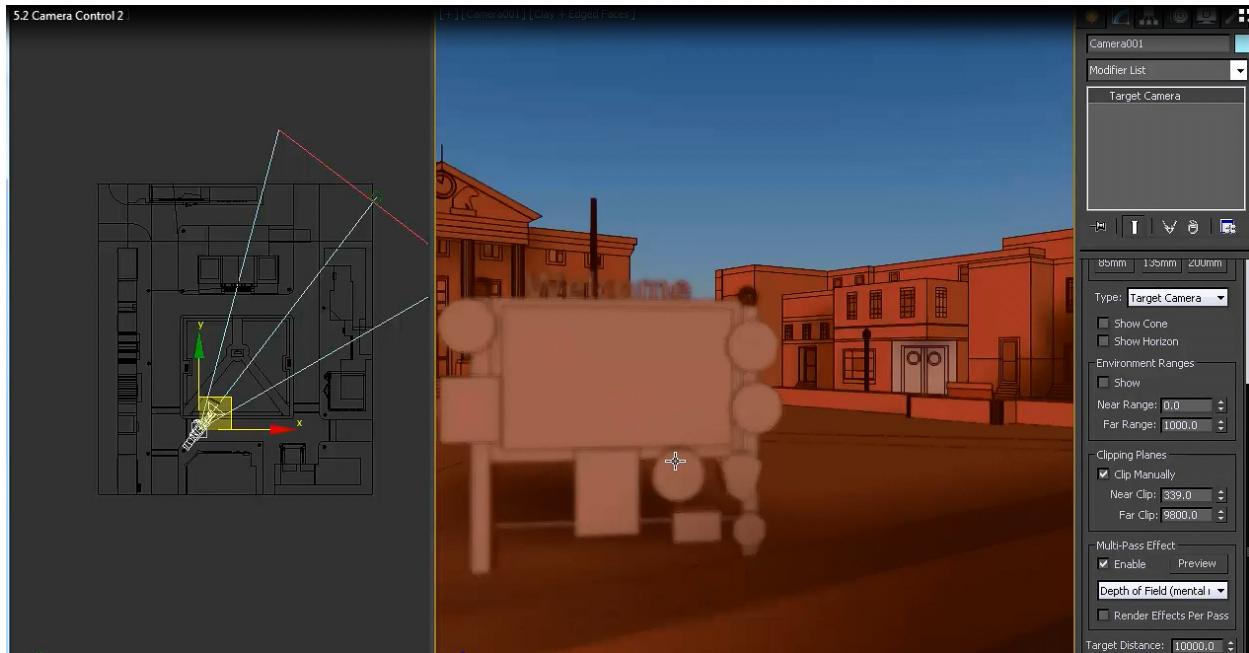
]

[end part 9 recorded 3/28]

Lesson: Depth of Field

[recorded 3/28, part 10]

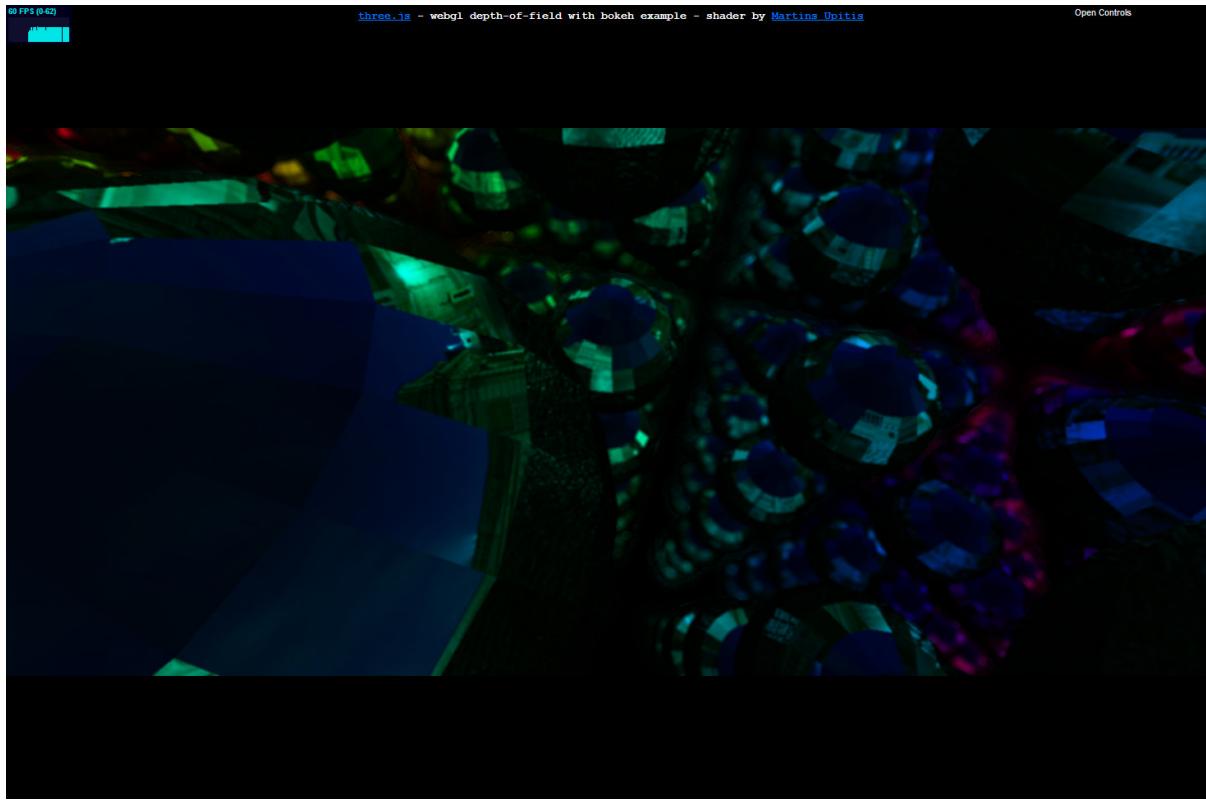
[video Unit6_FieldOfView, “5.2 Camera Control 3.mp4” - starts at 1:01 to 1:20 - let it run, then run the three.js demo footage]



There's one other feature of cameras that I'm going to mention in passing here: depth of field. This is the idea of simulating a real camera, where you can focus the lens to be at a particular distance. In depth of field algorithms you control the focal distance and also how blurry objects will appear when not in focus.

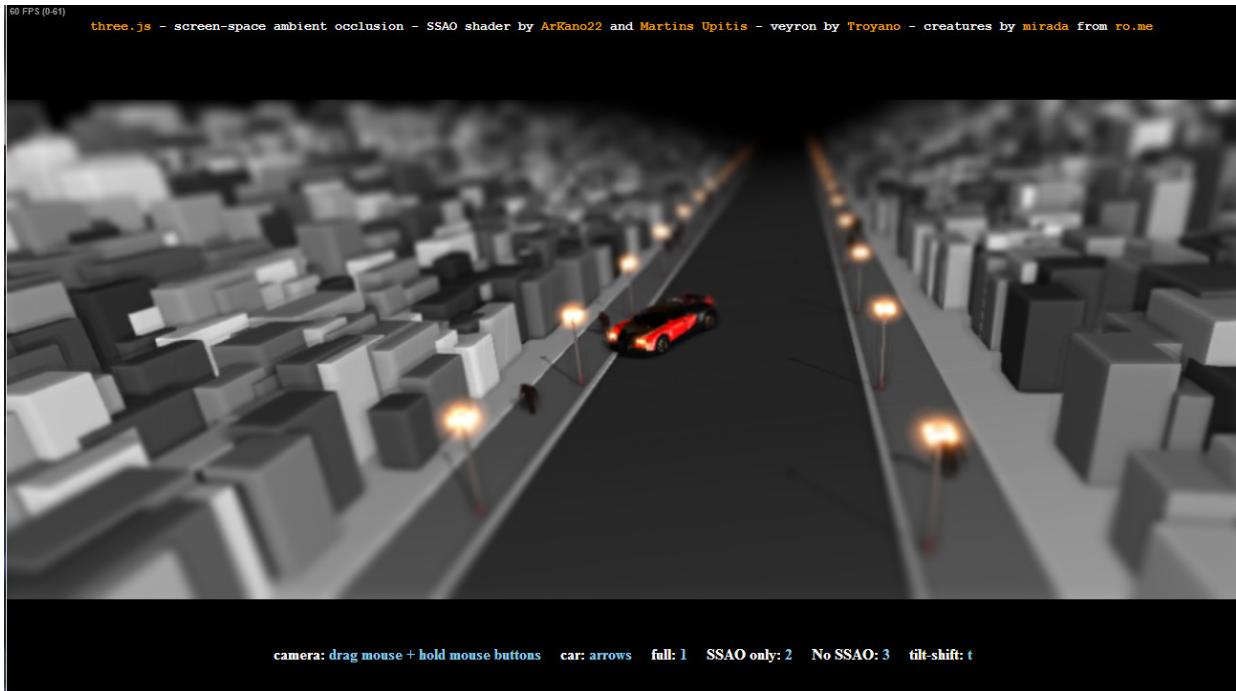
This type of algorithm is often done as a post-process. In other words, data is collected by rendering the scene, then image processing of some sort is used on this data to produce the final depth of field image. The difficult part is getting objects in the foreground to be blurry and also properly blend with the objects in focus behind them.

[http://mrdoob.github.com/three.js/examples/webgl_postprocessing_dof.html]



Three.js has a demo showing a depth of field effect. It will probably show up a bit dark here, so see the additional course materials for the link and give it a try. It avoids the problematic blurry foreground objects by putting the focus nearby, so that only distant objects are fuzzy.

[http://alteredqualia.com/three/examples/webgl_postprocessing_ssao.html]



There are other less expensive techniques you can do to get a depth of field effect. Here's a simple tilt-shift post-process that gives foreground and background blur. No scene information is used, a variable blur is just added to the top and bottom of the final image.

[Additional Course Materials:

The three.js depth of field demo is

[here](http://mrdoob.github.com/three.js/examples/webgl_postprocessing_dof.html).

AlteredQualia's tilt-shift demo is

[here](http://alteredqualia.com/three/examples/webgl_postprocessing_ssao.html).

]

Lesson: Window Coordinates

[record 3/28, part 11]

[**WPVMO**]

We've had a series of transforms applied to the object: the object's model transform, followed by the view and projection transforms generated by the camera. The perspective divide converts from clip coordinates to Normalized Device Coordinates. There's one last transform, and it's a simple one: moving from Normalized Device Coordinates to Window Coordinates. In other words, how do you move from a space of -1 to 1 in X, Y, and Z to an image with a depth buffer.

The answer is simple enough: add 1, divide by two, then multiply by the window's resolution.

$$Xw = ((X+1)/2) * Xres$$

$$Yw = ((Y+1)/2) * Yres$$

$$Zw = ((Z+1)/2) * Zres$$

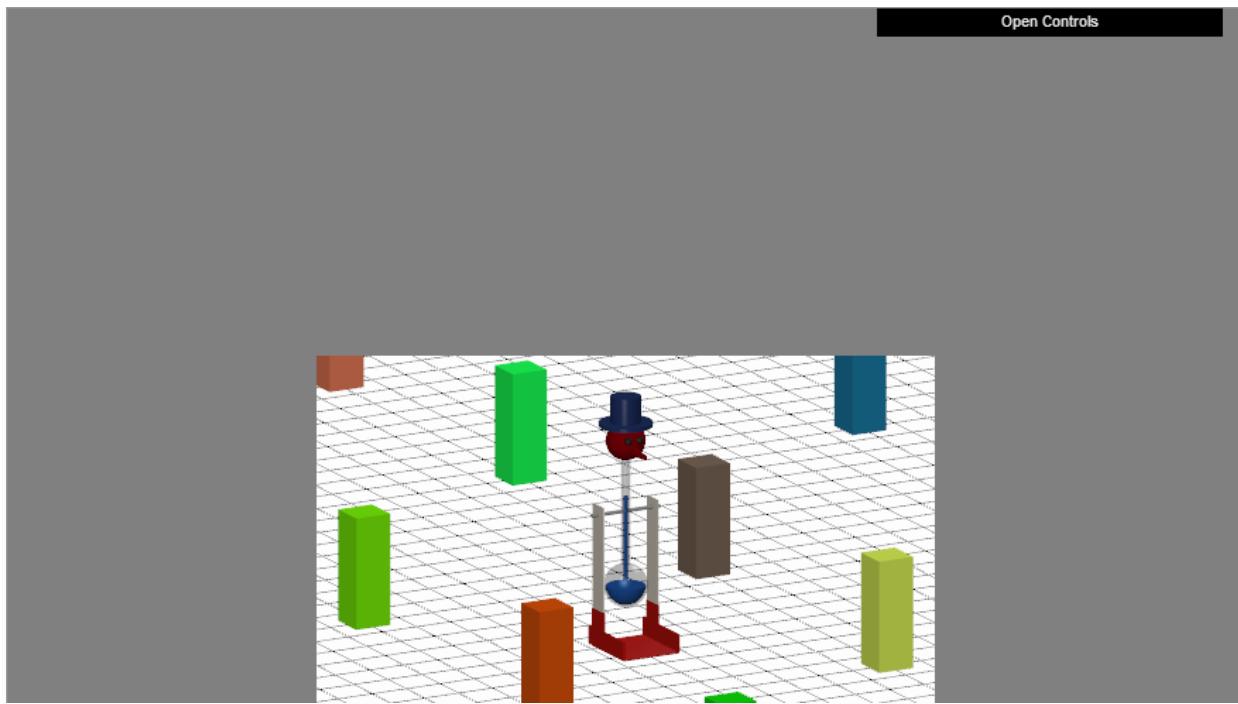
I should mention at this point that the other popular API, DirectX, has Normalized Device Coordinates for the Z value that range from 0 to 1 instead of -1 to 1. It doesn't really matter what range is used, it's just important to know that this range can vary. The X and Y ranges of going from -1 to 1 is standard throughout any system I've ever seen.

In three.js you select some part of the screen using the setViewport method on the renderer.

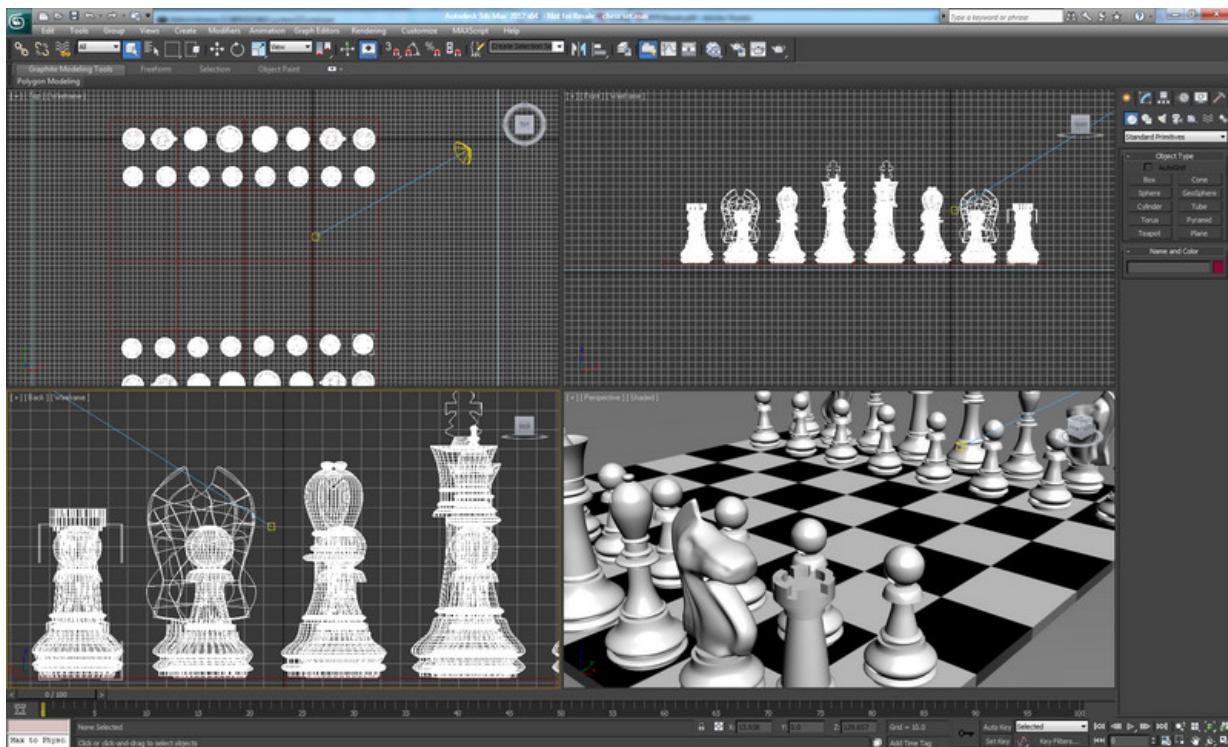
```
var canvasWidth = window.innerWidth;
var canvasHeight = window.innerHeight;
// small, middle of bottom of screen
renderer.setViewport(
    0.25 * canvasWidth, 0,
    0.5 * canvasWidth, 0.5 * canvasHeight );
```

```
var canvasWidth = window.innerWidth;
var canvasHeight = window.innerHeight;
// small, middle of bottom of screen
renderer.setViewport(
    0.25 * canvasWidth, 0,
    0.5 * canvasWidth, 0.5 * canvasHeight );
```

[Next to it]



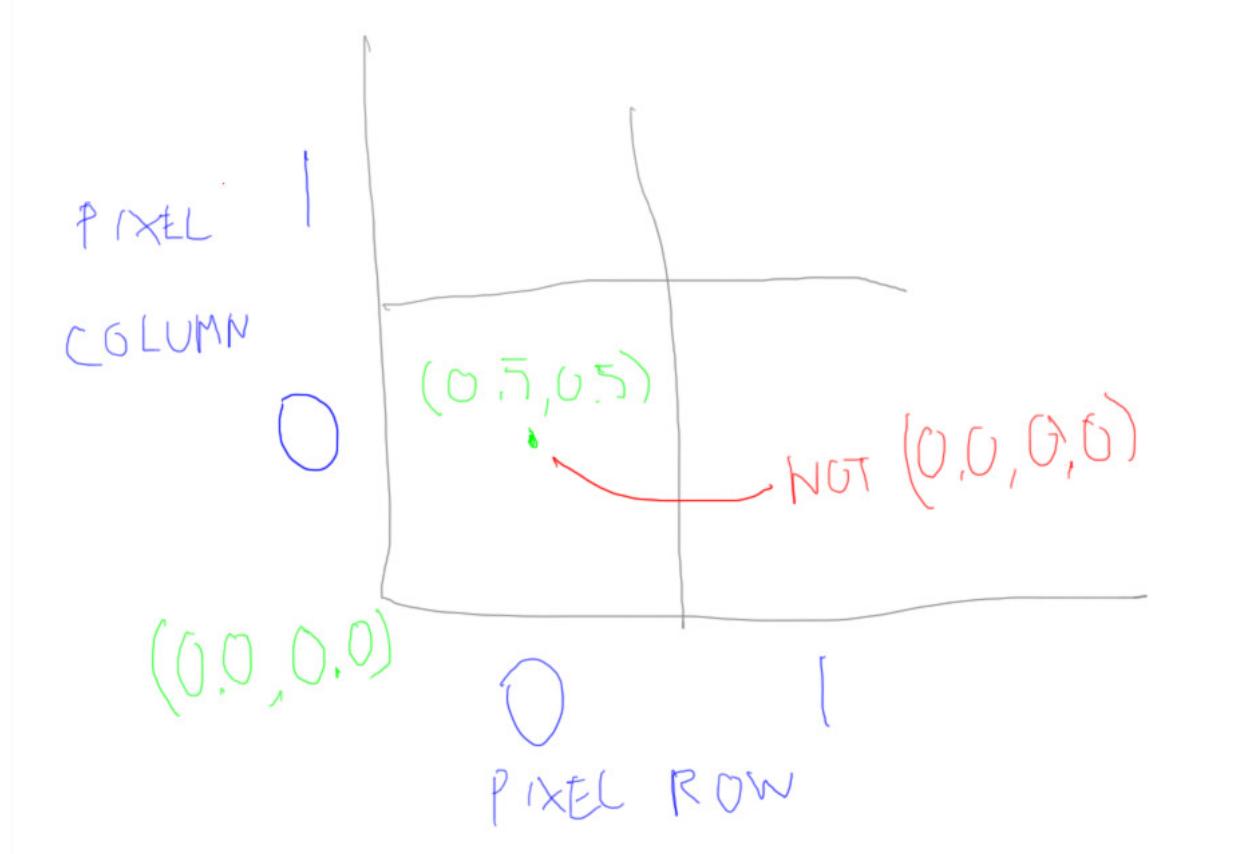
You give the lower left pixel corner and the width and height. The settings here say to put the lower left corner of the viewport a quarter of the way to the right of the origin, and that the viewport itself should be half the width and height of the window.



It's possible to have multiple viewports. Each viewport defines a piece of the image, and you render separately to each one. This is very handy in modeling, where you can have a few different views of the scene, possibly rendered in different ways.

The conversion I give from NDC to window coordinates here assumes that the lower left corner of the image is the origin, at 0,0. It's worth mentioning is that there can be a flip in the Y axis within some systems, such as the Document Object Model. Some systems that display the image generated consider that the upper left corner is 0,0. If you see a flip in the Y direction during manipulation of the resulting image, this is likely the mismatch.

[draw center of pixel, lower left corner view]



While we're talking about 0,0, please note that the lower left corner of the lower left pixel is 0,0. The center of the pixel is not 0.0, 0.0; it is 0.5, 0.5. Almost all the time this is how you want to consider the center of the pixel. DirectX 9 got it wrong, making the center 0.0, 0.0; they fixed this in DirectX 10. I've seen textbooks talk about 0.0,0.0 as the center - don't believe them. Using 0.0, 0.0 as the center of the pixel has the odd effect of making the lower left corner -0.5,-0.5. It makes simple conversion between floats and integers trickier much of the time. With the proper center you just drop the fraction. I've occasionally seen circumstances where offsetting half a pixel can

make things work out more efficiently, but much of the time you don't want to do this.

[Additional Course Material:

Lee Stemkoski has commented code for how to put [two viewports](<http://stemkoski.github.com/Three.js/Viewports-Dual.html>) and [four viewports](<http://stemkoski.github.com/Three.js/Viewports-Quad.html>) on the screen.

Three.js has [an interesting demo](http://mrdoob.github.com/three.js/examples/webgl_multiple_canvases_complex.html) showing an artistic viewport effect.

A fairly good overview of the OpenGL transformation pipeline can be found [on this page](http://www.songho.ca/opengl/gl_transform.html#projection). Although dated, [this page](<http://www.glprogramming.com/red/chapter03.html>) gives a more friendly walk down the camera pipeline.

The ancient article, "What are the Coordinates of a Pixel?" by Paul Heckbert, in the book "[Graphics Gems](<http://www.amazon.com/exec/obidos/ASIN/0122861663?tag=realtimerenderin>)", is worth tracking down. It's my strongest argument for why the center of a pixel or texel has the coordinates 0.5,0.5, not 0.0,0.0.

]

Lesson: Antialiasing

[draw pixels, note how we have a rough edge.]



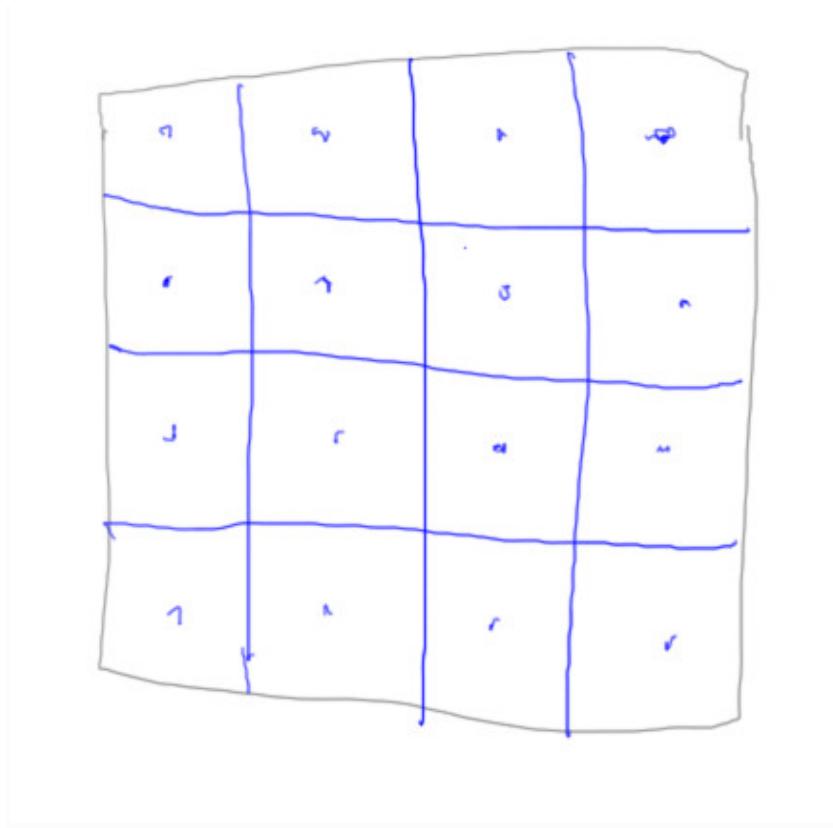
Now that we're at the pixel level, let's talk a bit about antialiasing.

By default, when we rasterize a scene, we find what is in the center of each pixel. This can give some fairly bad results, as edges of triangles will either be considered to fully cover a pixel or not cover it at all. The binary nature of this process causes what's called **aliasing**, giving a ragged look to edges. Informally this problem is called "**the jaggies**" or "**stair stepping**"; when you see it in animation it's called "**the crawlies**".

What we'd like to do is get a nice, smooth result, where each edge pixel is shaded proportionally to how much of it is covered by each object overlapping it. If the pixel is mostly covered by a triangle, use more of the triangle's color; less, use less.

The hard part is in figuring out this coverage. It's expensive to compute, store, and blend triangle areas for each pixel, though maybe this will happen someday, about the same time we get personal jetpacks.

[draw grid and show 4x4 samples]

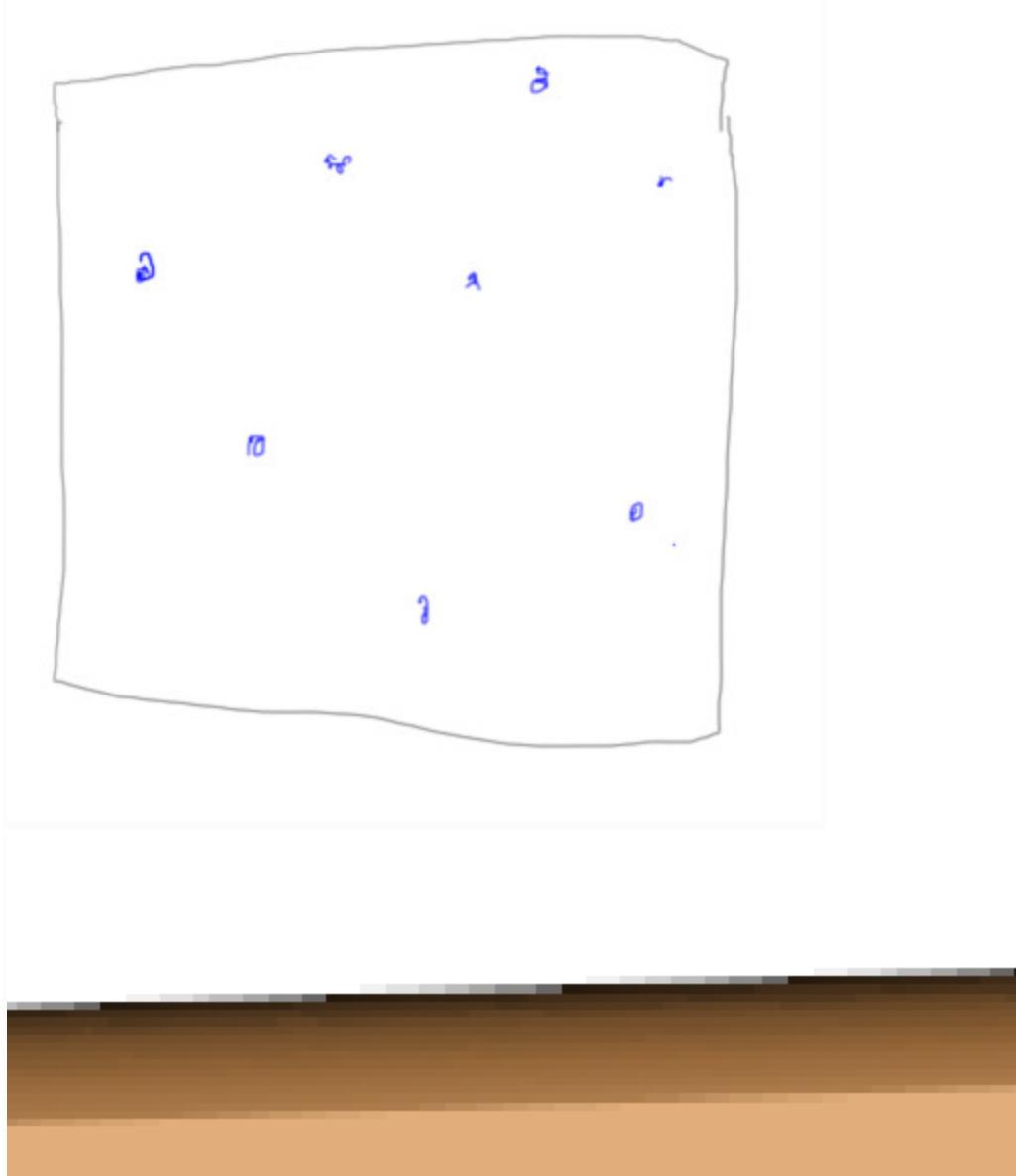


In the meantime there have been a huge number of antialiasing schemes proposed for interactive 3D graphics. On one end of the spectrum is **supersampling**, where you simply create a higher resolution image and then use all these extra samples to make the final image. For example, for a 1000x1000 pixel image you might render it at a resolution of 4,000x4,000. Now each pixel has 16 pixels associated with it in the high-res image. Blend these together and

you get a better result.

This scheme is considered A Bad Idea, for a number of reasons. One is that it's expensive in both memory and processing costs. Another is that sampling in a little 4×4 grid is not much help in fixing the jaggies for nearly horizontal or nearly vertical lines.

[draw MSAA with some 8 queens sampling pattern]



[show all three.]

A scheme commonly supported by the GPU is called [**MSAA, multisampling antialiasing**](#). The

idea here is to compute a shade for the whole fragment once, and compute the geometric coverage separately. In addition, a different sampling pattern than a grid is used. Doing so helps nearly horizontal and vertical lines considerably. This sampling pattern does vary depending on the hardware company making the GPU. The main costs here are processing and storage, but these costs are considerably less and give considerably more bang for the buck than brute force supersampling.

This is generally the form of antialiasing used by default by WebGL. I say “generally” because there’s just a toggle for antialiasing, on or off, and it’s not specified what form of antialiasing is used.

```
renderer = new THREE.WebGLRenderer( { antialias: true } );
```

To turn antialiasing on in three.js consists of setting a single boolean parameter called “antialias”. I should note that turning this on doesn’t necessarily do anything. It depends on whether the GPU supports antialiasing - most should - and whether the browser decides to allow it. Sometimes there are bugs that make it safer to keep antialiasing off. Sadly, most of the video lessons we’ve made showing three.js demos do not have antialiasing on, just because of such a bug. Welcome to the bleeding edge of technology - no one said there’d be cake!

[FXAA: Show closeups of AA on and off (use NPR Effects Sandbox)]



Another class of algorithms for antialiasing perform filtering on the image. This is a relatively new class of techniques. The first of these, called ***morphological antialiasing, or MLAA***, was developed in 2009. The idea is to use the existing image, and possibly other data, to detect sharp edges. When these are found, try to smooth just these edges by using nearby pixels. Amazing, such techniques work pretty well, though can fail on thin lines, text, and other special cases. One of the most popular filtering techniques is called ***FXAA***, which needs just the image itself to operate. I’ve used it myself in products, and it’s even included in three.js. I think part of the popularity is that the author, Timothy Lottes, put a serious amount of effort into both making this shader run on just about every GPU known to humankind and thoroughly documenting the code itself.

[one long breathless sentence]

Just to spell it out, in case you didn't get the hint: please do document your code, especially if you plan on having anyone see it, which of course anyone can do if you're using WebGL or three.js, so I'll say it again: document your code!

[Additional Course Materials:

See the FXAA algorithm in use in [this three.js demo](http://mrdoob.github.com/three.js/examples/webgl_shading_physical.html), though you'll have to disable it yourself to see the difference.

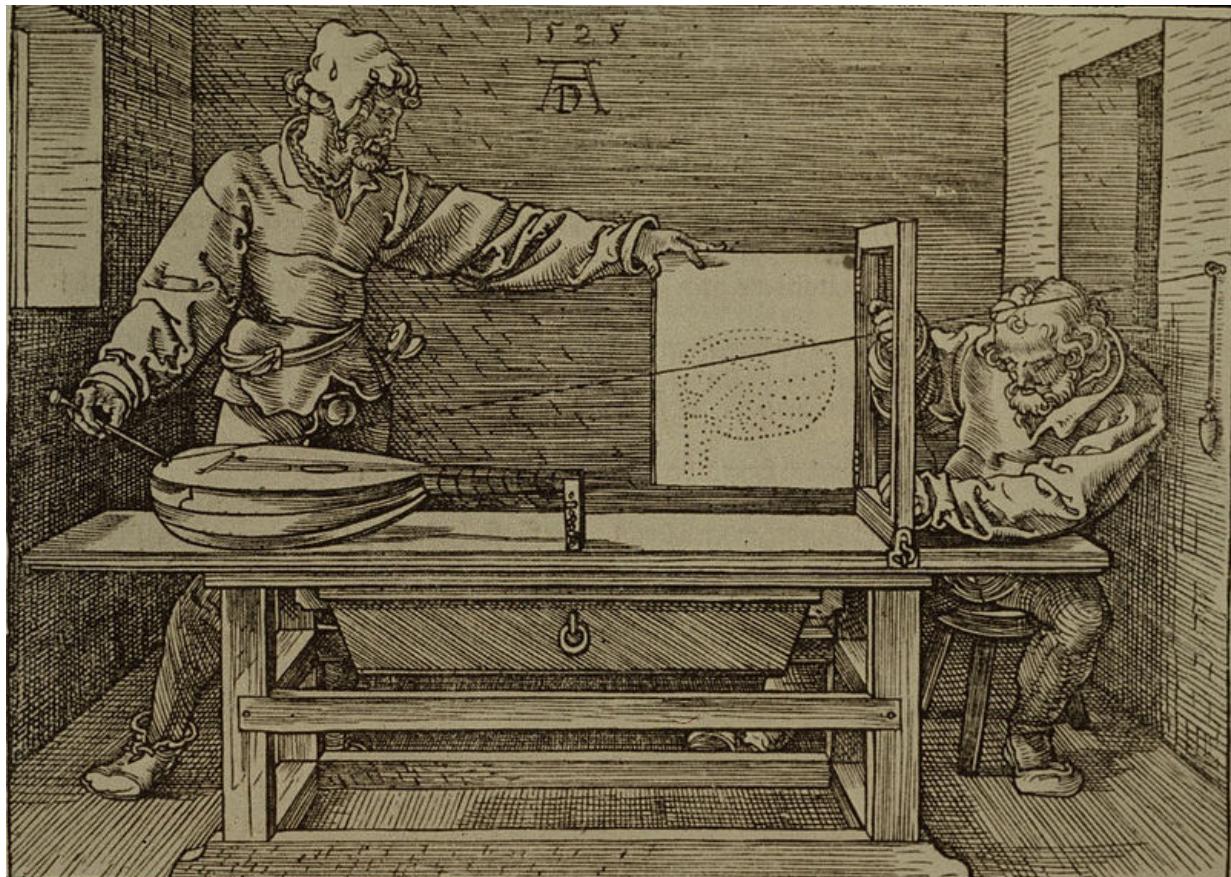
My view of FXAA is [here](<http://www.realtimerendering.com/blog/fxaa-rules-ok/>). There was a whole [SIGGRAPH course](<http://iryoku.com/aacourse/>) about filtering techniques.

Antialiasing does not work on Macs due to [an obscure OpenGL driver bug on NVIDIA chipsets](<http://code.google.com/p/chromium/issues/detail?id=159275>). By the time you read this, it may be fixed.

]

Lesson: Conclusion

[http://commons.wikimedia.org/wiki/File:D%C3%BCrer_-_Man_Drawing_a_Lute.jpg]



You now have knowledge of all the basic elements for creating a scene: objects, lights, and cameras. You have a considerably easier time of it than these people, as far as rendering the scene goes - the computer does all that for you, letting you focus on how to craft the objects themselves.

In the next unit we'll show how to significantly increase the visual richness of models in your virtual world through the use of texture mapping.

[Additional Course Materials:

[Man Drawing a

Lute](http://commons.wikimedia.org/wiki/File:D%C3%BCrer_-_Man_Drawing_a_Lute.jpg) from Wikimedia Commons.

]

[finished recording 3/28, part 11]

Problem Set: Camera

[begin recording 4/3 part 1]

Problem 7.1: Near Plane of Zero

Here's the perspective projection matrix:

$$\begin{bmatrix} \frac{1}{\text{aspect} \cdot \tan(\text{FOV} / 2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\text{FOV} / 2)} & 0 & 0 \\ 0 & 0 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} & \frac{2 \cdot \text{near} \cdot \text{far}}{\text{near} - \text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

I hate it when objects get clipped by the near plane. **What happens if I set the “near” distance to zero**, so there's no near plane to clip against?

- () The homogeneous coordinates cannot be turned into NDC values.
- () All the resulting NDC Z values are zero.
- () The X and Y values become mirrored (negated).
- () The resulting NDC Z values are almost all one.

Answer

[show matrix computed with near = 0: -1 and 0 in the third row. Show equations of what the new Z and W are:

Zout = -Zin
Wout = -Zin

]

If you look at this matrix and examine its effect on the various coordinates, you'll notice the

following. First, the last column is all zeroes, so the W value of the coordinate will have no effect on the result. This leaves only the incoming Z value of the coordinate affecting the output Z and W values. Whatever the Z value is coming in, the resulting Z and W values will be that original Z value multiplied by -1. In other words, the output Z and W values will be identical. When you divide Z by W, the result will always be 1, except for the case where W is 0. In other words, the distance of an object from the camera will almost always be 1, which is not a useful result if you're trying to use the z-buffer.

Problem 7.2: Graphics Pipeline Coordinates

Here are a bunch of different coordinate spaces. You're sending a model down the pipeline.

Select the coordinate type the model starts in and which types it may go through.

- A. View Coordinates
- B. Orthogonal Coordinates
- C. Window Coordinates
- D. Universal Coordinates
- [list other half to right]
- E. Normalized Device Coordinates
- F. Model Coordinates
- G. Polar Coordinates
- H. Clip coordinates
- I. World Coordinates

Starts in: _____ to _____ to _____ to _____ to _____ to _____

In practice we might skip one of these coordinate steps. What I'm looking for here is for every blank to have a letter from this list.

Answer

[F, I, A, H, E, C]

The object starts in model coordinates, choice F, and is transformed to world coordinates, choice I. The object then gets transformed to view space, A, so that the world is oriented with respect to the camera's frame of reference. In practice we often go directly from object space to camera space with a single transform.

The projection matrix is applied to get to clip coordinates, choice H, and the coordinates are divided by W to get the model into Normalized Device Coordinates, choice E. Finally, the data is converted to window coordinates to get it into an image to put on the screen, choice C.

[Additional Course Materials:

The idea for this question is from Ronen Barzel, with permission. The [slidesets for his graphics course](http://graphics.ucsd.edu/courses/cse167_w06/) are worth working through.

]

Problem 7.3: 4 Viewports

I've set up two viewports on the screen, a perspective view and a top view orthographic camera. I should mention there is one extra thing to set on the renderer in three.js when using multiple viewports:

```
// don't clear when multiple viewports are drawn
renderer.autoClear = false;

// don't clear when multiple viewports are drawn
renderer.autoClear = false;
```

This setting tells three.js to not clear the screen before rendering the viewport.

[Draw top view, from side, and how its code works]

```
// OrthographicCamera( left, right, top, bottom, near, far )
topCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
// set X to be the up axis
topCam.up.set( 1, 0, 0 );
```

```
// OrthographicCamera( left, right, top, bottom, near, far )
topCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
// set X to be the up axis
topCam.up.set( 1, 0, 0 );
```

The top view works by defining a volume in space. The up vector for the view is set, somewhat arbitrarily, to be the +X axis.

```
// top view
topCam.position.copy( cameraControls.target );
// move up a unit and look down at target
topCam.position.y +=1 ;
topCam.lookAt( cameraControls.target );

// top view
topCam.position.copy( cameraControls.target );
// move up a unit and look down at target
topCam.position.y +=1 ;
topCam.lookAt( cameraControls.target );
```

During rendering the camera position itself is set to be up one unit, looking down at the target. The idea is to look at the target and then move back a bit, to establish the lookat direction. Note that the lookat and up directions should never be parallel!

Your task is to add two more viewports. Add a front viewport to the upper left, side to the lower right. ***The front viewport should look up along the positive X axis, the side viewport down along the negative Z axis.***

[show final result: unit7-db_4view_solution.js]

Both front and side viewports should track the target position similar to how the top viewport works. When you're done the program should look like this.

Answer

```

frontCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
frontCam.up.set( 0, 1, 0 );
sideCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
sideCam.up.set( 0, 1, 0 );

frontCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
frontCam.up.set( 0, 1, 0 );
sideCam = new THREE.OrthographicCamera(
    -aspectRatio*viewSize / 2, aspectRatio*viewSize / 2,
    viewSize / 2, -viewSize / 2,
    -1000, 1000 );
sideCam.up.set( 0, 1, 0 );

```

There are two parts to the solution. The front camera and side camera are created in a similar fashion to the top camera. The “up” direction needs to be set to Y up instead of the top camera’s X up setting. Really, the default is Y up, but it doesn’t hurt to set it.

```

// front view
frontCam.position.copy( cameraControls.target );
// move down along X axis a unit and so look at front at bird
frontCam.position.x -=1 ;
frontCam.lookAt( cameraControls.target );
renderer.setViewport( 0, 0.5*canvasHeight, 0.5*canvasWidth, 0.5*canvasHeight );
renderer.render( scene, frontCam );

// side view
sideCam.position.copy( cameraControls.target );
// move back up along Z axis a unit and so look at side at bird
sideCam.position.z +=1 ;
sideCam.lookAt( cameraControls.target );

```

```

renderer.setViewport( 0.5*canvasWidth, 0, 0.5*canvasWidth, 0.5*canvasHeight );
renderer.render( scene, sideCam );

// front view
frontCam.position.copy( cameraControls.target );
// move down along X axis a unit and so look at front at bird
frontCam.position.x -=1 ;
frontCam.lookAt( cameraControls.target );
renderer.setViewport( 0, 0.5*canvasHeight, 0.5*canvasWidth, 0.5*canvasHeight );
renderer.render( scene, frontCam );

// side view
sideCam.position.copy( cameraControls.target );
// move back up along Z axis a unit and so look at side at bird
sideCam.position.z +=1 ;
sideCam.lookAt( cameraControls.target );
renderer.setViewport( 0.5*canvasWidth, 0, 0.5*canvasWidth, 0.5*canvasHeight );
renderer.render( scene, sideCam );

```

During rendering, the front and side cameras are repositioned almost identically to how the top camera is moved.

[end recording 4/3 part 1]

Problem 7.4: Rear-View Camera

[begin recording 4/3 part 2]

In this exercise you'll implement a rear-view camera. It's not quite a rear view mirror, because left and right will not be reversed. Right now the rear-view camera is just focused on the center of the model; the position and target never change. Your job is to make the rear-view camera actually look backwards, directly away from wherever the full-screen camera is looking. For its position, use the same location as the forward camera. You'll probably need to use the camera's "*lookAt*" method, as well as some **Vector3** methods - at least I did.

[show unit7-db_rearview_solution.js]

I've set up the two viewports. The one tricky bit in the code is that scissoring is implemented. A scissor box defines a part of the screen that you want to clear. This is needed so that the rear-view camera viewport properly clears its area on the screen, since it overlaps the larger viewport.

Answer

```
renderer.render( scene, camera );  
  
rearCam.position.copy( camera.position );  
rearTarget.copy( camera.position );  
rearTarget.sub( cameraControls.target );  
rearTarget.add( camera.position );  
rearCam.lookAt( rearTarget );  
  
// rearview render  
renderer.enableScissorTest( true );  
  
renderer.render( scene, camera );  
  
rearCam.position.copy( camera.position );  
rearTarget.copy( camera.position );  
rearTarget.sub( cameraControls.target );  
rearTarget.add( camera.position );  
rearCam.lookAt( rearTarget );  
  
// rearview render  
renderer.enableScissorTest( true );  
  
[ draw target and position, show going to opposite target. ]
```

Here's the code added to the render() method. The position for the rear camera is copied from the regular camera. The rear camera's target is set by computing the vector from the front camera's target to the position and then adding the position itself to the vector.

Problem 7.5: Division by Zero

Given this perspective matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \\ \end{bmatrix}$$

[use figure of view situation for the matrix, then replace it with the vectors below]

Say you multiply a point with coordinate $[3, 7, 0, 1]$ [show vertically] by this matrix. I'll save you the effort: you get $[3, 7, -2.2, 0]$ [show vertically]. W is 0. If you try to divide -2.2 by zero without safety glasses on, the world ends. In fact, for any point $[X, Y, 0, 1]$ we'll get a W=0.

What do all these points have in common?

- () The points are exactly on one of the faces of the frustum itself.
- () The points are in the anti-frustum, a mirrored frustum behind the camera.
- () The points are on a plane parallel to the near plane that goes through the origin.
- () The points are behind the camera.

Answer

[show the side view again]

A set of points described by Z=0 is a plane going along the X and Y directions through the origin, so the third answer is the correct one. This is where the camera is located in projection space.

The points in the last three answers are all located outside the frustum. These points should be culled and any edges formed from them clipped to the view volume. This culling and clipping is done before division by W. Points on or inside the view frustum are guaranteed to have valid W values.

[end recording 4/3 part 2]

Problem 7.6: Camera Matrices Matching

[NOTE: 38-q_x/38-s-x should be moved from the Final question set

to here]

[MIX UP ORDER, make it clear where answer goes, use a 4-pane layout]

[view matrix]

(-0.46 0 -0.89 -1160

(0 1 0 350

(0.89 0 -0.46 -600

(0 0 0 1

(1.60 0 0 0

(0 3.17 0 0

(0 0 -1.0005 -2.0005

(0 0 -1 0

(727.5 0 0 727.5

(0 -351 0 351

(0 0 0.5 0.5

(0 0 0 1

Which matrix is which:

A. View Matrix

B. Perspective Matrix

C. NDC to Window Matrix

Answer

This matrix has values in the first three positions in the last row, so it is a projection matrix. Of our choices, only the perspective matrix is a projection.

Rotations affect the off-diagonal elements of upper 3x3 of the matrix. The view matrix is the only one of the three choices with rotations in it.

This leaves this third matrix to be the NDC to Window Matrix. This conversion is a scale and translate, which is what this matrix certainly contains.