Lesson 2: Points, Vectors, and Meshes

The basics of 3D geometry definition

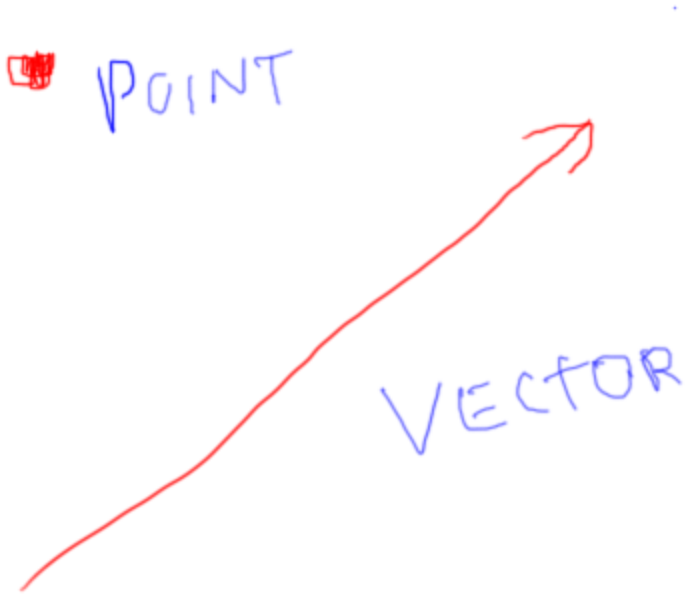
Lesson: Coordinate System

In this unit I'm going to talk about how geometric objects are defined for rendering. Along the way I'll discuss points and vectors and how these get used in computer graphics.

Let's dive right in. There are two distinct mathematical objects we use in computer graphics all the time: points and vectors. I'm going to focus on the 3D aspect, since that's what we commonly use. Working in two dimensions is similar; 3D is just 50% more interesting.

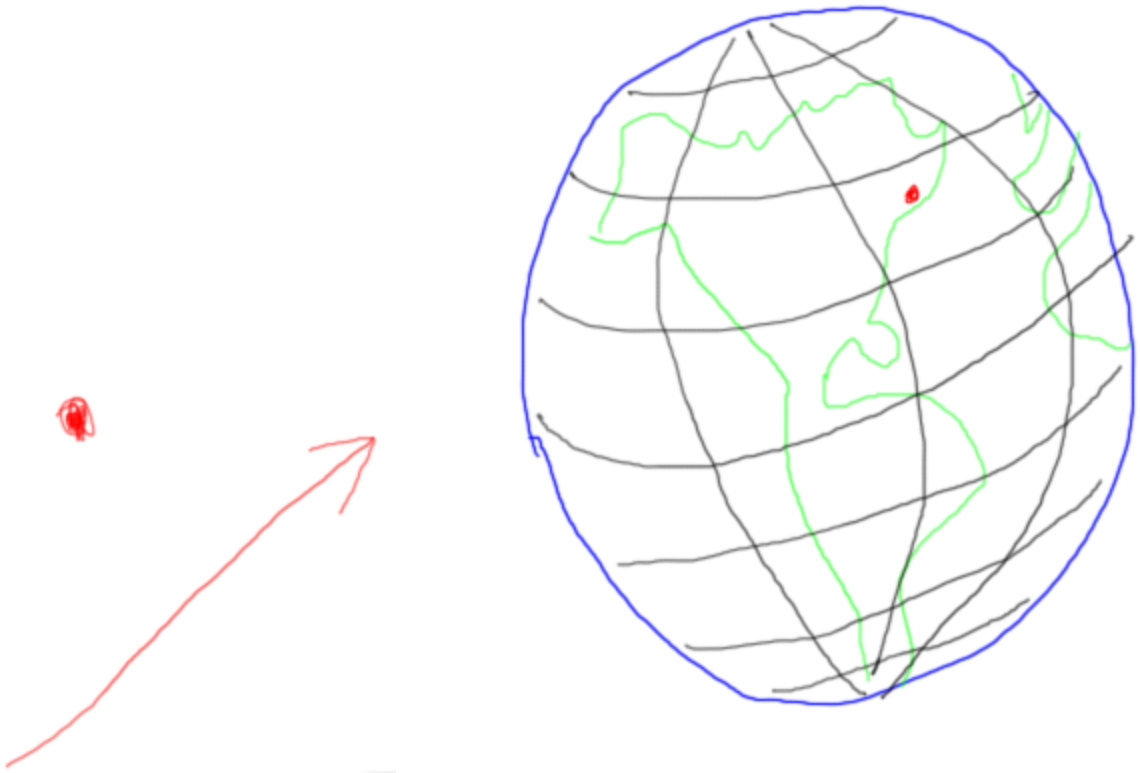
To start, we'll define a 3D point as a location in space. A 3D vector defines a direction in space.

[point and vector drawing]



A location in space has to be defined with respect to something else. For example, on the Earth we define a location in terms of its latitude, longitude, and altitude. This is a spherical *coordinate system*, in that the lines of latitude and longitude wrap around a sphere, the Earth.

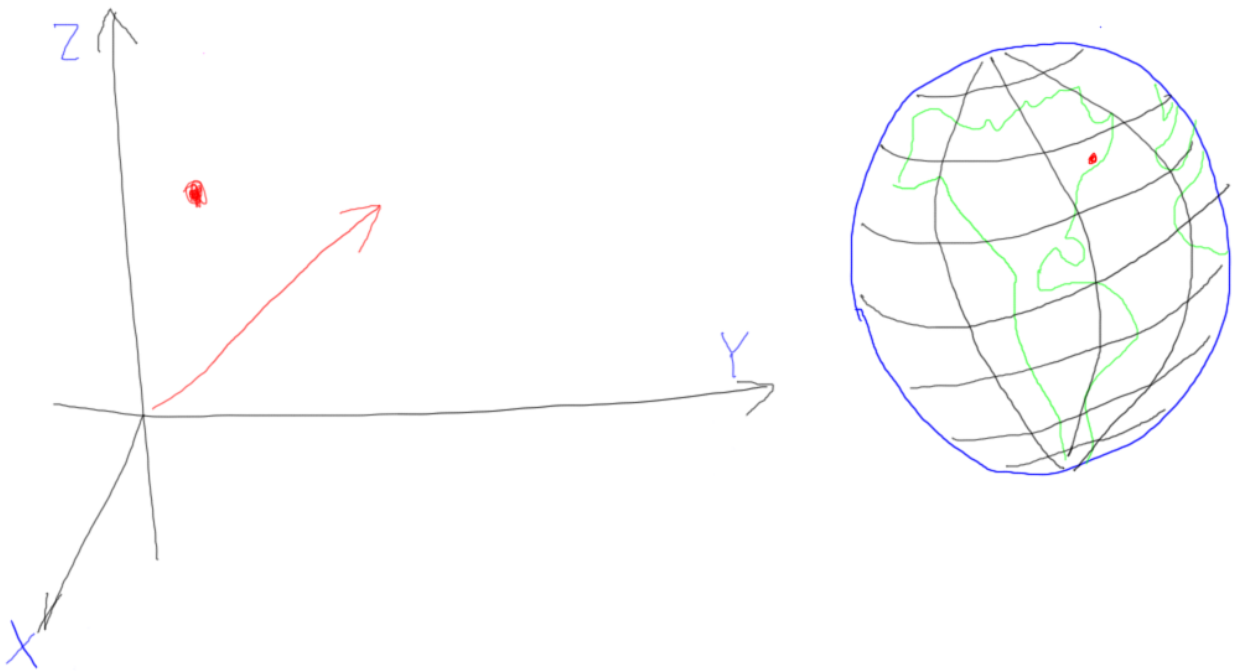
[draw earth with lat & long - **note to self: probably should label latitude and longitude**, since these matter later.]



Lesson: Cartesian Coordinates

In 3D computer graphics we usually use a 3D Cartesian coordinate system.

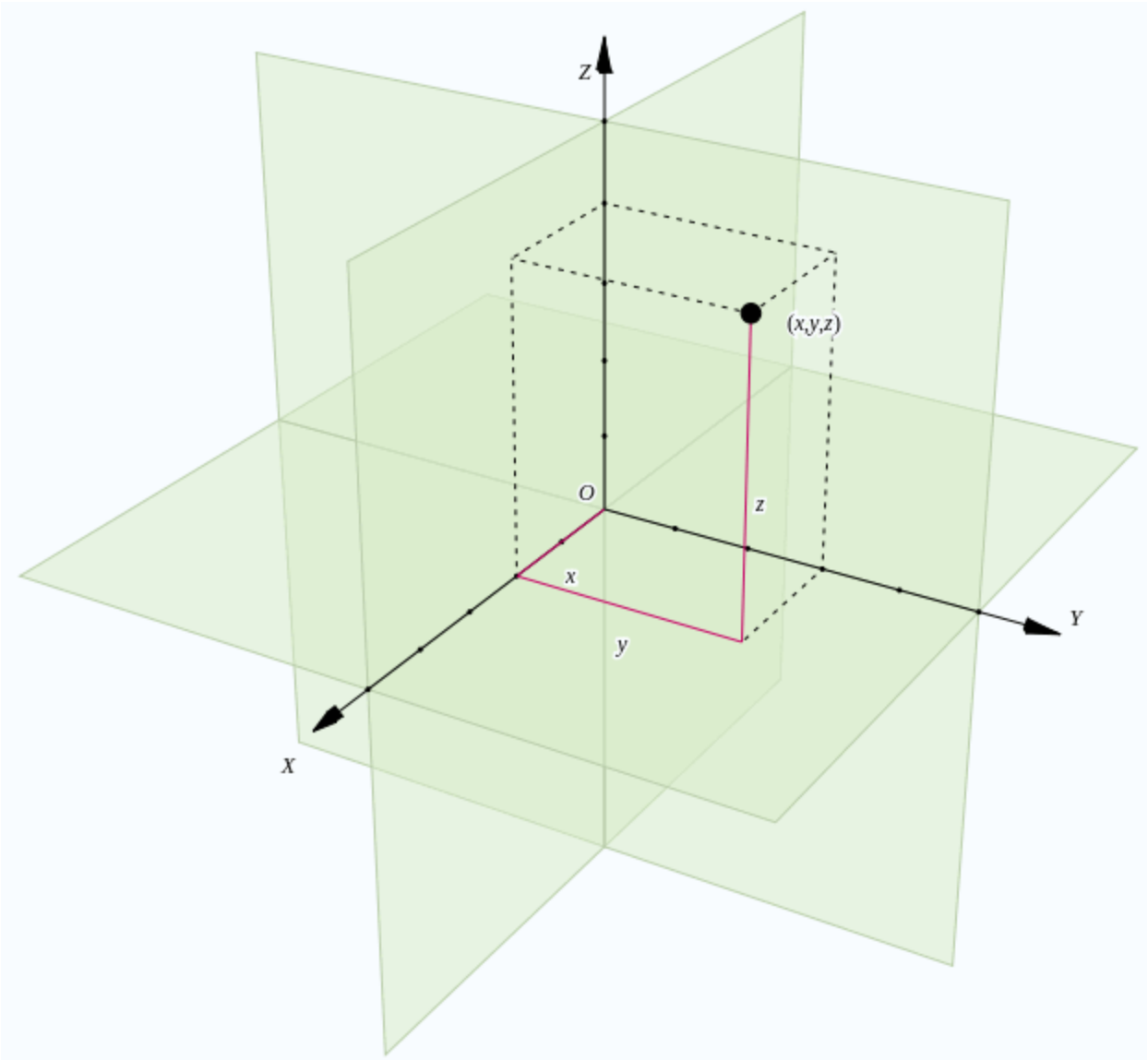
[IMPORTANT: draw X and Y in plane, Z coming out]



In this system we have an origin and three direction vectors, called X, Y, and Z. The origin is some fixed point whose location everyone agrees on. This point can be anything: I could build a world based on the location of a surveyor's marker, a dot drawn on a wall, or a reference point I simply imagine.

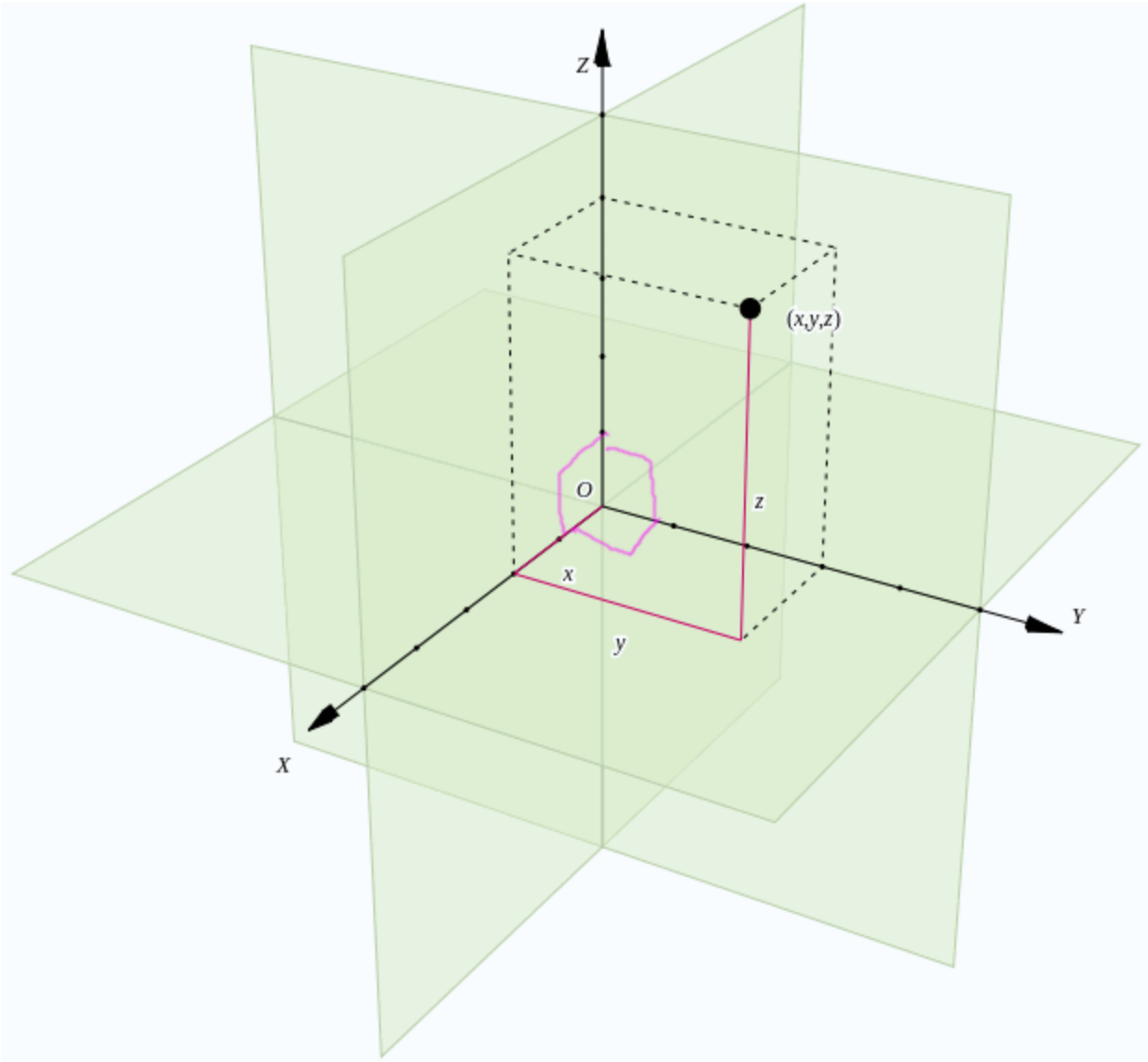
For example, if I'm designing a great new teapot, I don't really care where the rest of the world is located, so I could state that my origin is just a point in empty space. I might then build up the teapot's model so that this origin point is in the center of the bottom of the teapot.

[From http://en.wikipedia.org/wiki/Cartesian_coordinate_system]



Here's a prettier view. since I want you to see the 3D coordinates better. The three direction vectors define the axes of the coordinate system. They are normally each perpendicular to the other, that is, the angles here, here, and here are each 90 degrees.

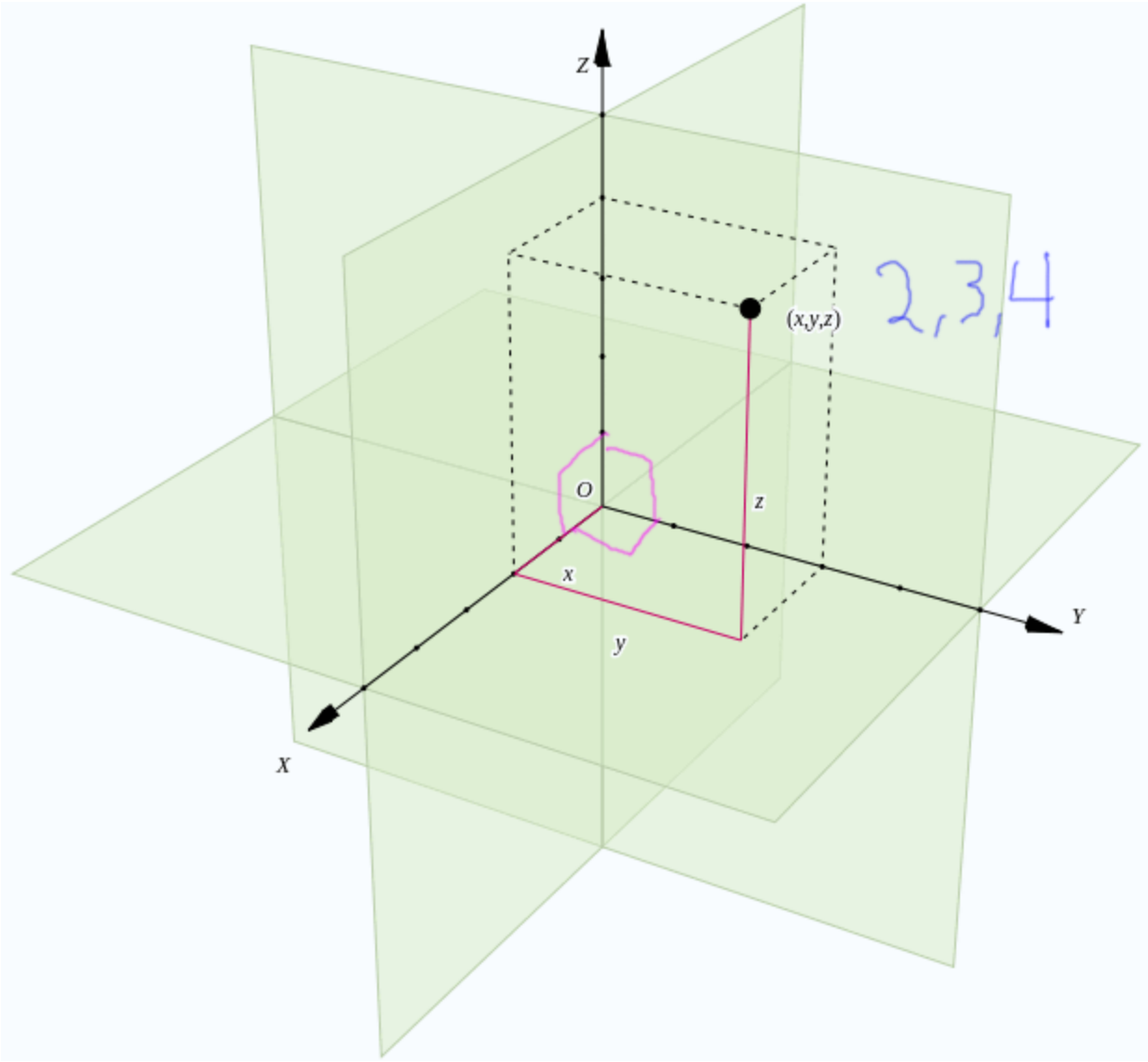
[draw angles as squared off, at origin]



It's possible to define a coordinate system with axes that are not perpendicular, but this is rarely done in practice. For example, finding the distance between two points becomes a tricky concept when the axes are not mutually perpendicular.

Given an origin and three vectors, we can then define the location of any point in space with a triplet of numbers. For example, this point has a triple of 2,3,4, as it is two units of length along X, three along Y, and 4 along Z.

[write (2, 3, 4)]



3D Vectors are defined by a similar coordinate reference system, except that the origin is not needed. A vector describes a motion, which again can be described by three numbers. That is, it describes how far to move to get from one point to another. However, this movement is not fixed in any particular location in space. To understand the difference, think about time. A specific time is like a point. An amount of time is like a vector: it specifies a duration, but no starting time is given.

[Instructor comments: you can read more about the Cartesian coordinate system here, http://en.wikipedia.org/wiki/Cartesian_coordinate_system#Cartesian_coordinates_in_three_dimensions. For a stronger mathematical presentation of the material in this lecture, see Gortler's "Foundations of 3D Computer Graphics", http://www.amazon.com/Foundations-Computer-Graphics-Steven-Gortler/dp/0262017350]

Question: Point or Vector?

In these statements, is the thing described with a number more like a point or a vector?

[visual cue: anything I write in blue will appear on the screen.]

Point Vector

() () - Let's meet at 8 am tomorrow, if that's not too early.
() () - The continental shelf is generally no deeper than 150 meters from sea level.
() () - This film is almost three hours long!
() () - The statue is 200 yards north of town center.

Answer

[point, vector, vector, point. Though the last one could almost be considered a vector - this is why I like the time analogy, because location and direction are pretty intertwined in our heads and timeline location and duration are not.]

This first answer gives a definite time, which is like a point on a timeline. So, "point" is the correct answer.

The second describes a distance, which is like a vector. No specific point at sea level is given, so this does not describe a point. So, this statement describes a vector.

This third sentence describes a duration and no specific time, so is like a vector.

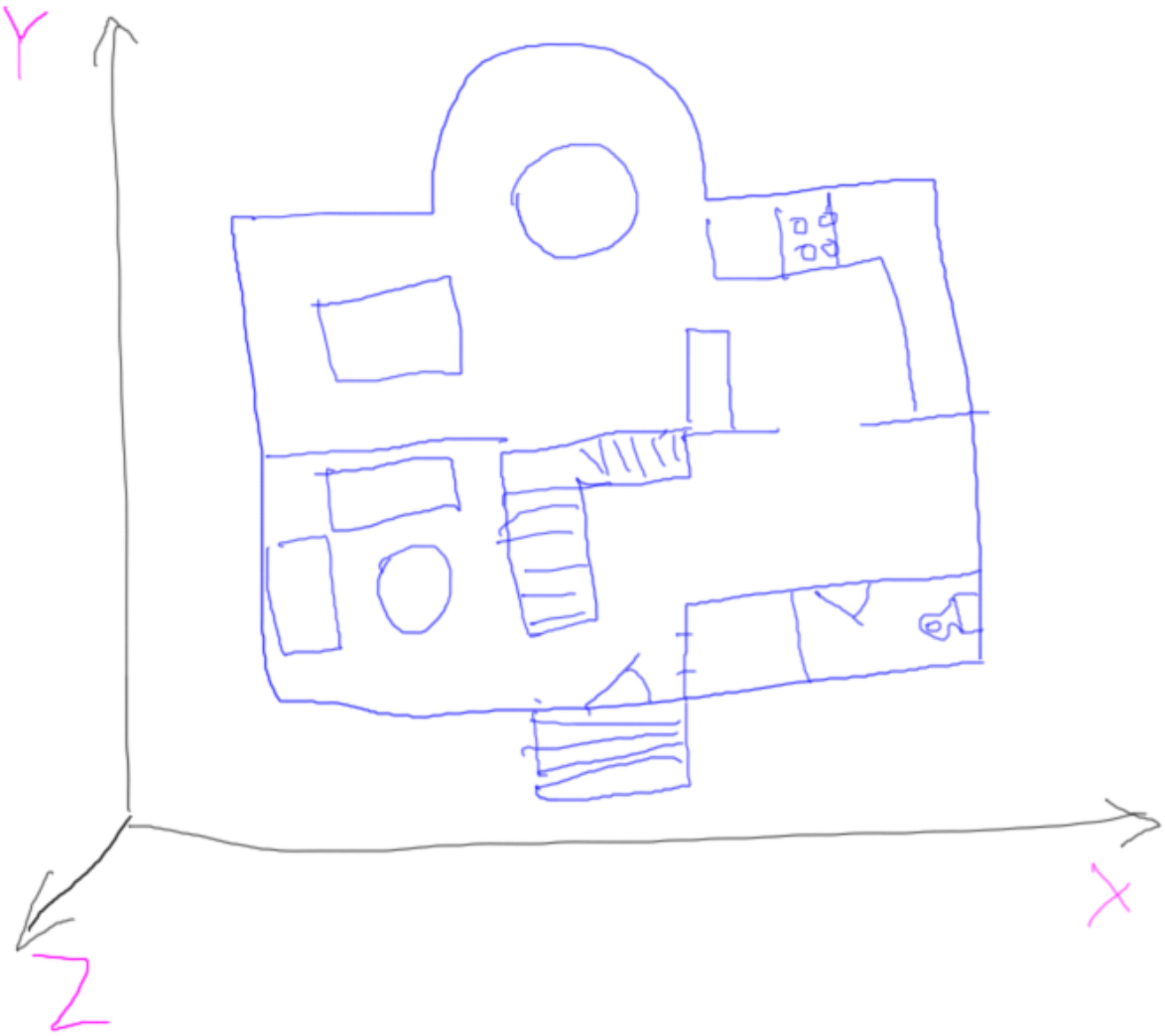
The fourth describes a location given another location and a motion vector, so is a specific point.

Lesson: Right-Handed vs. Left-Handed

Just like your right and left hands both have five fingers but look different, coordinate systems also have an orientation to them. When you define a set of axes, you also need to specify one other fact: whether the axes are in a right-handed or a left-handed system.

Here's an example of a right-handed coordinate system. Say I want to build a building. I define the X axis as going east and the Y axis as going north. Now my floor plans show X pointing right, Y pointing up. This is the standard 2D Cartesian coordinate system. If I define altitude as the Z axis, with the arrow coming out of the diagram towards the viewer, this is a right-handed

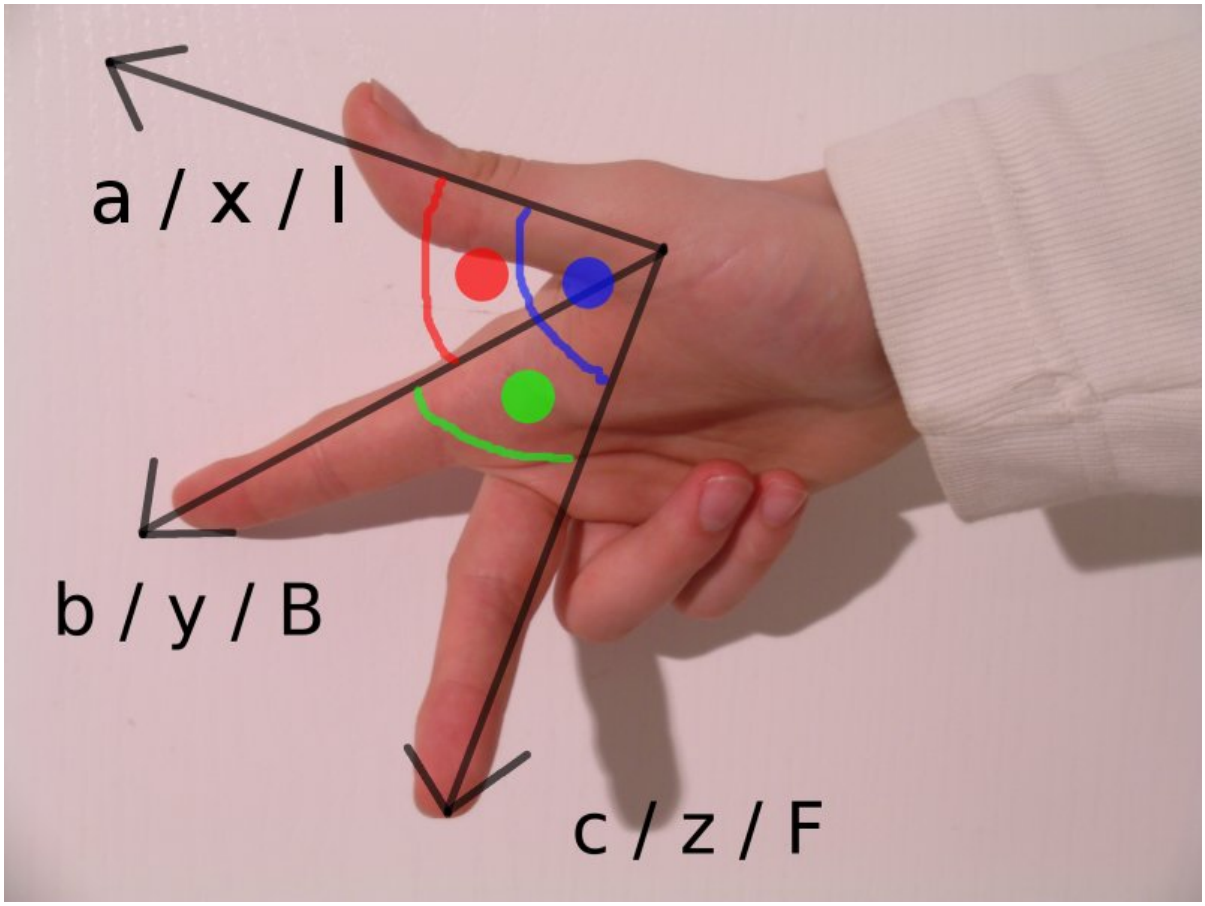
coordinate system.
[Picture drawn here of X and Y in a plane, with a floor plan]



Why is this right-handed? If you place your thumb along the X axis and the index finger along the Y axis, the middle finger is the Z axis and points towards you. You'll notice if you try to do this same thing with your left hand you can't match up all three axes, at least not comfortably!

[show my hand in the video, orient my fingers with the drawing above. Then switch to still image of both.]

[picture from http://en.wikipedia.org/wiki/Right-hand_rule - I can do better, I want the Z axis pointing up.]



[switch back to view of just the drawing - erase Z pointing up, add a Z axis pointing down.]

The key thing to realize is that a left-handed coordinate system is equally valid - it does not change the underlying reality of the situation. For example, intelligent rabbits might agree that X is east and Y is north. However, they might decide the Z axis points *down*, into the page. The rabbits cares more about how far down a tunnel is dug from ground level and want to say “5 feet” instead of “negative 5 feet”.

[Orient left hand so that it aligns with the axes.]

This is a left-handed coordinate system. The underlying reality is the same in both cases, nothing has changed place, just how we and the rabbits refer to a location differs. When we say 12 meters, the rabbits would say negative 12 meters, and vice versa. To convert between these particular two systems is simply a matter of negating the Z coordinate value.

[Additional course materials: This is a long-standing question, whether to use left-handed or right-handed coordinates and when. You can see some of the ancient debate here: http://steve.hollasch.net/cgindex/math/matrix/column-vec.html]

Question: Right or Left?

[Is longitude/latitude/altitude right-handed or left-handed?]

Is longitude/latitude/altitude right- or left-handed?

* Consider East to increase the longitude
* Consider North to increase the latitude

Order is also important: I list longitude first here, since we tend to think of that direction as the X direction. In reality, locations are usually listed as latitude first, then longitude.

( ) long./lat./alt. is right-handed
( ) long./lat./alt. is left-handed

Answer

[Put the lines and axis arrows in a separate layer, to reuse later in answer]

The X axis is west, the Y axis is north, the Z axis is up, so the answer is “right-handed”.

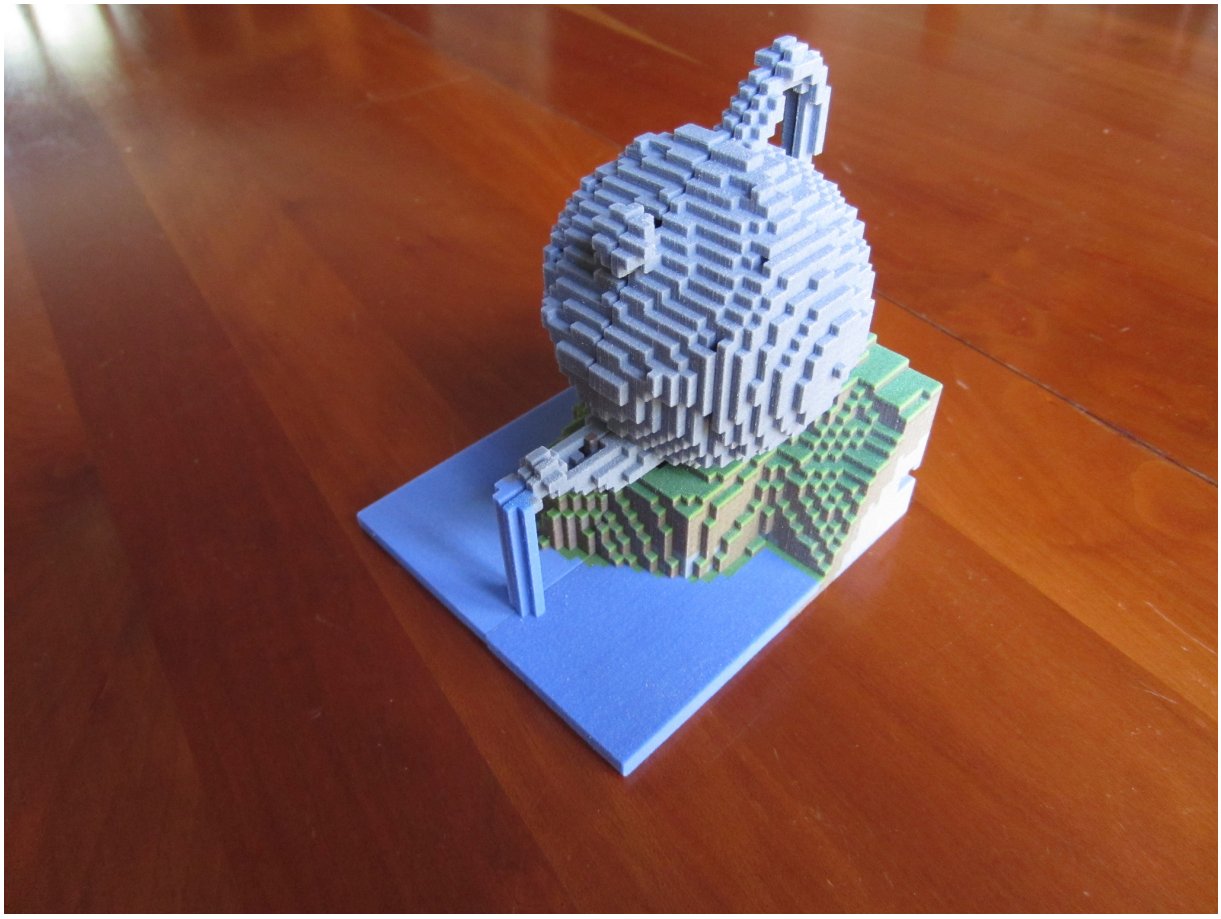
What’s interesting is that if you swap longitude and latitude and so list the coordinates as latitude, longitude, and altitude, the coordinate system is now left-handed. It also swaps if you consider west to increase the longitude or South to increase the latitude.

Question: The Sun Also Rises

I happen to love the game Minecraft, along with a few other million people. Yes, I built a teapot in the game.

I even wrote a program to help me 3D print models in Minecraft.

[switch between this and this]



One confusing thing in the game is the coordinate system. In the beta version of Minecraft the coordinate system was as follows: the X axis points south, the Z axis points west, the Y axis points up.

[write this out]
[Beta: The X axis points south, the Z axis points west, the Y axis points up.]

You could tell these were the axis directions because in the game the sun rose in the east.

When the game was released, the developers changed the direction the sun moved. The sun started rising in what we used to think was the south but was now considered the east. In other words, the X axis now points east, the Z axis points south, the Y axis points up.

[Release: The X axis now points east, the Z axis points south, the Y axis points up.]

So, what handedness are the beta and the release coordinate systems?

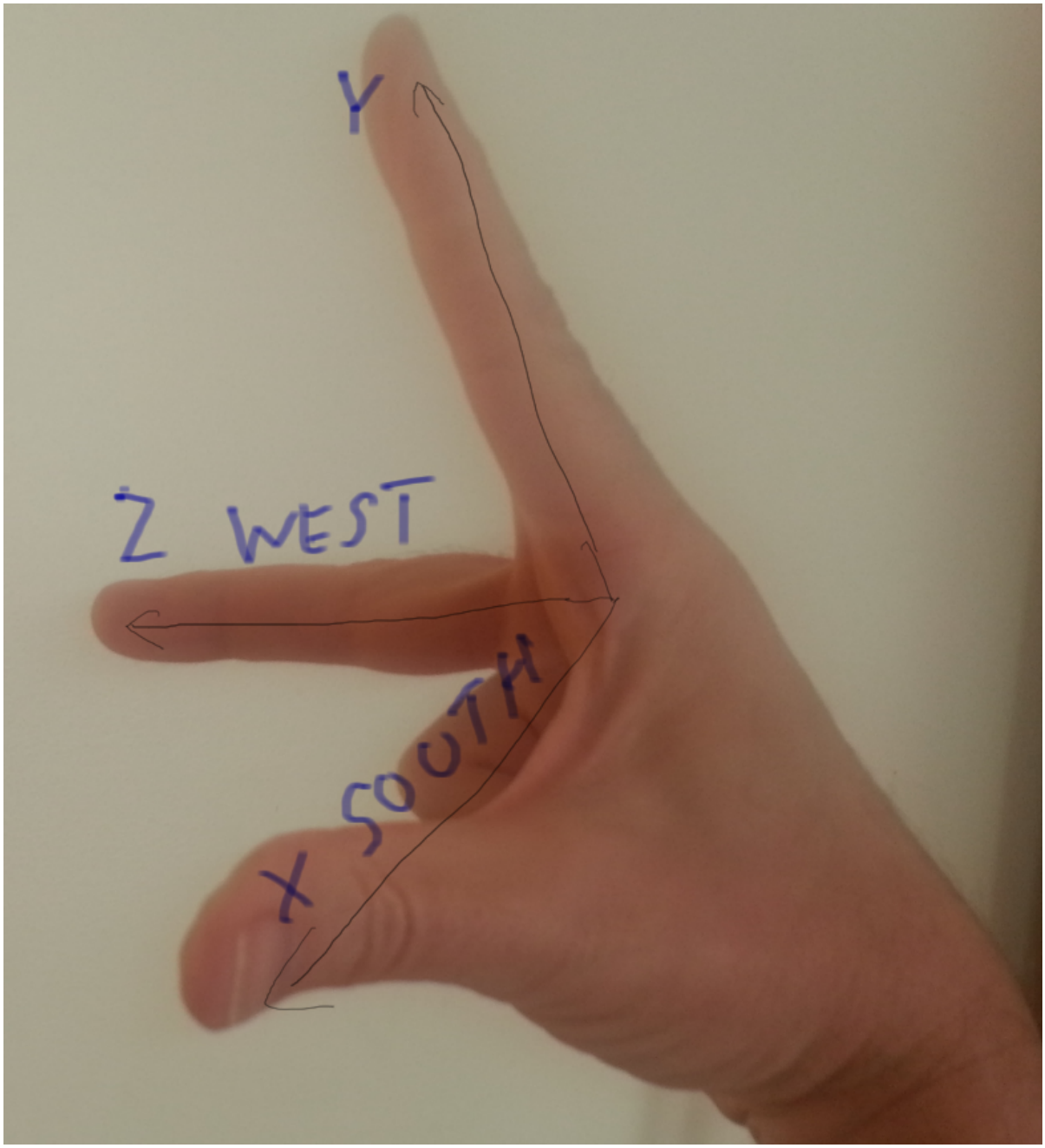
[] Beta is right-handed, release is right-handed
[] Beta is left-handed, release is left-handed

[] Beta is right-handed, release is left-handed
[] Beta is left-handed, release is right-handed

Answer

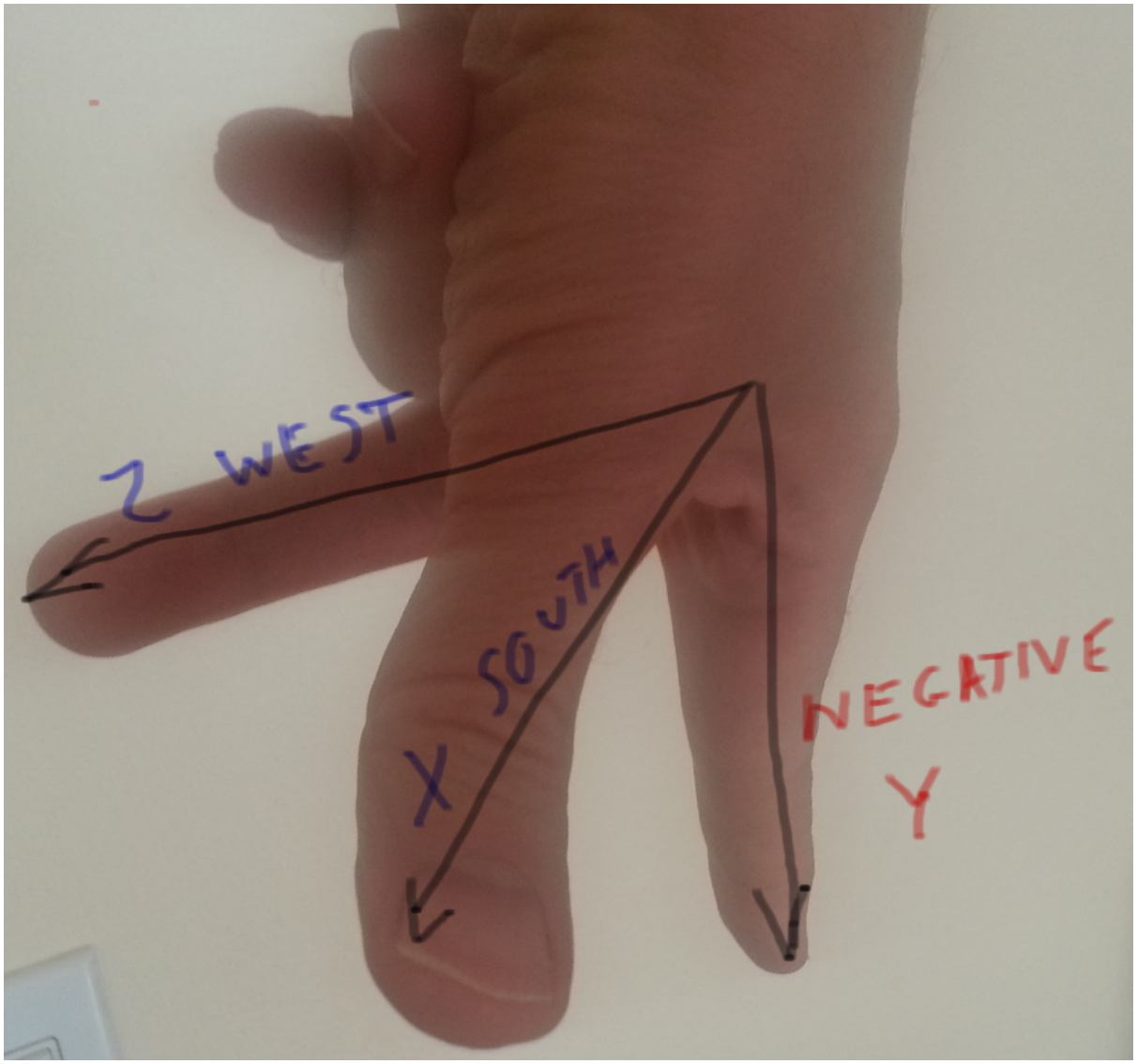
The way I solved this puzzle was first to align my right hand to the directions given in the beta: X south, Z west, Y up. If I could do it, then I'd know the system was right-handed. Easy enough, my right-hand aligns just fine.

[show right hand picture with axes labeled]



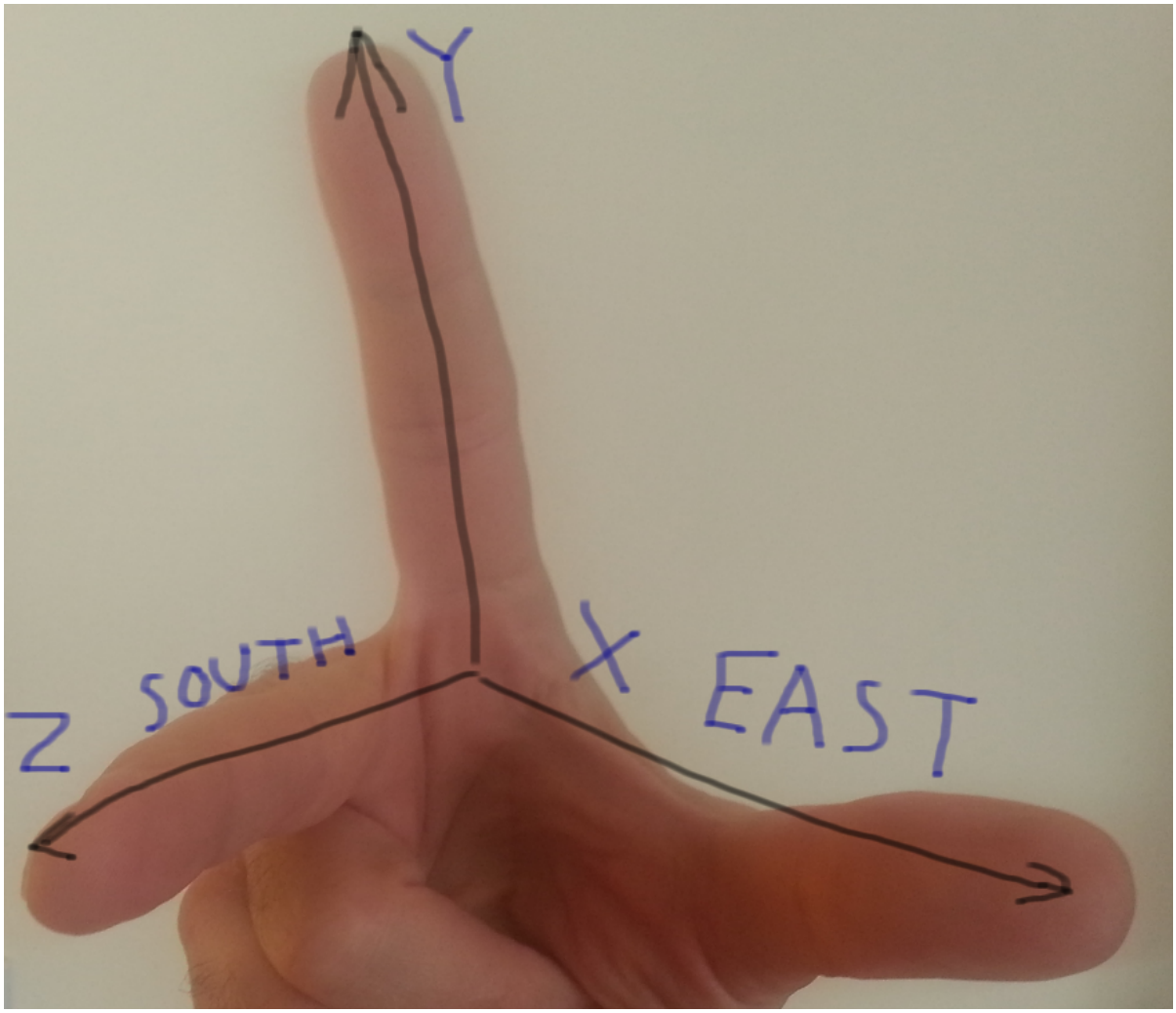
If I tried this same thing with my left hand, I can get two out of three axes to align, but the third is always in the opposite direction.

[show left hand]



So we know the beta version of Minecraft used right-handed coordinates. I could solve the second half of the question by reorienting my right hand and again noticing that the fingers can properly align with the axes, so it is also right-handed.

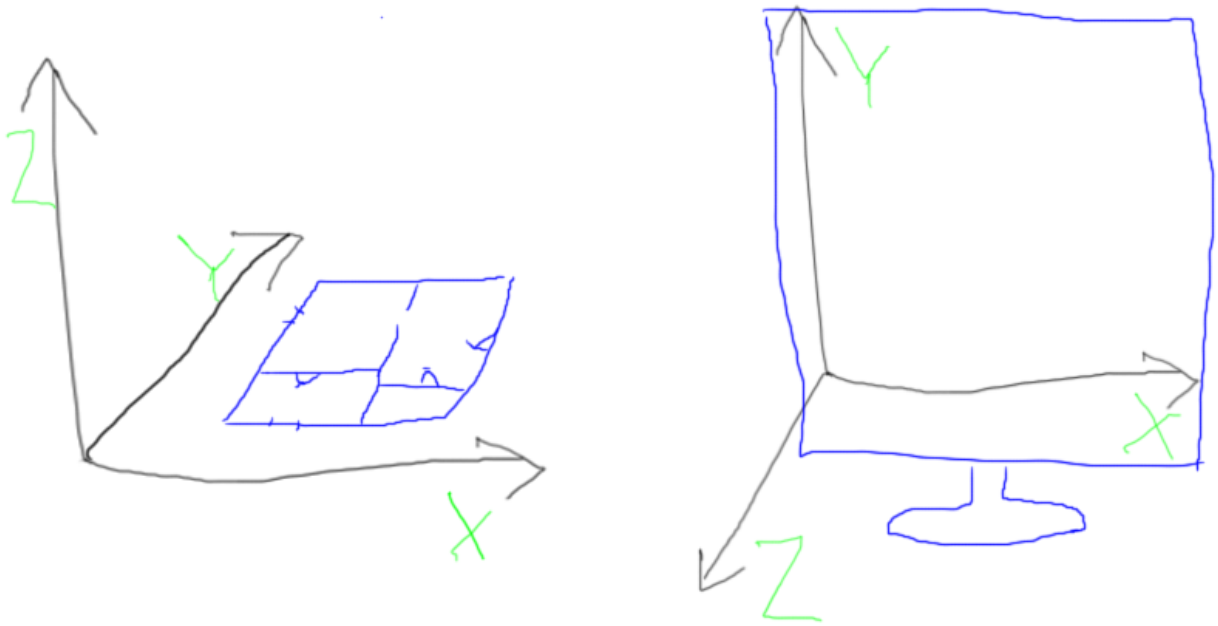
[show reoriented right hand]



However, you don't really need to use your hands to this out. What happened within the world itself was simply that the sun started to rise from a different direction. None of the world's data had changed locations at all. The only thing that changed was our perception of which direction was East. Since the world was the same as before, the underlying coordinate system itself had not changed. Realizing this, I could conclude that the released version is still right-handed, just like the Beta. But, it never hurts to check with my right hand.

Lesson: Which Way is Up?

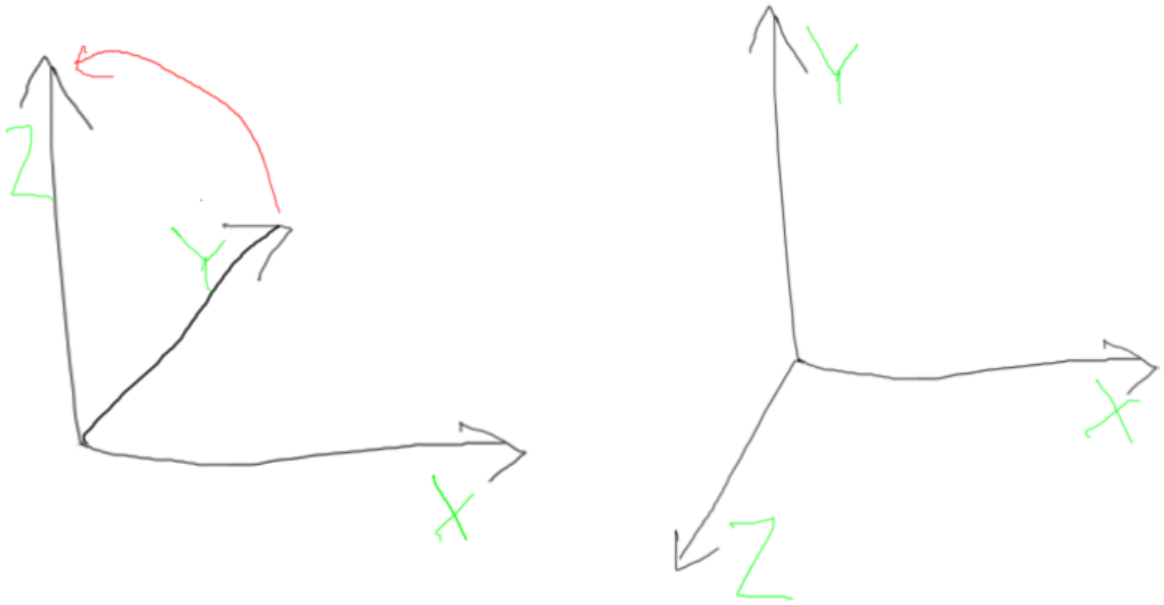
There is no correct answer for which is better, a left-handed or right-handed coordinate system. One or another may be more convenient for people or intelligent rabbits, depending on the situation. One other area in computer graphics where opinions vary is whether the Y axis or the Z axis should be the "up" direction.
[Draw X Y Z-up and X Y-up Z -]



Architects tend to think in terms of floor plans. So, the X and Y axes are directions on the ground, and Z is then the height.

However, another common way to think about the coordinate system is with respect to our display. Looking at a monitor and starting in the lower left corner, the X axis goes to the right, the Y axis goes up. If we want to present a right-handed coordinate system with respect to the monitor, the Z axis then points towards us. We'll see this type of definition in a later unit, when we cover how to define a camera.

The difference between these two coordinate systems is just a ninety degree rotation along the X axis, so is easy enough to fix. We'll talk about rotations in a later unit.
[show rotation]



However, now you've been warned: if you read in some model someday into your program and the object is sideways, it's probably because the model's up direction is different than what you're expecting.

Question: Coordinate System Transform

To work out your brain a little, say we have two coordinate systems with the same origin. The first uses the architect's scheme, X goes east, Y goes north, Z goes up. The second uses Minecraft's current system of X goes east, Y goes up, Z goes south.

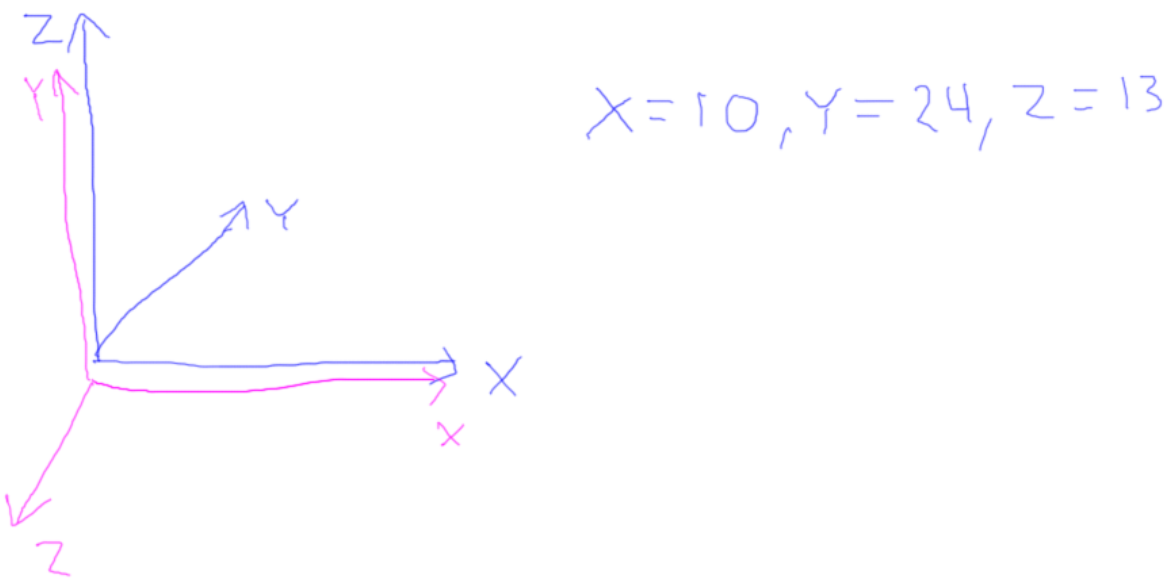
If I have a point (10,24,13) in the architect's system, what is this point's location in the Minecraft system?

X = [ ], Y = [ ], Z = [ ]

[I think it's best to write this exercise out, not showing any coordinate systems - it's part of the question for the student to draw a coordinate system, if needed.]

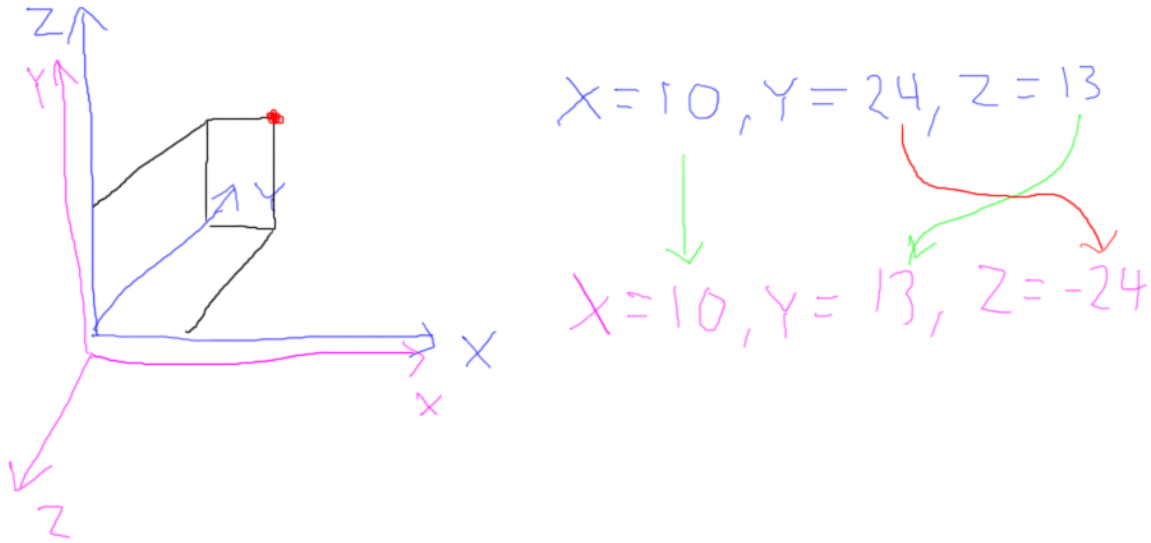
Answer

Here are both coordinate systems, with the architect's scheme in blue, Minecraft in purple.
[coordinate systems]



We're in luck with the X axes. In both systems we know the X axes match. Whatever happens with the Y and Z axes won't affect the X axis value. So in the second system X will also be 10.

[show the two coordinate systems, and a point X = 10, Y = 24, Z = 13 followed by a line X = , Y = , Z = , drawing as we go, showing the coordinates copy into these blanks. Need to fix in final: Y axis blue label got obscured by the black lines.]



In the first system Z goes up, and in the second Y goes up. That's simple enough: the height of the point in the first system was 13, the height is still 13 in the second system, so Y will be 13.

Finally, in the first system Y goes north, Z goes south. The Y coordinate in the first system was

24. Since Z is going the opposite direction, Z must then be -24.

So, the answer is (10,13,-24).

Lesson: Points and Lines

There are only three basic primitives we ever send to the Graphics Processing Unit: points, line segments, and triangles. Of these, triangles are by far the most important. For example, just about everything you see in a 3D game is ultimately made of triangles.

Sometimes it's obvious where the triangles are used, such as in Minecraft. Yes, another teapot:

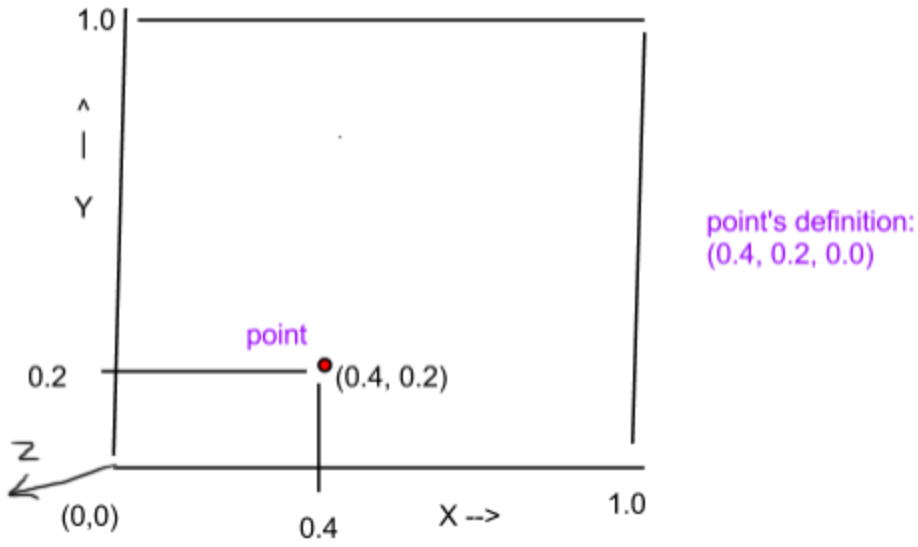
[These will be videos. Could freeze-frame and highlight a triangle with the pen.]

Sometimes it's not, as in this shot from Battlefield 3, captured by Duncan Harris at Dead End Thrills:.

[http://deadendthrills.com/?cat=1596 Chariot of Fire]

We're going to touch on points and lines only briefly so we can focus on triangles. Defining a point is simple, you just specify a **3D** location:

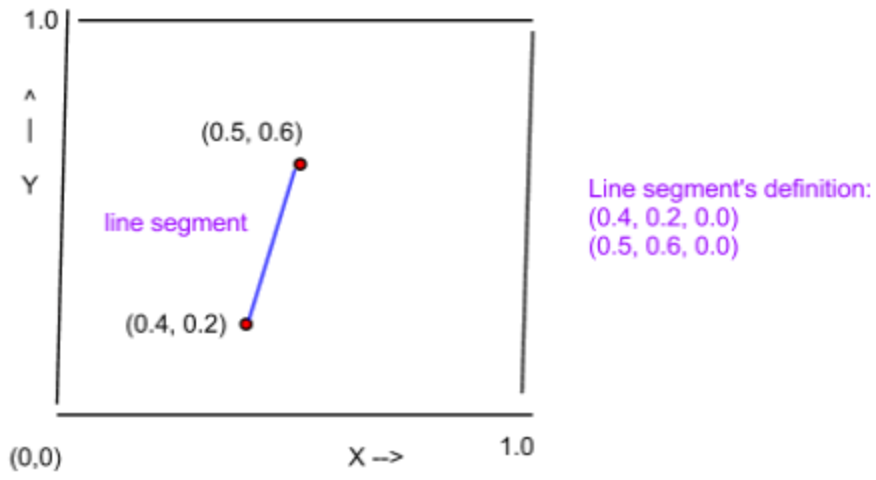
[point - note: numbers should be in blue everywhere- NOTE 0.0 should be put for Z here. SAVE BASIC GRAPH OUTLINE IN A SEPARATE LAYER FOR REUSE!]



By the way, for all these examples I'm going to make the Z value 0, but it can certainly have a different value. In 3D computer graphics you tend to set all three coordinates - X, Y, and Z - even

if you're mostly working in 2D.

A line segment is specified as a pair of two points:
[in reality we'll just add to the existing figure above - colors need changing, and Z-axis added.]



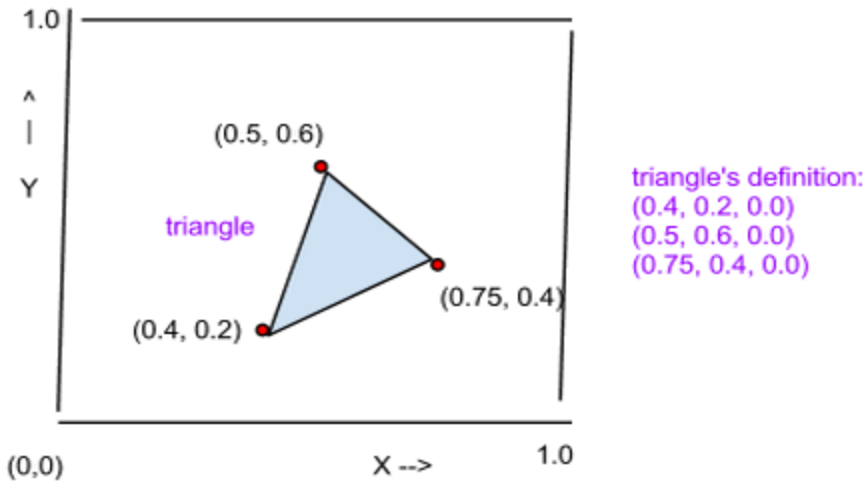
The order of the points does not matter.

[Additional course materials: For some stunning screen shots of what can be done at interactive rates (or nearly so), see Dead End Thrills http://deadendthrills.com. The second screenshot is from http://deadendthrills.com/?cat=1596; many thanks to the site's creator, Duncan Harris.]

Lesson: Triangles

A triangle is specified by three points, one two and three:

[again, add in Z axis up.]



We've progressed from a zero-dimensional point, to a one-dimensional line segment, to a two-dimensional triangle. That is, even though the coordinates of each object are specified in three dimensions, the objects themselves have these lower dimensions. Two-dimensional surfaces are usually as far as we need to go. We normally care about how light bounces off an object's surface, we don't care all that much about representing a solid volume. That is, we don't need to represent what is inside the object in any detail.

[Optional: for an exercise we could repurpose ch2_RenderingModes.html from the Web Beginner's Guide, see http://voxelent.com/html/beginners-guide/1727_02/ch2_RenderingModes.html . To be honest, I don't think it's worth doing - points and lines are rarely used, so I would rather mention them here in passing and move on.]

Exercise: Draw a Square

In the code in this exercise we give you a ground plane grid to view. In three.js you define a Mesh object by providing a Geometry object and a material. The Geometry object consists of vertices, along with a face. "Vertices" are the corners of a polygon, usually a triangle. Here's sample code:

```
var material, geometry, mesh;
material = new THREE.MeshBasicMaterial( { color: 0xff0000, side: THREE.DoubleSide }
);

var geometry = new THREE.Geometry();
geometry.vertices.push( new THREE.Vector3( 5, 10, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 5, 5, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 10, 5, 0 ) );   // vertex 2
```

```
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

var mesh = new THREE.Mesh( geometry, material );

scene.add( mesh );
```

The first three lines define the three points we'll use to make a triangle, putting them into an array. Note that in three.js there are no separate classes for points versus vectors: Vector3 is used for both, consisting of 3 values.

The face definition here says which of the vertices to use to make the triangle, namely the vertex 0, 1, and 2.

[draw arrows from face index to vertex]

```
var material, geometry, mesh;
material = new THREE.MeshBasicMaterial( { color: 0xff0000, side: THREE.DoubleSide }
);

var geometry = new THREE.Geometry();
geometry.vertices.push( new THREE.Vector3( 5, 10, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 5, 5, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 10, 5, 0 ) );   // vertex 2

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

var mesh = new THREE.Mesh( geometry, material );

scene.add( mesh );
```

The mesh is then defined and added to the scene, which can then be rendered.

Your job is to make a square on the screen, going from location 3,5 to 7,9

[
X, Y
3, 5 lower-left corner
7, 9 upper-right corner]

Use only one mesh object - no fair simply making two separate meshes, each with a triangle.

[The exercise is here: http://www.realtimerendering.com/udacity/?load=unit2/triangle-mesh.js]

[exercise would be graded by whether the square is drawn correctly.]

[In actuality, the solution the student gives here could be checked further: if the geometry vertex array has more than four vertices, a second exercise could be given, namely “You were able to draw the square, which is great. Your next challenge is to do this exercise again and try to draw the square by defining just four points.”]

[Additional course materials: You probably won’t need it for this exercise, but you will eventually. There are JavaScript debuggers for each browser out there. Chrome and Safari’s debuggers are built in, Firefox uses Firebug. Read about how to use them on the following pages - a debugger can be invaluable for examining variables, or seeing what parameters are available in a class and what their class types are. The Chrome debugger is described here http://developer.chrome.com/extensions/tut_debugging.html, Firefox’s Firebug debugger here http://getfirebug.com/javascript, Safari here http://petewarden.typepad.com/searchbrowser/2008/07/how-to-debug-ja.html and then here http://developer.apple.com/library/safari/#documentation/appleapplications/Conceptual/Safari_Developer_Guide/DebuggingYourWebsite/DebuggingYourWebsite.html#//apple_ref/doc/uid/TP40007874-CH8-SW1]

Answer: Draw a Square

There are many ways to specify a square.

[Solution code is *redacted*]

It turns out that the three.js library happens to support a four-point face, so you could just use one face:

```
geometry.faces.push( new THREE.Face4( 0, 1, 2, 3 ) );
```

That said, the way that WebGL, DirectX, and other low-level APIs work is to define only triangles. When the three.js library defines a four-sided polygon in this way, the quadrilateral is rendered by two triangles being sent to the GPU.

If you happened to solve this exercise by drawing a single quadrilateral, that’s great that you read the three.js documentation - bonus points for you! - but it’s worth your while to try this exercise again using just triangles.

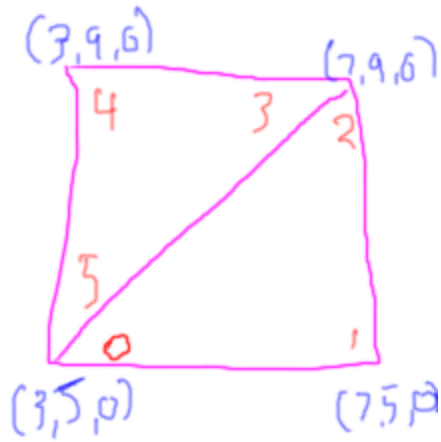
One solution with triangles is a bit brute force:

```
// first triangle
```

```
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 7, 5, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );   // vertex 2

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

// second triangle
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );  // vertex 3
geometry.vertices.push( new THREE.Vector3( 3, 9, 0 ) );   // vertex 4
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );   // vertex 5

geometry.faces.push( new THREE.Face3( 3, 4, 5 ) );
```

Two entirely separate triangles are created, here and here.
[draw on screen and label coordinates 0 through 5]

```
// first triangle
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 7, 5, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );   // vertex 2

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

// second triangle
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );  // vertex 3
geometry.vertices.push( new THREE.Vector3( 3, 9, 0 ) );   // vertex 4
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );   // vertex 5

geometry.faces.push( new THREE.Face3( 3, 4, 5 ) );
```

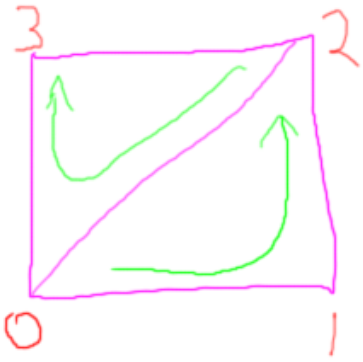


This works, but there's duplication of triangle indices here - vertex 2 and 3 are the same, as are vertex 0 and 5. A more compact and efficient way to draw the square is to define the four vertices needed and then define two triangle faces:

```
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 7, 5, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );   // vertex 2
geometry.vertices.push( new THREE.Vector3( 3, 9, 0 ) );   // vertex 3

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );
geometry.faces.push( new THREE.Face3( 2, 0, 3 ) );
```

[indexed version - then show loops]

```
geometry.vertices.push( new THREE.Vector3( 3, 5, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 7, 5, 0 ) );  // vertex 1
geometry.vertices.push( new THREE.Vector3( 7, 9, 0 ) );  // vertex 2
geometry.vertices.push( new THREE.Vector3( 3, 9, 0 ) );  // vertex 3

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );
geometry.faces.push( new THREE.Face3( 2, 0, 3 ) );
```

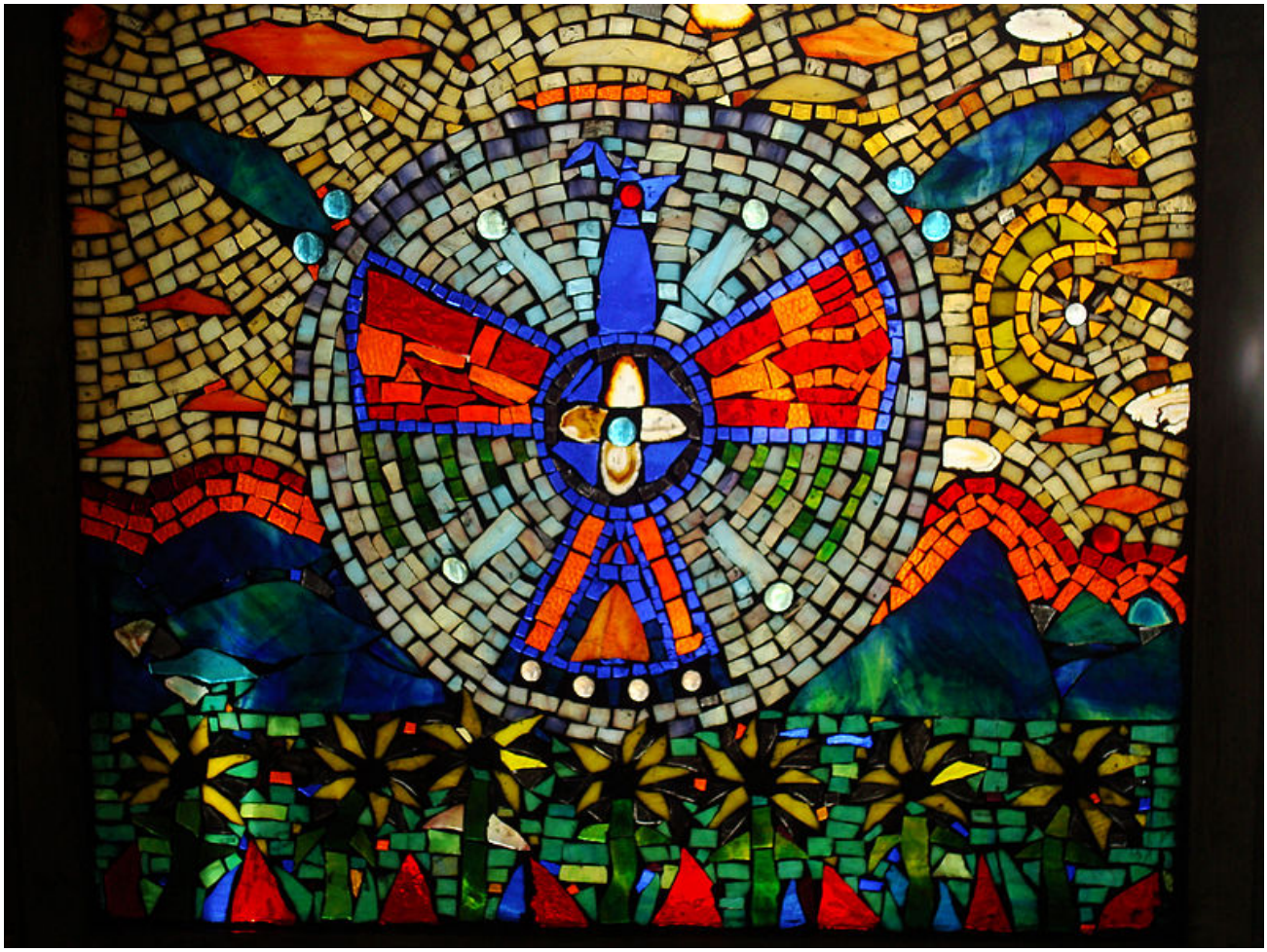


[Note I purposely reverse the order of the second triangle, as we'll use this next exercise.]

Lesson: Triangulation and Tessellation

"Tessellate" is the most misspelled word in computer graphics, so give it a good look and think of doubling every letter you can when you spell it. Yes, two "L's" looks wrong to me, too, but that's how it works, and I've forced myself to spell it correctly. The word "tessellate" is from the Greek word "tessella", which means a small stone in a mosaic.

[Draw word Tessellate above image and then outline a stone - http://commons.wikimedia.org/wiki/File:Nunnally_Victor_8.jpg]



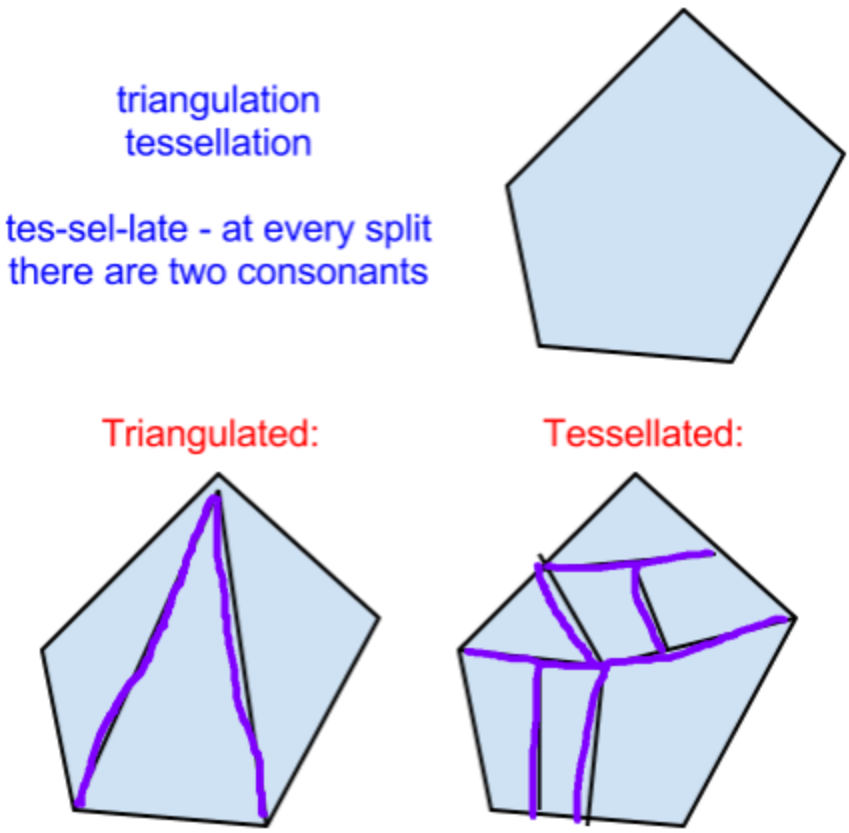
In English, breaking an object into polygons of any sort is called tessellation.

A curved surface is represented by triangles by placing points on its surface and connecting these together. For example, a geodesic dome is one tessellation of a sphere, such as the Biosphere in Montreal.

[http://commons.wikimedia.org/wiki/File:Biosph%C3%A8re_Montr%C3%A9al.jpg - may want to zoom in here, to see the triangles better.]

When it comes to making surfaces, the GPU understands only triangles. To draw any polygon, we break that polygon into a set of triangles. This process is called *triangulation.* Usually we use the existing vertices of the polygon when performing triangulation. The major rule is that when we add an edge, it has to be entirely inside the polygon.

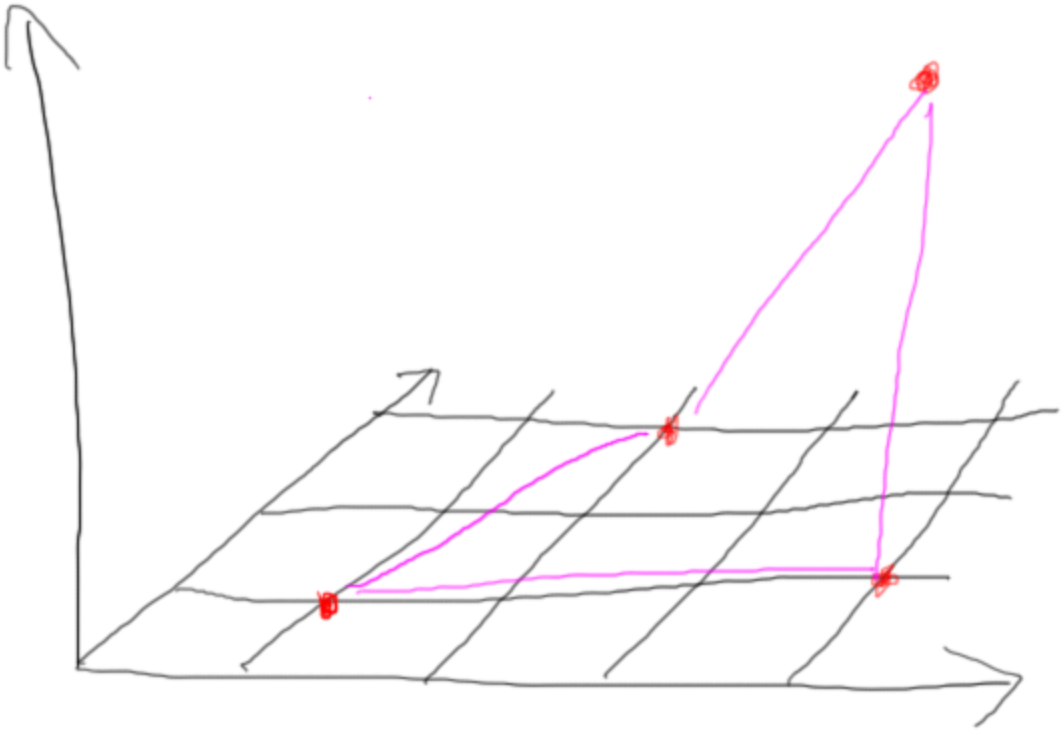
Triangulation is a particular form of tessellation. Here's the same pentagon tessellated in a fairly arbitrary way, with new vertices added inside the pentagon and connected together.



Once upon a time some older graphics hardware from the 1980's supported objects such as polygons or even more complex objects. For example, NVIDIA's first chip, the NV1, supported ellipsoids as a basic primitive. Graphics hardware manufacturers quickly came to realize that the triangle is a fine building block for almost everything we want to draw. For example, one problem with a quadrilateral is that the surface is poorly defined if all four points are not in the same plane. A triangle's three points are, by definition, always in the same plane and so have no such problems.

By the way, these were sweet machines: the graphics coprocessor I used to develop on in the late 1980's was from Hewlett Packard and cost about $35,000 at the time. It was perhaps at best one one-millionth as fast as current GPUs, at only the price of a BMW.

[draw 3D coordinate system, three points in plane and a fourth above it.]



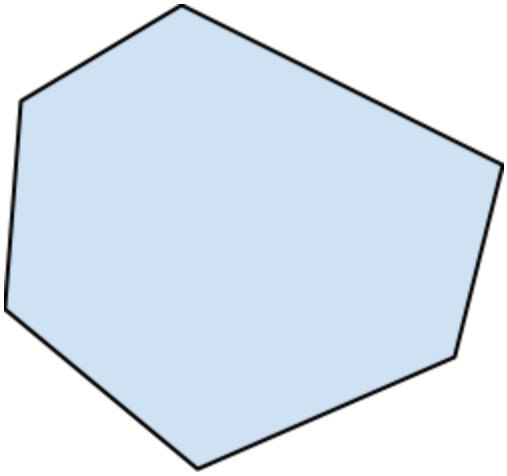
[Instructor Comments:

Give Wikimedia Commons image info
http://commons.wikimedia.org/wiki/File:Nunnally_Victor_8.jpg]

If you can spell “frustum” and “tessellation” correctly, you’ve avoided about 95% of the spelling errors you can make in the field of computer graphics, see http://www.realtimerendering.com/blog/do-you-spell-these-two-words-correctly/. About the only other error I recall seeing is “composting” instead of “compositing”.]

Quiz: Minimum Triangulation

Using triangulation or tessellation, what is the minimum number of triangles you can use to represent this hexagon?

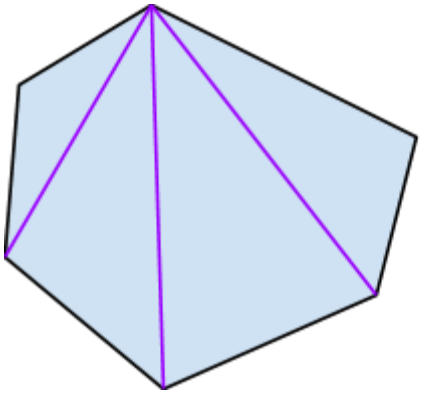


[ ] triangles are the minimum needed.

Answer

General tessellation won't help - adding vertices will just need more triangles to join these up.

The answer is four triangles. In fact, the rule is that a polygon with N edges will need N-2 triangles to represent it.



Polygon with N edges,
you need N-2 triangles to
triangulate

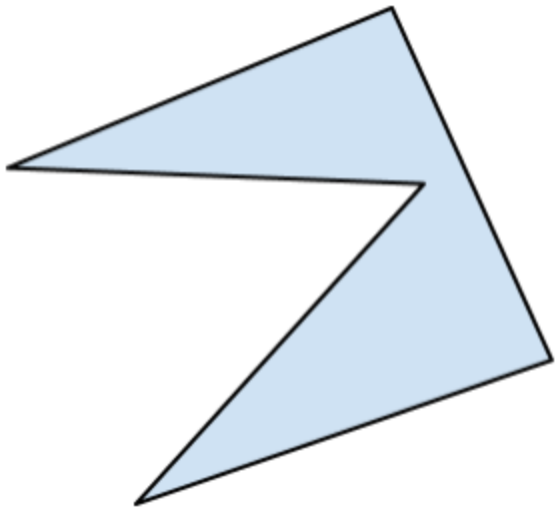
We don't care all that much in interactive rendering *which* triangulation is used. It's important in other fields, such as stress analysis programs that use meshes, but isn't so important in interactive rendering.

[Udacity: we could add a quiz here that says something like "given a hexagon, how many distinct triangulations can be formed?" The answer's lessee 12, I think, but I'm not sure there's any point in making someone go through figuring these out. We don't really care all that much in interactive rendering *which* triangulation is used - it's important in other fields, like finite element analysis, but not here.]

If you're a puzzle buff, here's a bonus question for absolutely no credit and little use in computer graphics itself, but rather just for the sake of it: how many different unique triangulations of this hexagon are there? Once you solve that, is there a general rule for polygons? Feel free to discuss this on the forums.

Quiz: How Many Triangulations?

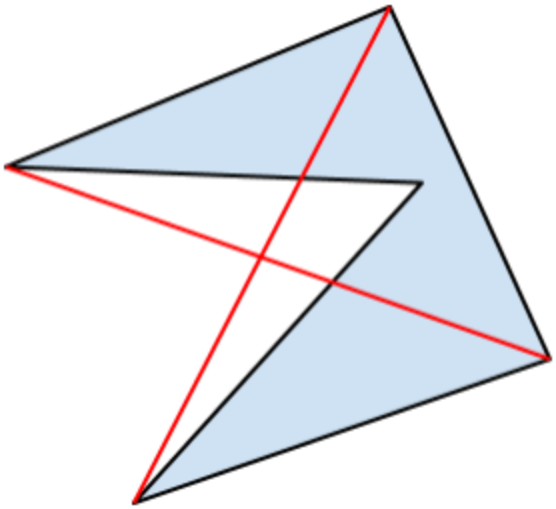
Given this pentagon, how many ways can we triangulate it, without adding new vertices?



[ ] How many ways can we triangulate this pentagon?

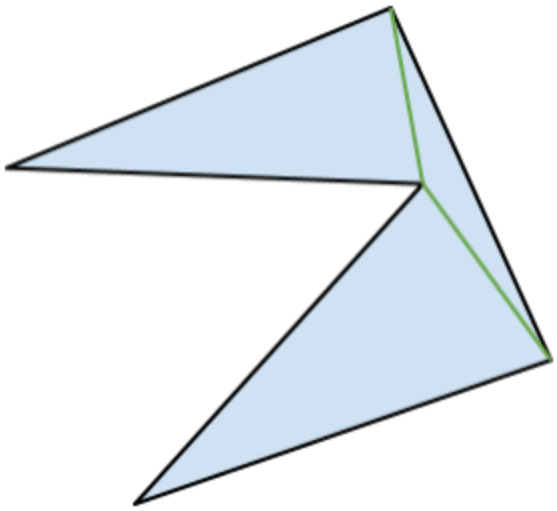
Answer

Surprisingly, there is only one way to triangulate this figure. Because of the deep concavity, we can't add edges here and here.
[show bad edges]



Instead, the only places to add edges are here and here, giving the only possible triangulation.

[solution]

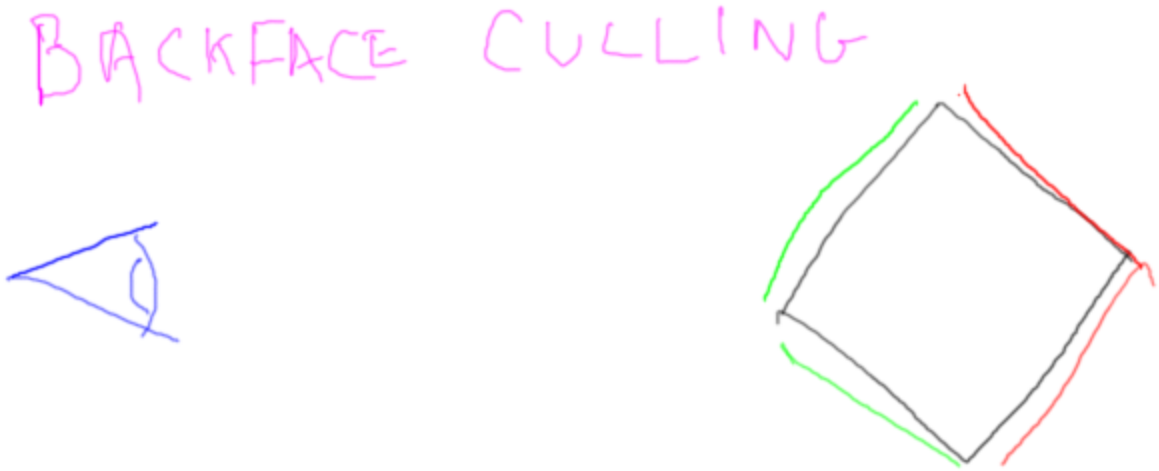


The good news is that, no matter how complex the form, any polygon can be triangulated into triangles, so can be displayed using a GPU.

Lesson: Vertex Ordering and Culling

3D computer graphics uses an interesting concept to speed up object display, called *back-face culling*. Imagine you are looking at a box. Here's the side view in 2D:

[eye with box from side - phrase *back-face culling* at top]



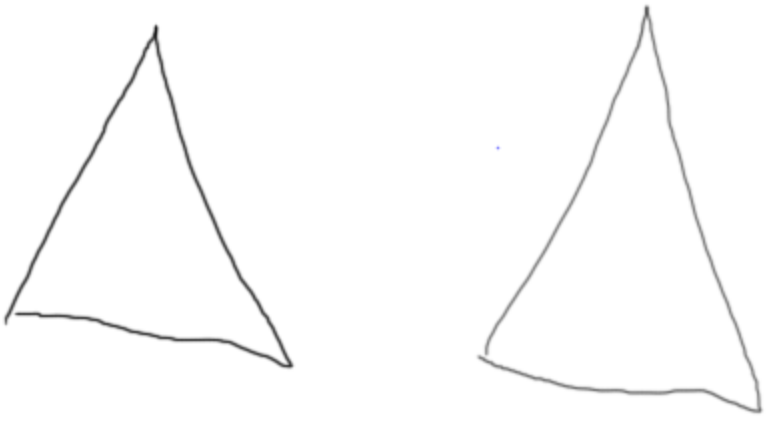
Only the sides of the box that face toward the camera are visible. The backfaces here and here do not need to be rendered. The fastest polygon to render is the one you don't have to render at all. Culling can throw away about half the faces in an object, so is a worthwhile optimization.

Backfaces can be culled for any solid object: this V shape seen from the side has only two sides facing towards the camera. These other two faces can be thrown away without further processing. It turns out this face here is hidden from view by this front face, but we let the z-buffer take care of that.

[eye with V from side]

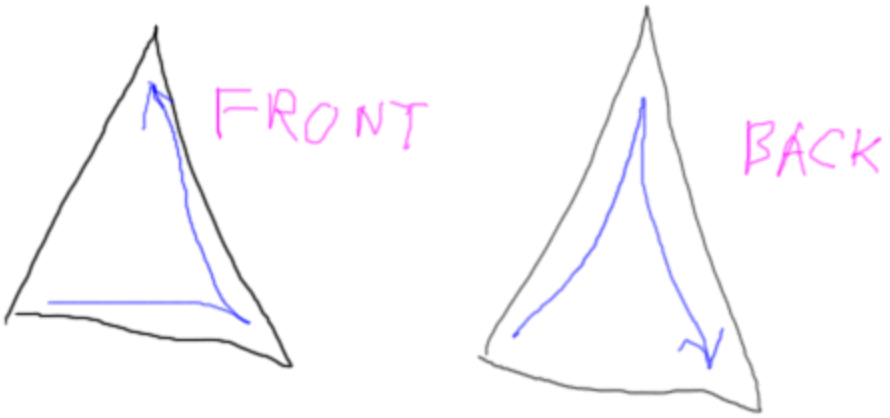
We can set whether backface culling is enabled or disabled. But, how do we determine if a triangle is a front face or a back face? There are just three points defining the triangle, and no flag is stored for additional information.

[Draw two triangles]



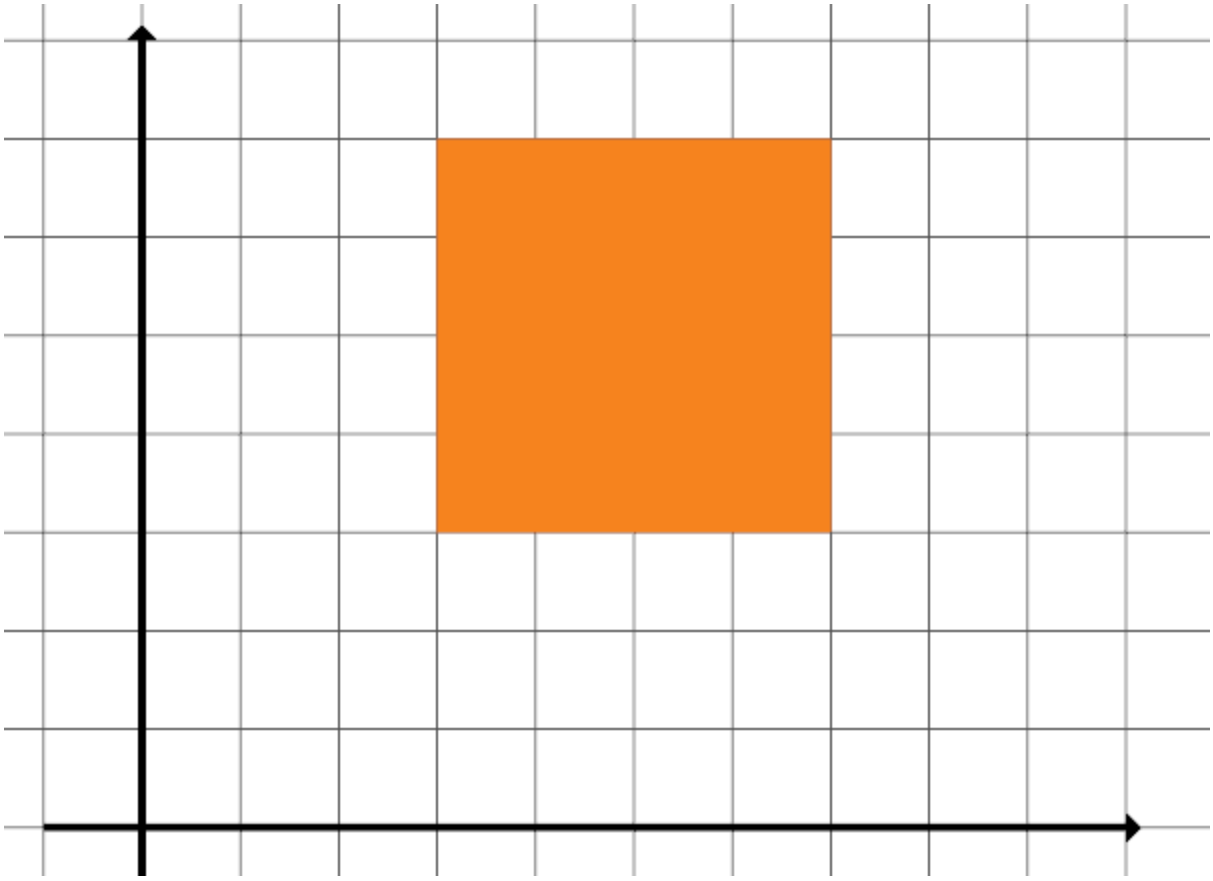
What determines whether a triangle is front or back facing is the order of the vertices after they're projected onto the screen. In WebGL, if the order is counterclockwise, the triangle is front facing, if clockwise, it's back facing.

[Two triangles with order of vertices shown]



Going counterclockwise like this is often called the right-hand rule. The fingers of your right hand wrap around the edges of the triangle in order. If your thumb points at you, the triangle is then front facing.

Exercise: Fix the Vertex Order



This is a fairly simple exercise. You're given a program that draws two triangles to make a square, similar to how this was done earlier. Backface culling has been turned on, which has caused one triangle to disappear. Determine the problem and fix the triangle's representation so that both triangles appear, forming a square:

```
function someObject () {
    var material, geometry, mesh;
    material = new THREE.MeshBasicMaterial( {
        color: 0xF6831E, side: THREE.FrontSide
    } );
    geometry = new THREE.Geometry();

    // Student: some data below must be fixed
    // for both triangles to appear !
    geometry.vertices.push( new THREE.Vector3( 3, 3, 0 ) );
    geometry.vertices.push( new THREE.Vector3( 7, 3, 0 ) );
    geometry.vertices.push( new THREE.Vector3( 7, 7, 0 ) );
    geometry.vertices.push( new THREE.Vector3( 3, 7, 0 ) );

    geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );
    geometry.faces.push( new THREE.Face3( 2, 0, 3 ) );

    mesh = new THREE.Mesh( geometry, material );

    scene.add( mesh );
}
```

Here's the method you need to fix. You can ignore the material calls at the top and the mesh and scene calls at the bottom. The problem has to do with the geometry calls in the middle.

By the way, in three.js backface culling is turned on by setting the "side" variable in the material. So here in the material creator we set "side" to FrontSide, which means that only the front faces are vislble. By default, culling is off until you set this "side" parameter.

[Exercise initial code is here:
http://www.realtimerendering.com/udacity/?load=unit2/vertex-order.js]

[Basically takes the two triangle program above, turn on culling (note how this is done in the code, using a comment). The fix will be to reverse the vertex order of the second face.]

[Gundega: you can additionally test if the student is cheating by rendering the scene with the camera location reversed - there should be *no* triangles visible when viewed from the other side.

Answer

[One good solution is here: *redacted*]

```
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );
// the order of the face is flipped (was 2, 0, 3):
geometry.faces.push( new THREE.Face3( 2, 3, 0 ) );
```

The vertices themselves are not the problem.

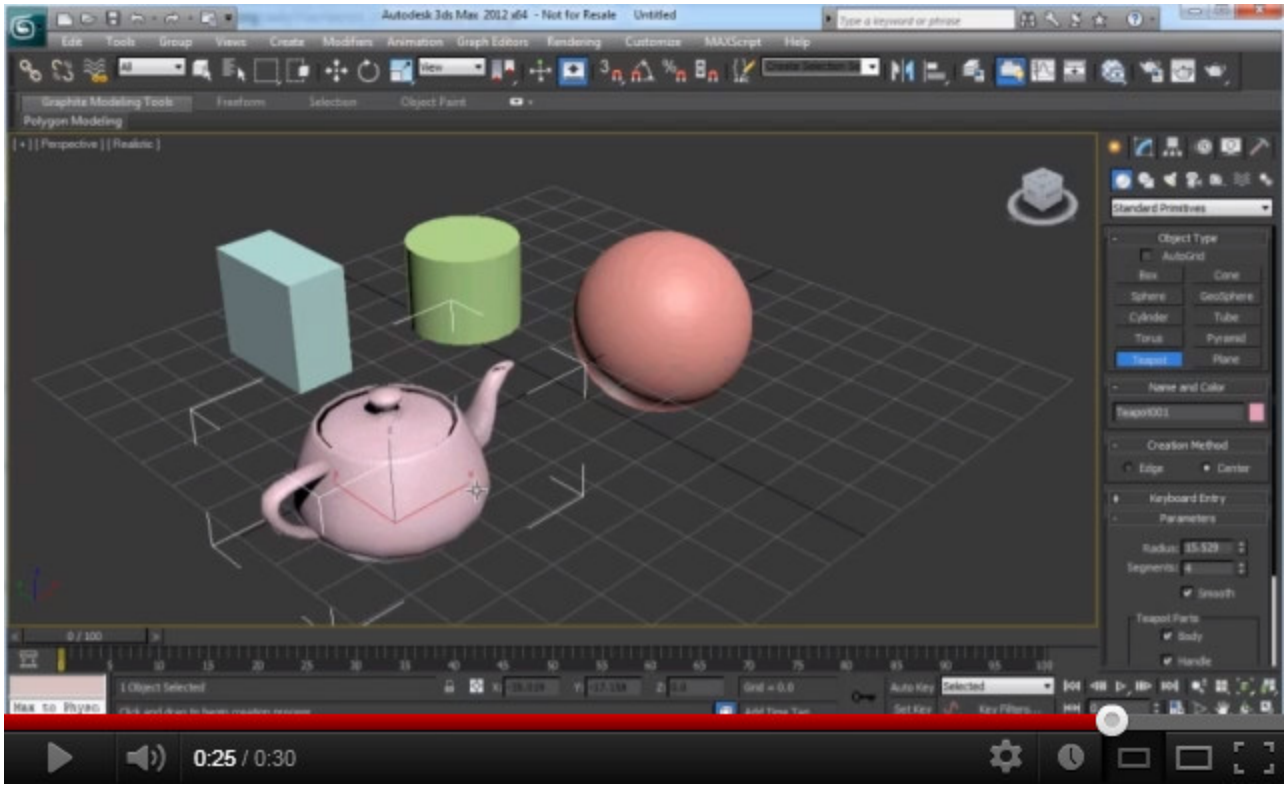
The first triangle is the one that appears. You can verify this by commenting out this triangle and seeing that nothing then appears on the screen.

The second triangle is then the one that's causing problems. We've been told the triangle disappears when culling is turned on, so it's probably the vertex order. By swapping any two indices, you reverse the order of the triangle - here I've swapped the first two. Doing this solves the problem and gives us the right answer.

Lesson: Higher-Order Primitives

[see this YouTube video, http://www.youtube.com/watch?v=VfoDjvkucm8, for the background being formed below.]

[**UDACITY** I have a video clip I have given to Katy. There's the fast and teapot-madness version, which I prefer]



Dealing with individual triangles is pretty painful, as you've probably noticed from the exercises so far. Creating routines such as a polygon creator makes life worth living again.

However, the most common way to generate geometric objects is to use a modeling program. Writing such programs is the bread and butter for the company I work for, Autodesk. Here I'm using 3DStudio MAX to create a number of basic objects, including the teapotahedron.

There are plenty of other ways to acquire models. For example, laser scanners can be used to find a cloud of points. This data is then processed to make a mesh. Another method is to take a series of photos and have a program derive the shape of the object from these.

Whatever process is used, you normally end up with a geometric mesh of triangles. Within WebGL and three.js there's a whole process where you can take a mesh in some 3D file format and convert it to a form that WebGL can read. I could spend the next few units discussing this process, but it's not all that relevant to interactive rendering itself. Take a look in the additional course materials for where to learn more about this process.

[Instructor Comment: As far as converting objects into resources that a WebGL program can read in, three.js includes a number of model loaders. See the Loader class and the various loaders available, http://mrdoob.github.com/three.js/docs/53/#Reference/Loaders/Loader

webgl-loader http://code.google.com/p/webgl-loader/ is a project by Google that helps you

convert models into a compressed format for fast loading into WebGL web pages.

Various ways to load geometry into WebGL are discussed in Chapter 2 of the WebGL Beginner's Guide, http://www.amazon.com/WebGL-Up-Running-Tony-Parisi/dp/144932357X?tag=realtimerenderin and Chapter 7 in WebGL: Up and Running http://www.amazon.com/WebGL-Up-Running-Tony-Parisi/dp/144932357X?tag=realtimerenderin

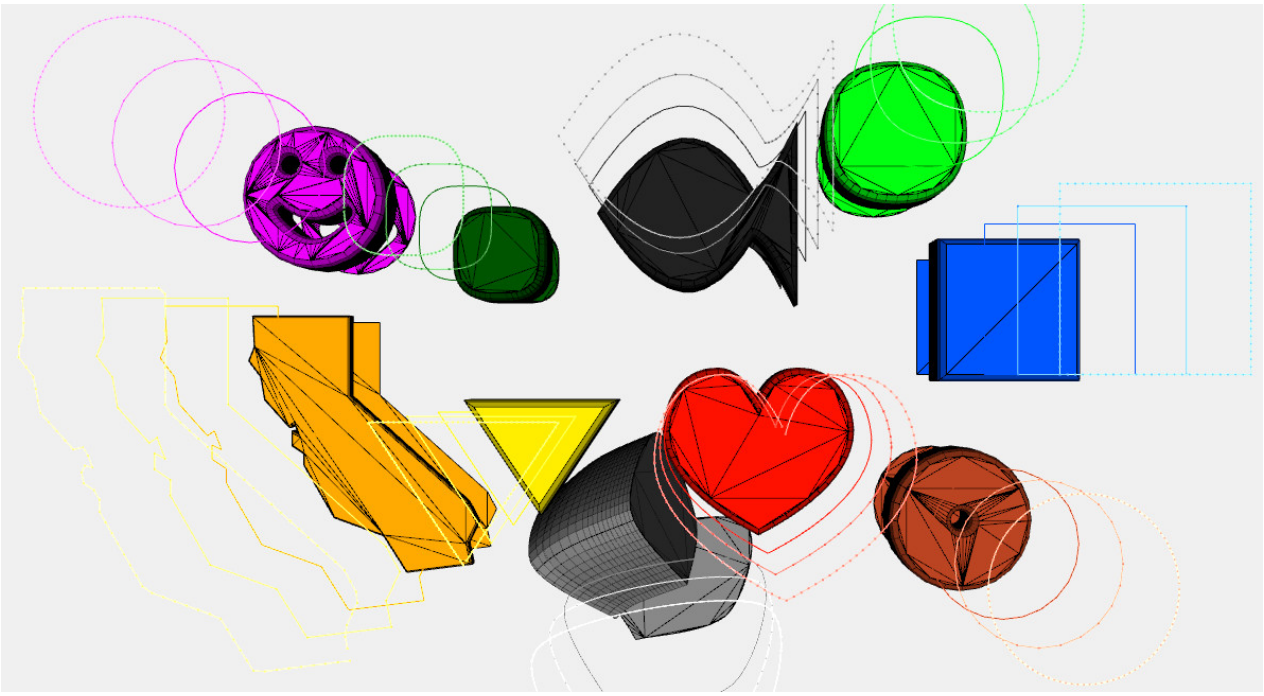
Lee Stemkoski gives commented code showing how to load models from Blender http://stemkoski.github.com/Three.js/Model.html. You'll want to get his code from GitHub https://github.com/stemkoski/stemkoski.github.com to see the model file itself.
]

Lesson: Conclusion TODO again

Recorded 3/25

[* Conclusion (voice over demo) -
Use http://mrdoob.github.com/three.js/examples/webgl_geometry_shapes.html as background]

"You've now got down what's sometimes the hardest thing in 3D computer graphics: getting a single triangle to show up on the screen. In the problem sets that follow, you'll get a chance to do a lot more programming, using triangles and other primitives to create different objects."



Interview: Sam Black

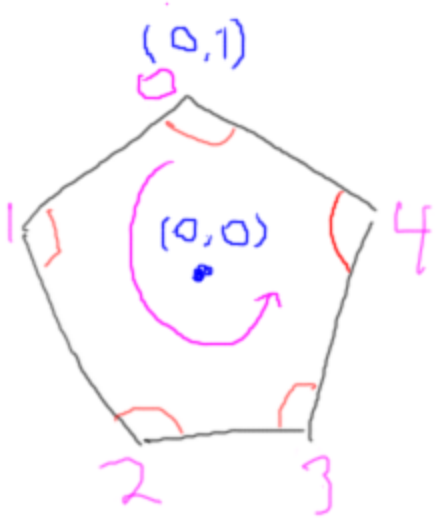
[this is where Sam Black's interview on materials and performance will go]

Problem Set 2

Problem 2.1: Make a Polygon Creator

Your task in this problem is to write a regular polygon creator. A regular polygon is one that has equal angles and equal length sides.

[draw pentagon and show regularity]



Successive points will be counterclockwise. Your job is to form the minimum number of triangles.
[put code on screen]

Your draw function is called PolygonGeometry in the code. It takes as an input the number of sides desired.

[point to code]

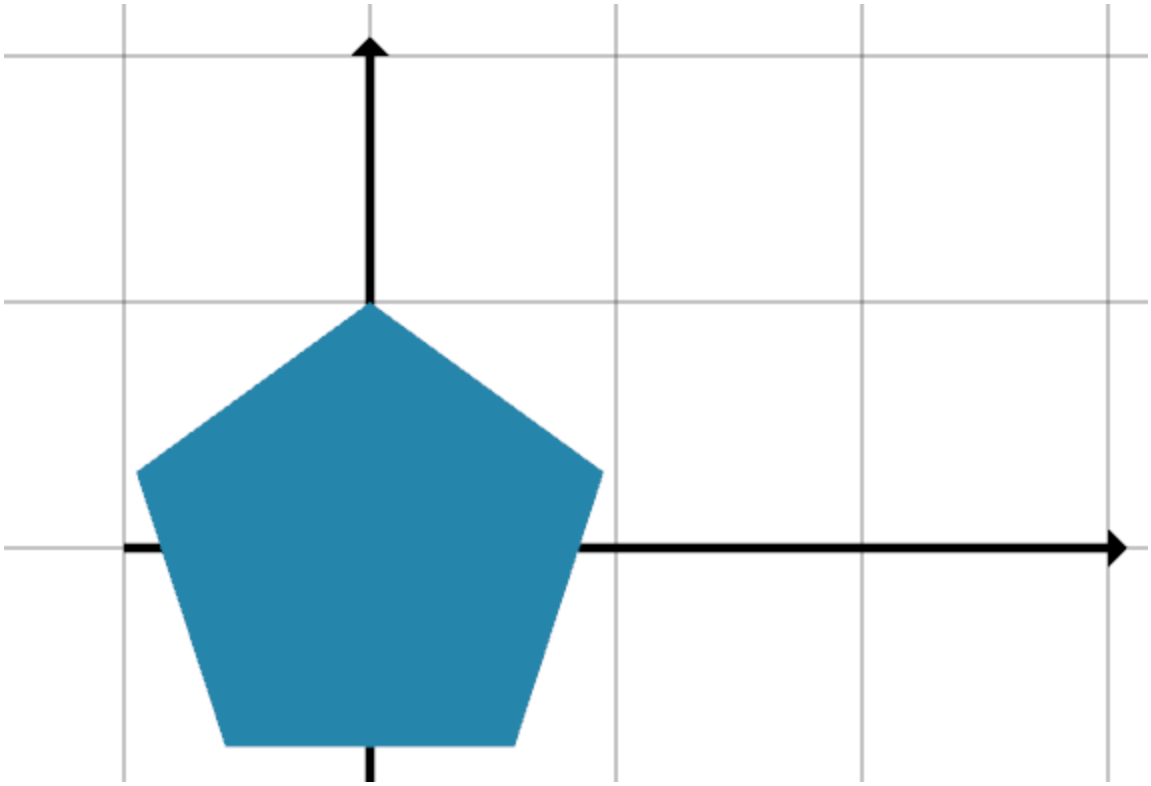
I'll provide you the basic trigonometry function for generating the polygon's points when the radius is 1 and the location is the origin.

```
function PolygonGeometry(sides) {
    var geo = new THREE.Geometry();

    // generate vertices
    for ( var pt = 0 ; pt < sides; pt++ )
    {
        // Add 90 degrees so we start at +Y axis, rotate counterclockwise around
        var angle = (Math.PI/2) + (pt / sides) * 2 * Math.PI;

        var x = Math.cos( angle );
        var y = Math.sin( angle );

        // Save the vertex location - fill in the code

    }

    // Write the code to generate minimum number of faces for the polygon.

    // return the geometry object
    return geo;
}
```

This code that produces x and y creates the polygon's vertices.

For example, when the loop variable pt is 0, the 2D point (x,y) is going to be the point (1,0). Your job is to save the vertices, then use these vertices in a loop that generates the faces for the polygon.

Here's what you're aiming for, a regular polygon centered around the origin:



Answer

There are two parts to this problem. The first part is saving the vertex in the Geometry object. This is pretty straightforward, and you've seen it before: you add the point to the vertices:

```
var x = Math.cos( angle );
var y = Math.sin( angle );

// Save the vertex location
geo.vertices.push( new THREE.Vector3( x, y, 0.0 ) );
```

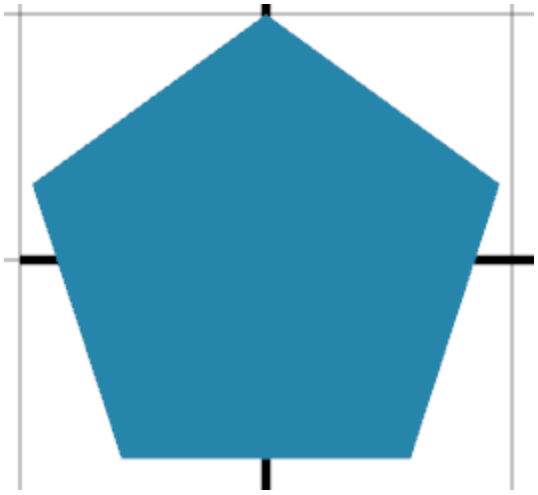
The trickier part is generating the faces. There are a number of ways to do this, here's mine:

```
for ( var face = 0 ; face < sides-2; face++ ) {
    // this makes a triangle fan, from the first +Y point around
    geo.faces.push( new THREE.Face3( 0, face+1, face+2 ) );
}
```

You may recall from the triangulation exercises that a polygon with some number of points generates points minus 2 triangles. The loop here uses this fact, with "face" looping through this many triangles. My loop makes this fan, with all triangles including this topmost point and

radiating from there.

[DRAW triangles on this pentagon]

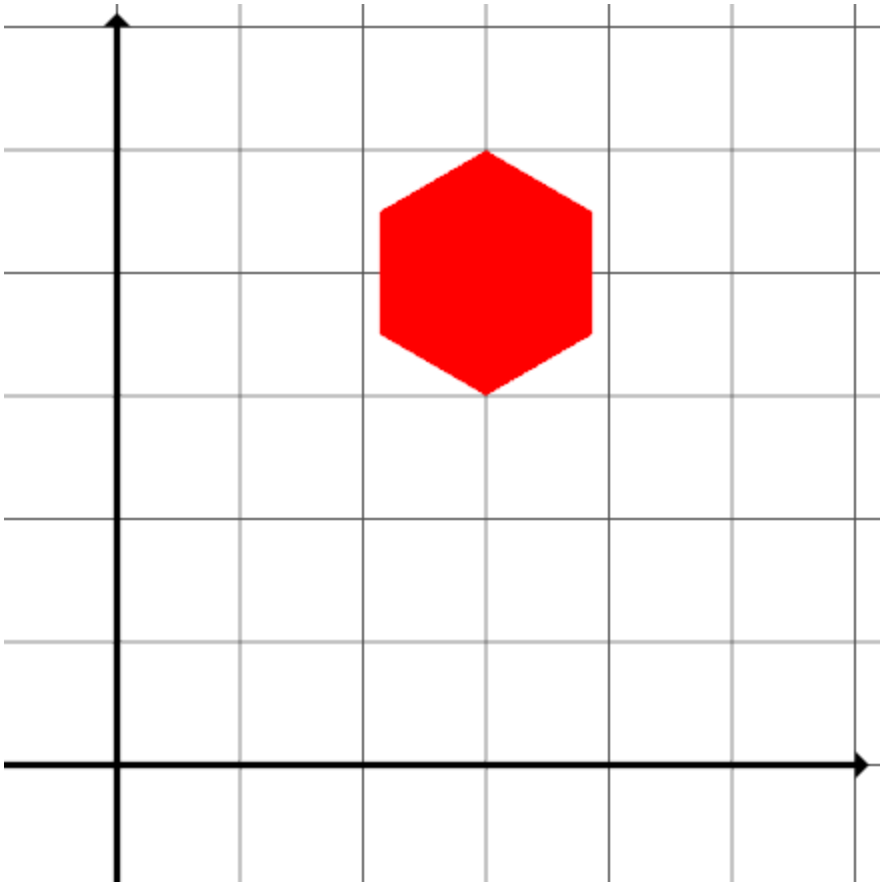


Problem Set 2.2: Move a Polygon to a Location

Your next problem is not too tough. The PolygonGeometry function has been extended to include a location. This is where the center of the polygon is to be located when it's created.

```
function PolygonGeometry(sides, location)

// location will be given as THREE.Vector3(x, y, z)
// you can access the values as location.x, location.y etc
```

Note that the location is passed in as a Vector3, so you'll access the location as shown, with location.x and location.y. You can ignore the value of location.z - I'm going to! Your program should look like this when running correctly:



Answer

The solution is to add the location to the x and y coordinates being generated. Doing so moves all the vertices by this amount. Since all the vertices are generated relative to the origin, this essentially moves the origin to this new location.

```
var x = Math.cos( angle ) + location.x;
var y = Math.sin( angle ) + location.y;
```

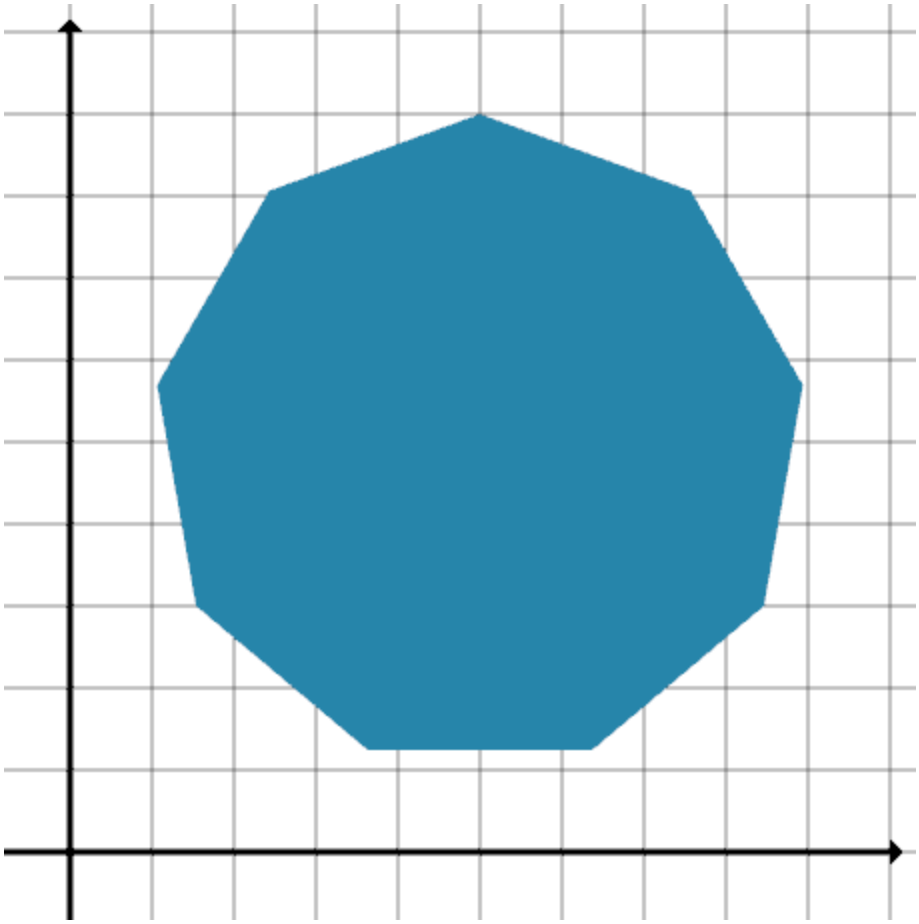
Problem Set 2.3: Change the Polygon's Radius

There's one more useful thing we could add to our polygon creator: changing the radius of the polygon. By default, the distance of all points from the polygon's center is one. Now modify the program to use a radius to make the polygon generated larger, or smaller. Here's your new

interface:

```
function PolygonGeometry(sides, radius, location)
```

Notice that we've changed the order of arguments here. In the previous problem the location was the second argument, now it's third. This is usually not a great idea, especially in JavaScript, as this language does not really check if the number of arguments passed in matches the number of arguments expected. We did it this way because we felt the location change was a bit easier than the radius change.



Answer

You again have to modify the X and Y values computed. By multiplying the original point by the radius, this expands - or contracts - these points around the origin. After that, you add the location to move the whole polygon elsewhere, same as before.

```
var x = radius * Math.cos(angle) + location.x;
var y = radius * Math.sin(angle) + location.y;
```

This gives you a little taste of the unit on Transforms, where we'll be talking about these sorts of operations in depth.

Problem 2.4: Build a Stairway

```
function createStairs() {
```

Here's your chance to build a stairway to, well, not heaven, but at least to a gold cup. A virtual gold cup.

This exercise gets you familiar with how most of three.js's geometric objects work. You'll be spending your time working inside the create stairs function.

```
// +Y direction is up
// Define the two pieces of the step, vertical and horizontal
// THREE.CubeGeometry takes (width, height, depth)
var stepVertical = new THREE.CubeGeometry(stepWidth, verticalStepHeight, stepThickness);
var stepHorizontal = new THREE.CubeGeometry(stepWidth, stepThickness, horizontalStepDepth);
```

We're going to use the CubeGeometry method to create two pieces for steps, a vertical piece and a horizontal plank. CubeGeometry is not a perfect name, since a cube is defined as having all its dimensions be the same size - BoxGeometry would have been better, but no big deal. Here's the code for defining the geometry for our two steps. The way Cube Geometry works is that you provide it a size in X, Y, and Z. It then creates a set of triangles for you forming a box with these dimensions.

```
var stepMesh;

// Make and position the vertical part of the step
stepMesh = new THREE.Mesh( stepVertical, stepMaterialVertical );
// The position is where the center of the block will be put.
// You can define position as THREE.Vector3(x, y, z) or in the following way:
stepMesh.position.x = 0;            // centered at origin
stepMesh.position.y = verticalStepHeight/2; // half of height: put it above ground plane
stepMesh.position.z = 0;            // centered at origin
scene.add( stepMesh );
```

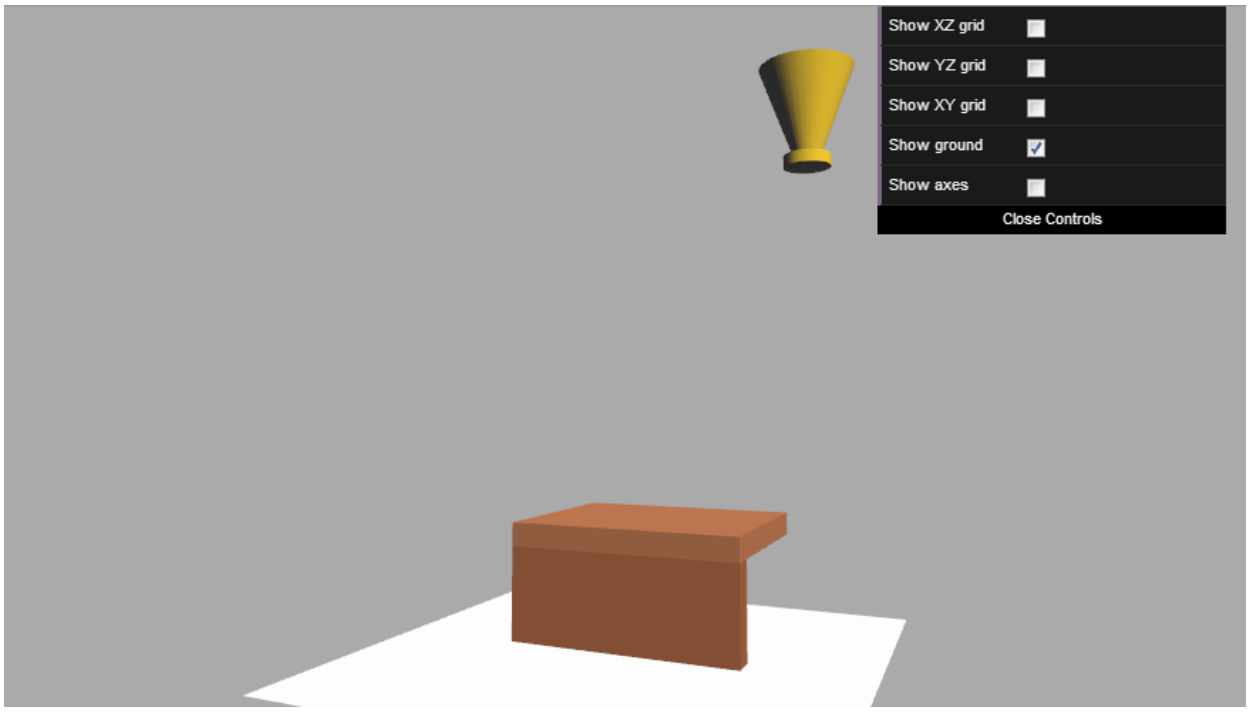
Once you have this geometric object, you can use it as many times as you wish to create three.js Meshes, each of which consist of geometry and a material. You can use the "position" parameter on the Mesh class to reposition the box, and in fact you'll need to do this for the

exercise. When you create a box, it is centered around the origin. For our vertical part of the step, we want this piece to be resting on the ground. To do so, we take the height of the piece and move it half this distance upward, so that the bottom is now at ground level.

```
// Make and position the horizontal part
stepMesh = new THREE.Mesh( stepHorizontal, stepMaterialHorizontal );
stepMesh.position.x = 0;
// Push up by half of horizontal step's height, plus vertical step's height
stepMesh.position.y = stepThickness/2 + verticalStepHeight;
// Push step forward by half the depth, minus half the vertical step's thickness
stepMesh.position.z = horizontalStepDepth/2 - stepHalfThickness;
scene.add( stepMesh );
```

The horizontal piece making up the flat part of the step is created with this code. In this case we first push the step up half its thickness, so it rests on the plane, then push it up the entire height of the vertical piece so that it rests on top. The z position is also changed, pushing the plank so that one edge is at the origin, then pulling it a bit back so it rests firmly on the vertical piece.

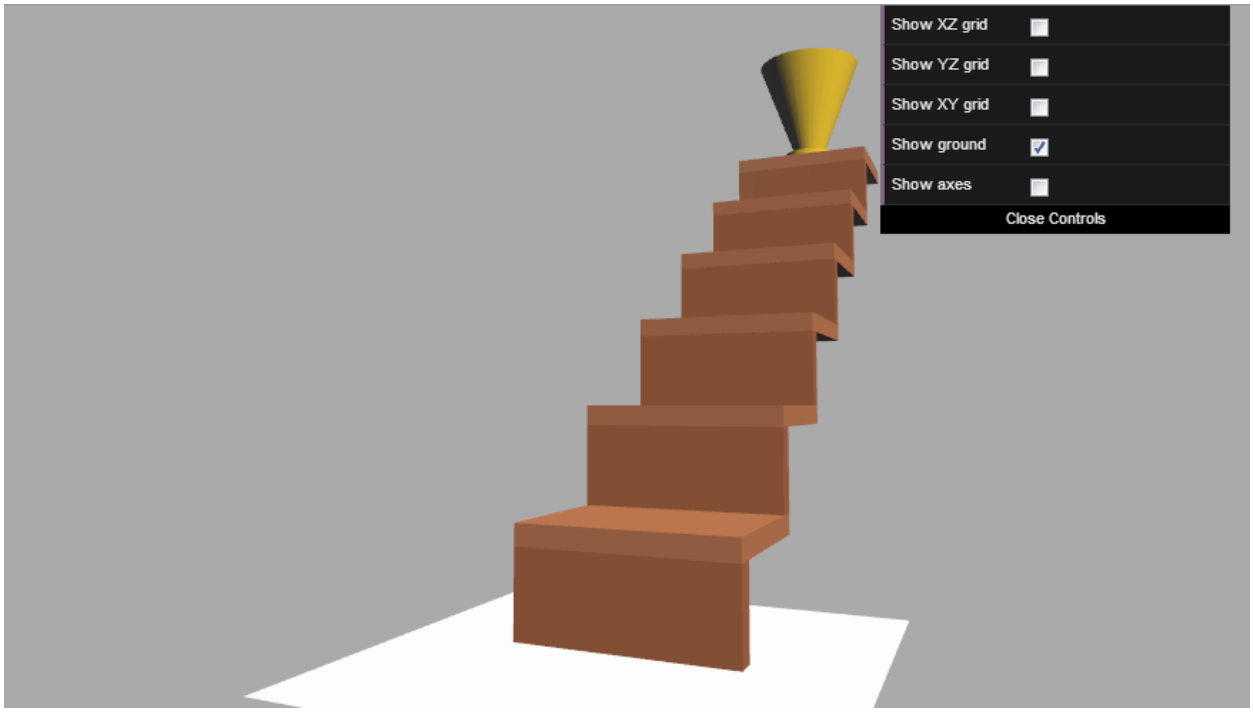
[run the program itself, have Gundega show it while I talk]



Got all that? Once you run the program you'll see the two step pieces, with the vertical piece resting on the ground plane. You'll also notice a cool program feature: there are some controls available. The grids can help you get a sense of distance along each axis. They have lines every 100 units. Each is labeled with the axes that are in the grid plane. For example, since the Y axis is up for this scene, the XZ grid is on the ground plane. [toggle this on]

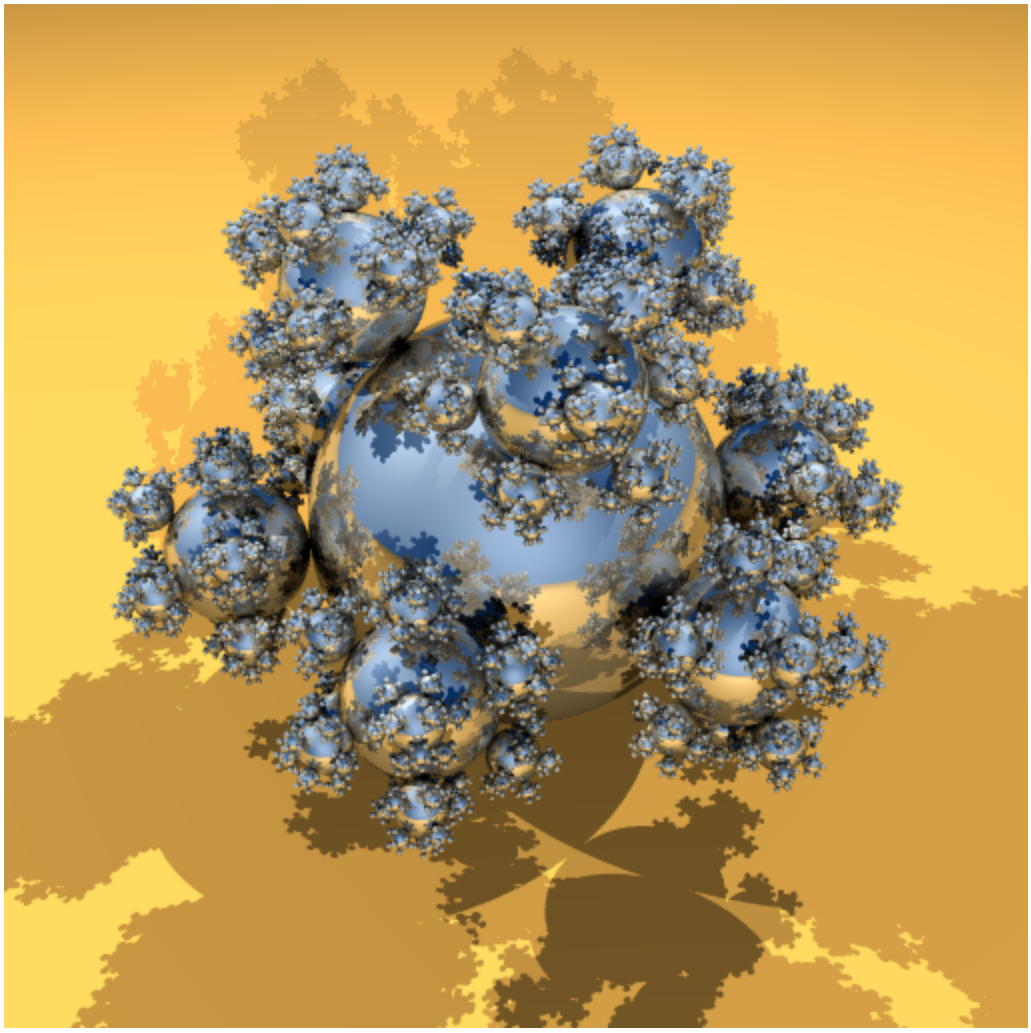
Your goal is to build a number of steps, one on top of the other going forward, that just reaches the gold cup.

[Run this program, too:]



Here's what things will look like when you're done. Don't build any more steps than are necessary, or we'll charge you for extra time and materials. Plus, you won't pass the problem.

I recommend that you first play around with the position parameter and see its effect on the model. Setting the position to 0 shows what the block looks like centered at the origin. You should also consider using a loop, as it's a lot easier in the long run than brute force - being a smart programmer can save you a lot of grunt work.



Using programs to generate geometry like this is called procedural modeling. Personally, I'm pretty bad at using modelers, so I like modeling this way. It's great that with a few lines of code you can generate pretty complex models. Here's a procedural model called sphereflake that I made more than 25 years ago. Around the same time, Pixar created a short called Red's Dream primarily using procedural modeling, since there simply weren't great modeling tools easily available back then. See the additional course materials to download the program for this model and others, along with pointers to many other related resources.

But don't get distracted right now - go ahead and build the steps!

[Additional resources: The sphereflake generator and other procedural model makers are available for download here http://tog.acm.org/resources/SPD/. It surprises me that code doesn't really rust over the years. L-Systems are a form of procedural modeling that is particularly suited for making organic forms http://en.wikipedia.org/wiki/L-system. For computer aided design, the free OpenSCAD program http://www.openscad.org/ is pretty useful for precise programmatic control. There are plenty of other interesting resources out there; a good place to start is the

Wikipedia page on procedural modeling http://en.wikipedia.org/wiki/Procedural_modeling
]

Answer

[Draw drawing of the first step, then the second step, and show the height and depth offsets and how they're computed. Label the riser height and riser depth]

Congratulations! I'm guessing this problem took you a bit to figure out. My teaching assistant Gundega Dekena came up with this problem, so it was fun for me to solve my own way. The key insight for me was that each step needed to be a constant distance up and forward from the previous step. For the second step, I started by adding the height of the first step to its location. This first step's height is the vertical step piece's height, plus the step thickness. The other piece of the puzzle was how far forward the second step is from the first. From the diagram we see that this is the horizontal step depth minus the step thickness.

With these two values in hand I could create a loop to generate all the steps.

```
// how much each step piece is moved up and forward
var riserHeight = verticalStepHeight+stepThickness;
var riserDepth = horizontalStepDepth-stepThickness;

for ( stepPair = 0; stepPair < 6; stepPair++ ) {
    // Make and position the vertical part of the step
    stepMesh = new THREE.Mesh( stepVertical, stepMaterialVertical );
    // The position is where the center of the block will be put.
    // You can define position as THREE.Vector3(x, y, z) or in the following way:
    stepMesh.position.x = 0;            // centered at origin
    stepMesh.position.y = verticalStepHeight/2 + stepPair*riserHeight;
    stepMesh.position.z = stepPair*riserDepth;
    scene.add( stepMesh );

    // Make and position the horizontal part
    stepMesh = new THREE.Mesh( stepHorizontal, stepMaterialHorizontal );
    stepMesh.position.x = 0;
    // Push up by half of horizontal step's height, plus vertical step's height
    stepMesh.position.y = stepThickness/2 + verticalStepHeight + stepPair*riserHeight;
    // Push step forward by half the depth, minus half the vertical step's thickness
    stepMesh.position.z = horizontalStepDepth/2 - stepHalfThickness + stepPair*riserDepth;
    scene.add( stepMesh );
}
```

Here's my code. At the top I computed the height and depth I need to offset between each step.

Through experimentation found I needed to make six steps with my loop.

On each loop iteration I modified both the vertical and horizontal parts of the step by adding in the height and depth offsets.

[Additional resources: The same resources page from the previous question page could be linked from here - I love procedural modeling, it’s a rich field and great for programmers.

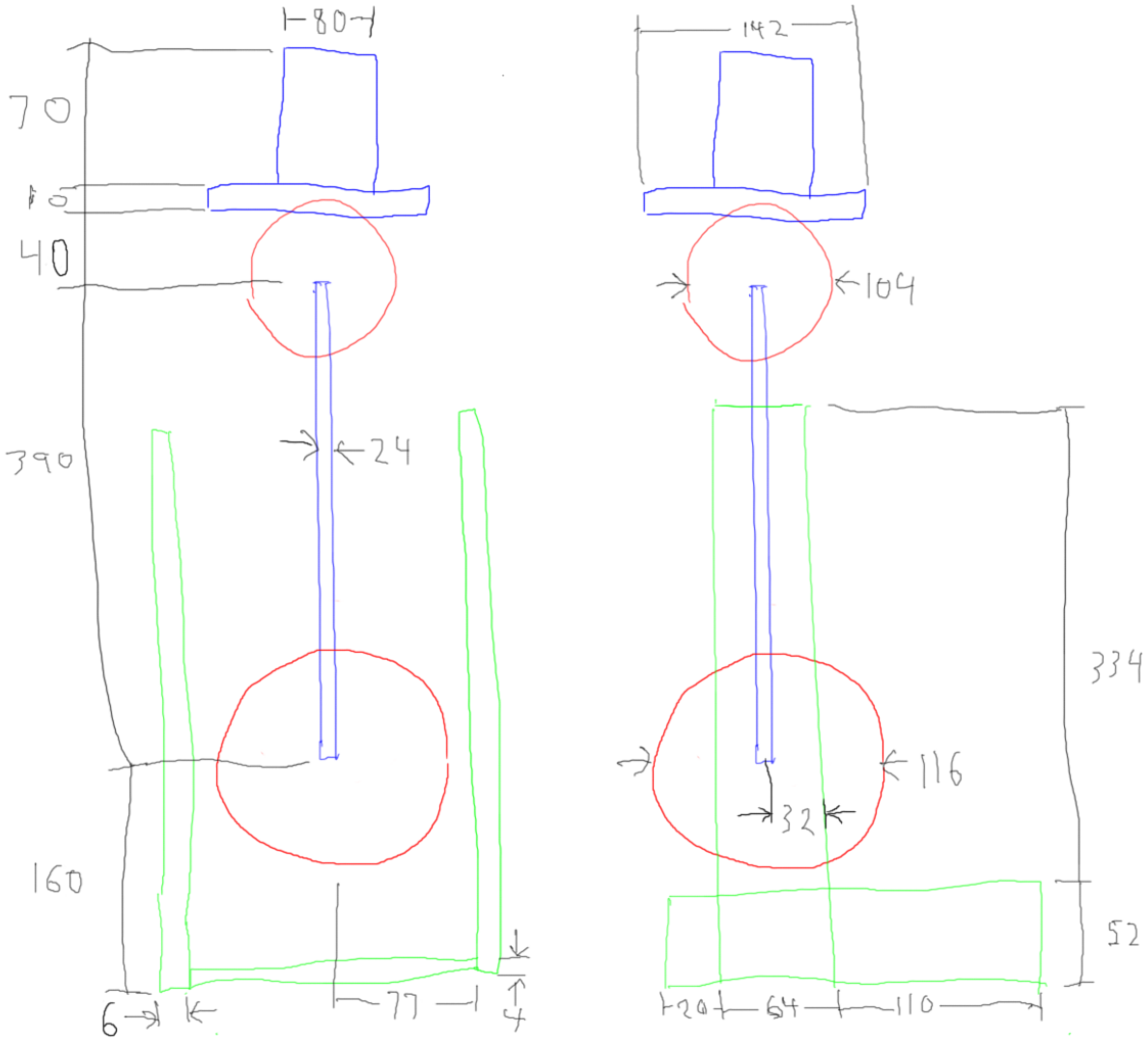
Problem 2.5: Make a Drinking Bird’s Body

[replaced with Katy’s video]

[**DRINKING BIRD VIDEO GOES HERE**]

Your goal is to make a simple model to manipulate in future units. You'll create a basic drinking bird model - very basic, at this point he won't even have a beak. We'll get to that in a later unit. The drinking bird is a little toy that drinks from a cup - Google it!

[Drinking bird - creative commons http://en.wikipedia.org/wiki/File:Sipping_Bird.jpg - use hi-res version at http://upload.wikimedia.org/wikipedia/commons/3/3a/Sipping_Bird.jpg and **make sure it's flipped to point to the right, as shown below**. This then matches the view of the exercise.]



The objects you'll make are the legs, body, head (without a nose or eyes), and jaunty hat. Here's a blueprint, showing front and side views. Well, it's more a back-of-the-napkin drawing, but sometimes that's all you get from a client. Me, my favorite client comment I heard that one designer received was, "could you please make that sphere rounder?"

[blueprint - I decided to not make the legs orange, even though they're orange above.

WRITE OUT ALL NUMERALS, so if there's a problem they can video edit and fix.]

[Write in parameter names (in different color than code):]

[width, height, depth
radius, segmentsWidth, segmentsHeight
radiusTop, radiusBottom, height, radiusSegments]

```
boxGeometry = new THREE.CubeGeometry( 125.6, 389.8, 202.1 );

sphereGeometry = new THREE.SphereGeometry( 202.1, 32, 16 );

cylinderGeometry = new THREE.CylinderGeometry( 29.4, 202.1, 553.5, 32 );
```

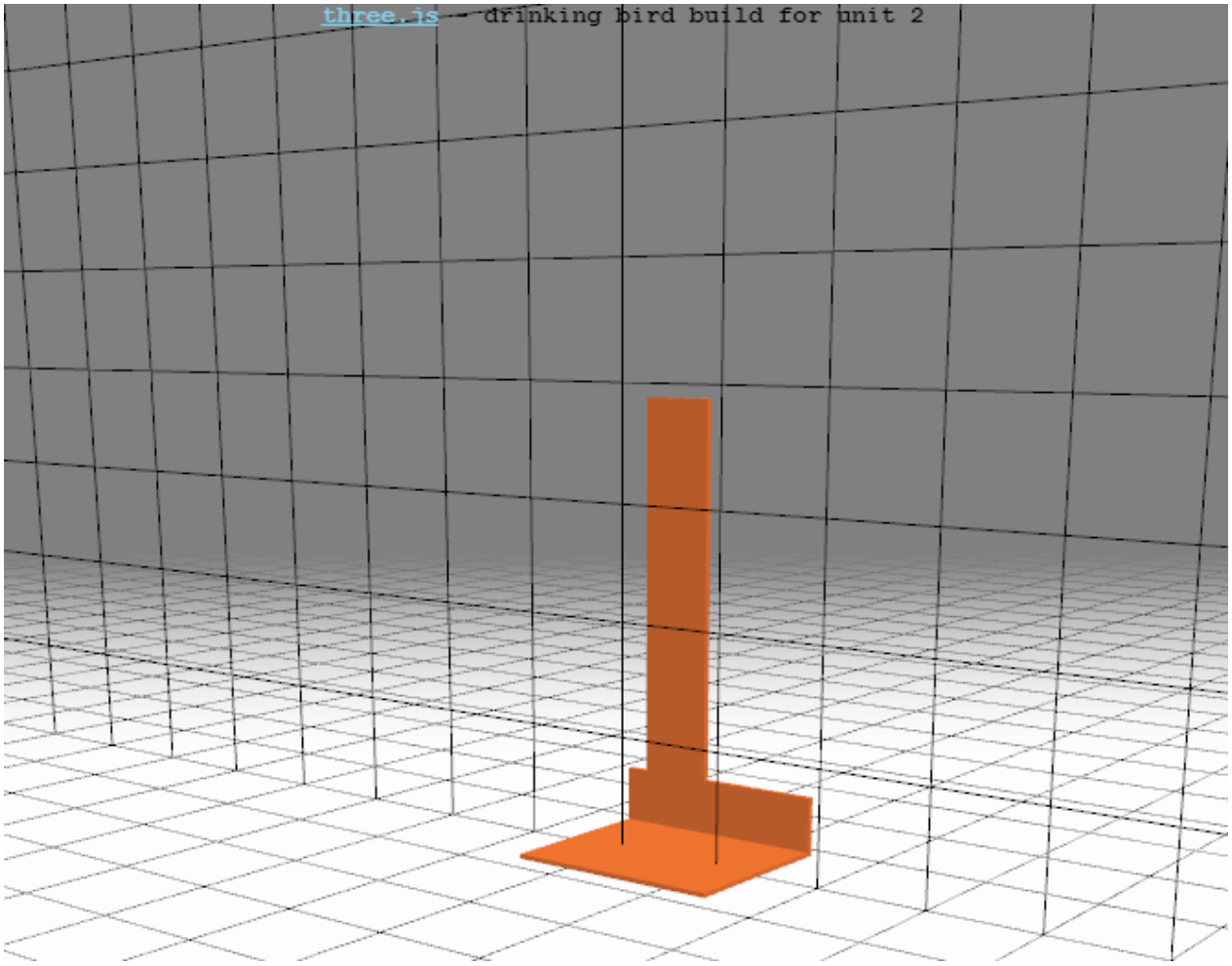
The three objects you'll use are the cube, sphere, and cylinder. Here are typical calls for creating these objects:

For the cube, the numbers passed in are the width, height, and depth of the box formed.

The sphere needs just a radius. The additional two numbers specify how much tessellation happens along the equator and from pole to pole, respectively. You might try changing these tessellation numbers to see the effect on the body, but please use these numbers 32 and 16 when you submit your answer.

The cylinder gives an upper and lower radius, followed by a height and by the amount of tessellation around its equator. Again, please use the number 32.

See the additional course materials for the documentation for these methods.



There are three functions you'll be working on..

The function **createSupport()** will make the drinking bird's support frame - base, legs and feet. To get you started, I've laid down a base that the legs connect to, along with the left leg. You have to add the right leg and feet.

[Now clear and write out function names]

The function **createBody()** will create the bird's body and spine.

The function **createHead()** will make the head and hat.

You have to create the objects in the correct location and add them to scene. When you have one that, you will see the model of the drinking bird, just like this [show demo of the solution]

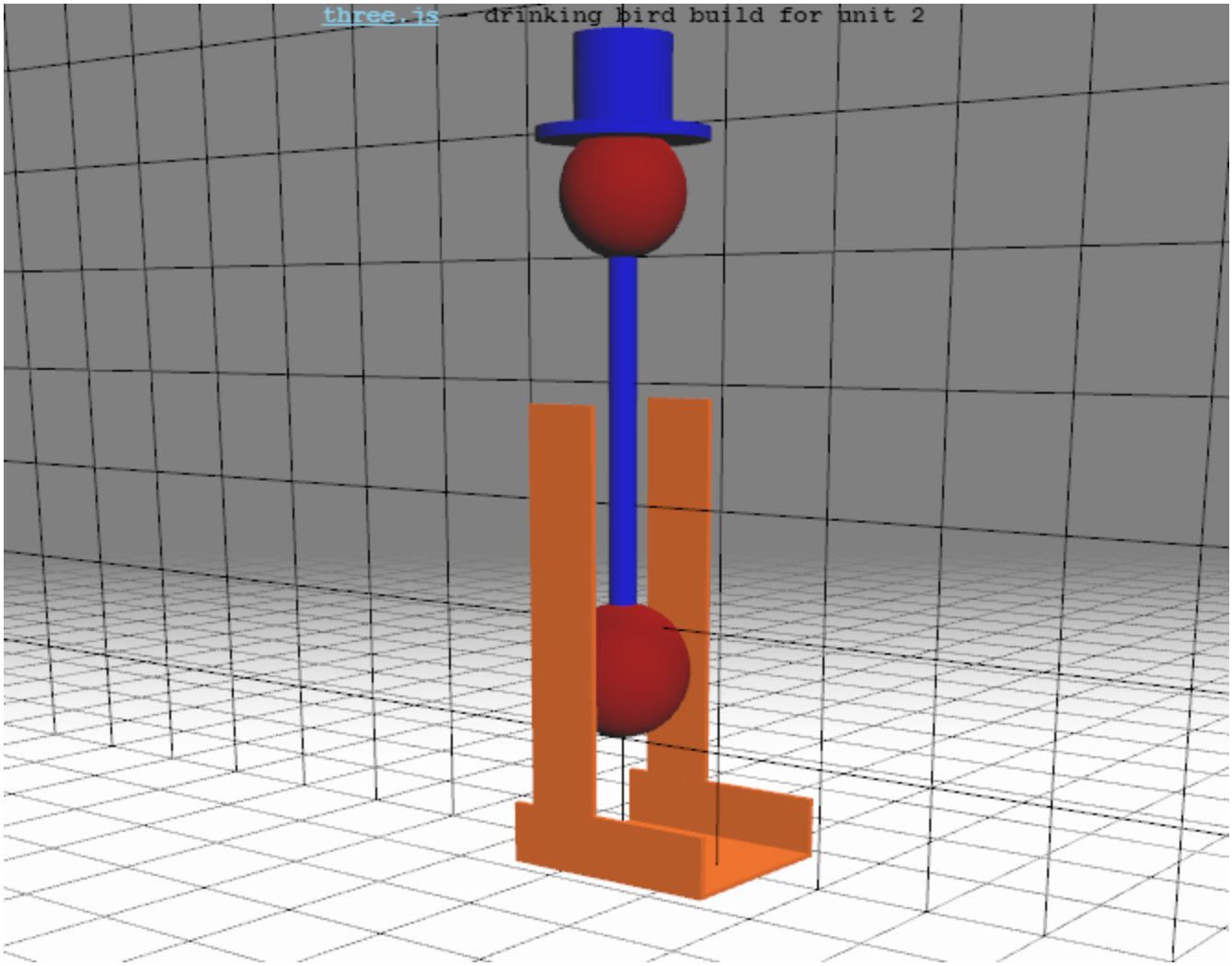
[code for this exercise is at
http://www.realtimerendering.com/udacity/?load=unit2/drinking-bird.js]

[Instructor Comment: See this web page http://mrdoob.github.com/three.js/docs/56/#Reference/Core/Geometry for three.js documentation. The cube, sphere, and cylinder methods are listed towards the bottom.

There are many other geometric objects supported by three.js. See Lee Stemkoski's demos for shapes http://stemkoski.github.com/Three.js/Shapes.html, extrusion http://stemkoski.github.com/Three.js/Extrusion.html, and 3D text http://stemkoski.github.com/Three.js/Text3D.html. There are also demos for subdivision surfaces http://stemkoski.github.com/Three.js/Subdivision-Cube.html, a way to smooth an initial mesh, and http://stemkoski.github.com/Three.js/CSG.html, a way to subtract one object from another.]

Learn about the drinking bird here http://en.wikipedia.org/wiki/Drinking_bird and here http://chemistry.about.com/od/physicalchemistrythermo/a/How-The-Drinking-Bird-Science-Toy-Works.htm]

Answer



Well, that was certainly a pain, wasn't it? Now you know why you should always buy Autodesk-brand 3D modelers and help pay my salary, or use a free alternative, like Blender. Well, I do want to put in a little plug here, which I had absolutely nothing to do with but am very happy about: all Autodesk software I know of is free to students, teachers, veterans, and unemployed professionals - see the additional course materials for information.

Really, there's not much to say about this exercise: either you got it or you didn't.

[Find solution at ...]

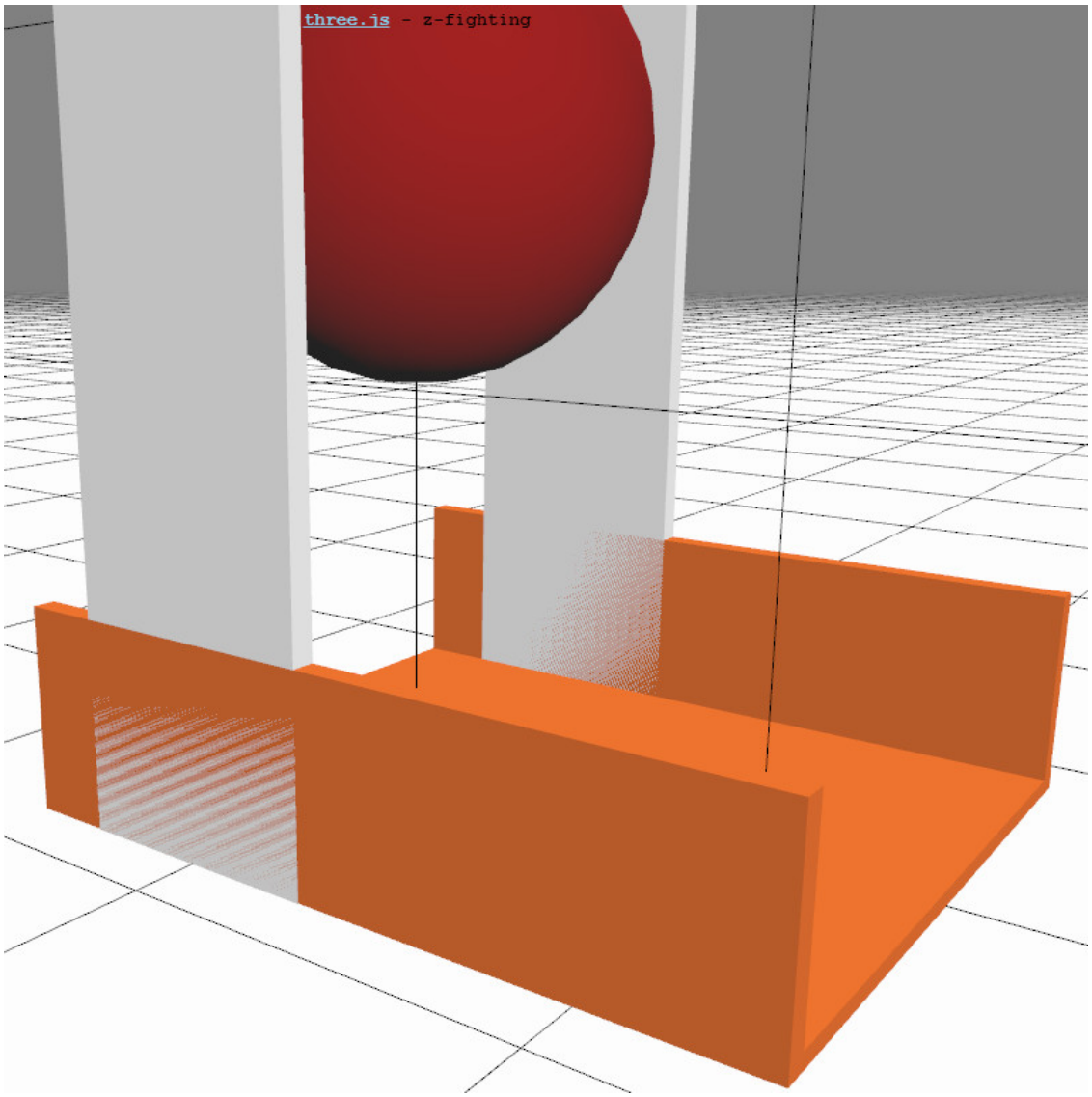
[Instructor Comments: I'd say the following even if I didn't work for Autodesk. If you're a student, faculty member, veteran, or out of work professional engineer, you can download and use all Autodesk software for free. See this link: http://www.realtimerendering.com/blog/all-autodesk-software-free-to-students-and-educators-and-betas-for-everyone/. There are other free modelers out there, see Wikipedia

http://en.wikipedia.org/wiki/List_of_3D_modeling_software. Nice list of 3D free modelers: http://www.hongkiat.com/blog/25-free-3d-modelling-applications-you-should-not-miss/

For how to import models into three.js, see the wiki: https://github.com/mrdoob/three.js/wiki
]

Problem 2.6: Z-Fighting

[see video at http://www.youtube.com/watch?v=ZWNKTxx2tvg&feature=youtu.be, which needs to be shot again - easy to do, just run http://www.realtimerendering.com/udacity/?load=demo/unit2-z-fighting.js]



The drinking bird is starting to look sort-of vaguely like the real thing, if you squint a lot. Modifying this model a bit more, I made the legs white. When I do this and run the program, this causes what is called z-fighting, as you can see in this video. What is happening is that two polygons, one from the foot and one from the leg, are overlapping, attempting to occupy exactly the same location. At one pixel one surface's z-depth will be infinitesimally closer than the other, due to precision limits. At the next pixel over the situation may be reversed.

While this is fine mathematically, it can't happen in the real world and clearly makes for unconvincing images.

Examine the problem by running the program - find the link in the additional course materials. The question to you is, which of these solutions will fix the problem:

[ ] Make each leg a bit thinner so that it doesn't overlap with the foot
[ ] Use a newer GPU with a higher-precision z-buffer
[ ] Switch to another rendering algorithm such as ray tracing
[ ] Remove the bottom part of each leg so that no part of the leg and foot overlap

[Instructor Comments: run this program http://www.realtimerendering.com/udacity/?load=demo/unit2-z-fighting.js and zoom in with the mouse wheel to see z-fighting in action. Your mileage *will* vary.]

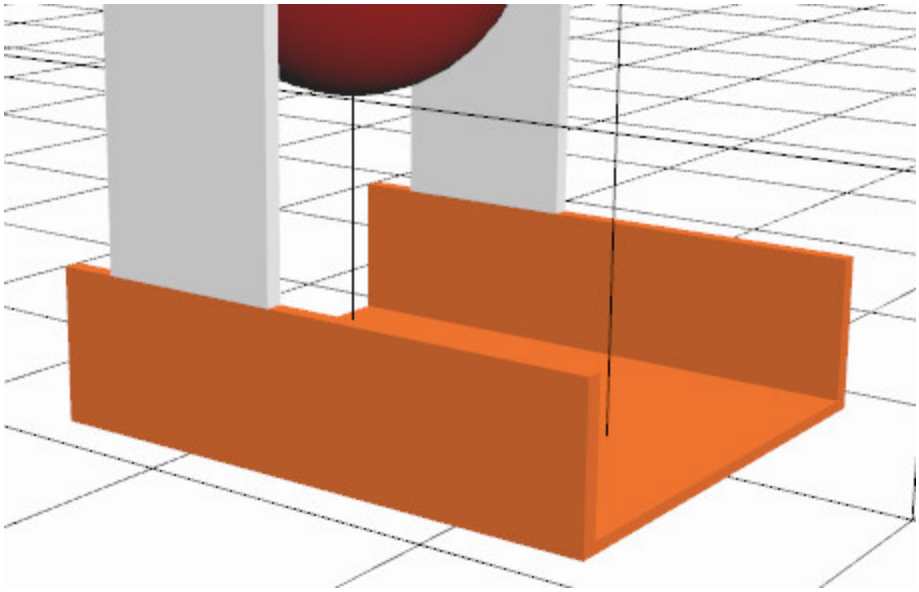
Answer

This first solution is acceptable for the most part. It solves the problem of two surfaces being in the same location. However, it does change the geometry of the model, which might not be acceptable. Also, even surfaces that are near each other can have z-fighting problems, as the Z-buffer has limits on its precision. So this solution works, but is not optimal.

The second solution sounds nice, a new graphics card is always fun, but doesn't help at all. Since both surfaces are in the identical location, which one happens to be closer is arbitrary. Another GPU can, and will, give you a different view of z-fighting. So even if you like the z-fighting effect, you can't control it if your users are on machines with different GPUs.

The third answer, switching to another rendering algorithm, again will just cause different precision errors to show up. The underlying problem of two surfaces being in the same place is a modeling error.

The fourth answer, removing the bottom part of the leg that overlaps with the foot, is certainly the cleanest solution. This is also the most efficient answer, as making an object smaller will mean it covers less pixels and so causes less work.



[Instructor Comments: the fourth solution, shortening the leg, is shown here:
http://www.realtimerendering.com/udacity/?load=demo/unit2-z-fighting_fixed.js]