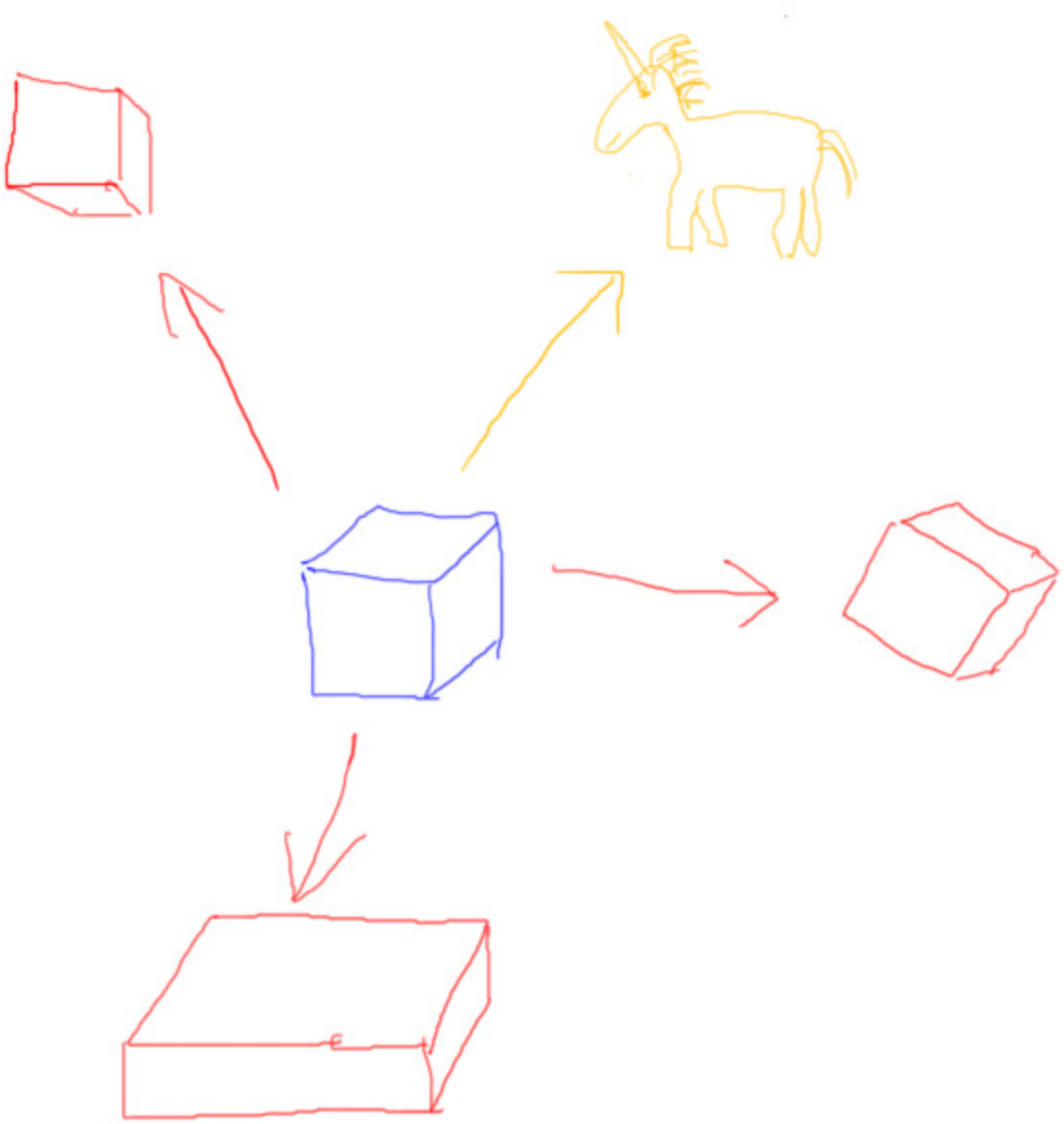
Lesson 4: Transforms

Lesson: Overview

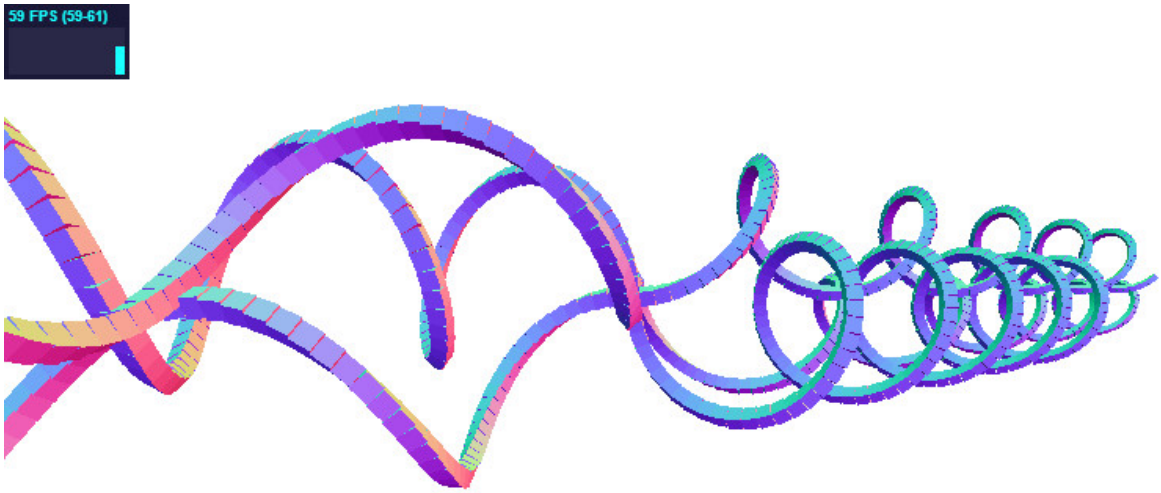
[Draw some simple object and translate, rotate, and scale it, and something else]



In English the word “transform” is a verb and means a dramatic change, such as “I saw the cube transform into a unicorn”. I’m going to use the more mathematical definition, where “transform” is a noun. A transform is an operation that changes the position, orientation, or size and shape of an object. Transforms are a key part of 3D computer graphics.

In these lessons I’ll be explaining how transforms work. You probably didn’t realize it, but you’ve already been using transforms in a few of the previous exercises. Whenever you set the position of an object you were using a type of transform called a *translation*.

[start running demo, just let it run
http://mrdoob.github.com/three.js/examples/webgl_geometry_hierarchy2.html]



We use transforms whenever we animate anything: objects, lights, or cameras. We’re going to spend a fair bit of time on transforms, since they allow us to do so much.

The study of transforms for computer graphics is associated with the field of linear algebra. This field is concerned with vector spaces, and is usually considered a prerequisite for any computer graphics programming course. If you know a bit about linear algebra, great; having a deeper and wider understanding of the subject will help. However, basic computer graphics needs just a few small bits of theory to get things going. My goal in these lessons ahead is to teach you the basic tools you’ll need to quickly and effectively be able to control your virtual world.

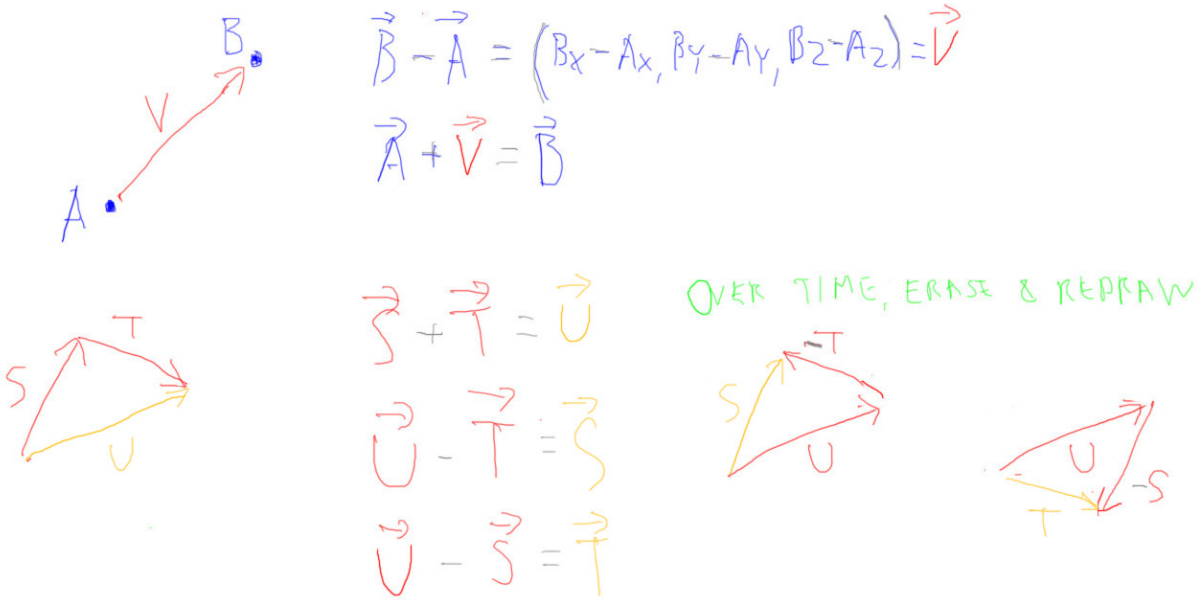
[Additional Course Materials: The demo running in this lesson is http://mrdoob.github.com/three.js/examples/webgl_geometry_hierarchy2.html]

Lesson: Point and Vector Operations

[draw point and vector]

A point is a position, and a vector describes a motion. We can combine points and vectors in various useful ways by adding or subtracting their coordinates from one another.

[diagram]



To start with, if you subtract the location of point A from point B, you get a vector describing how to get from point A to B. So B - A = (Bx-Ax, By-Ay, Bz-Az) = V

[make the vector sign over all vectors!]
B - A = (Bx-Ax, By-Ay, Bz-Az) = V

The reverse holds true: A + V = B

A + V = B

While we can subtract points to get a vector, our system doesn't produce anything geometrically meaningful if you add points together.

[draw vector addition]

We can add and subtract vectors. For example, S + T = U.

S + T = U

And we can reverse the process again, either way: U - T = S

[negations]

U - T = S

and U - S = T

U - S = T

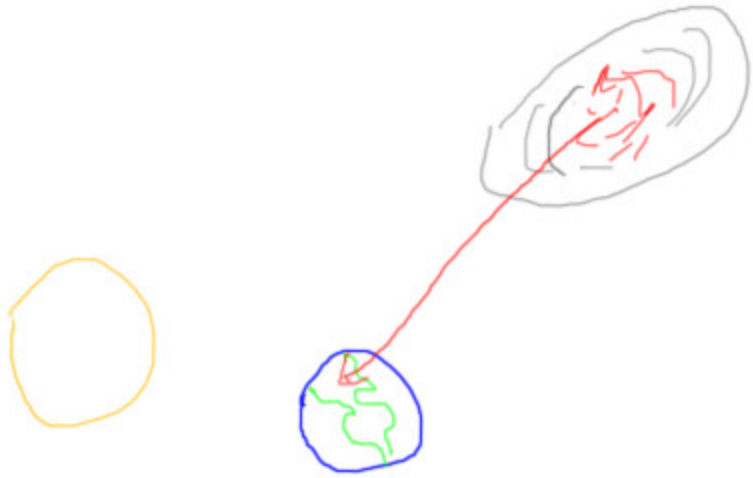
One way to think about vector subtraction is that you're instead adding a vector going the opposite direction, as shown here.

The other thing commonly done with vectors and points is to multiply them by a scalar, that is, a number. Multiplying a vector by a scalar is clear enough: you want to describe moving further in a given direction. If you multiply by -1, you reverse the vector's direction. Multiply by a different negative number and you reverse the direction and change how far to move.

Multiplying a point by a number is a way to make an object larger or smaller. This type of operation, called scaling, we'll discuss in detail later.

Coordinate Values

[CENTER it - give room to the left. draw earth, sun, galaxy]

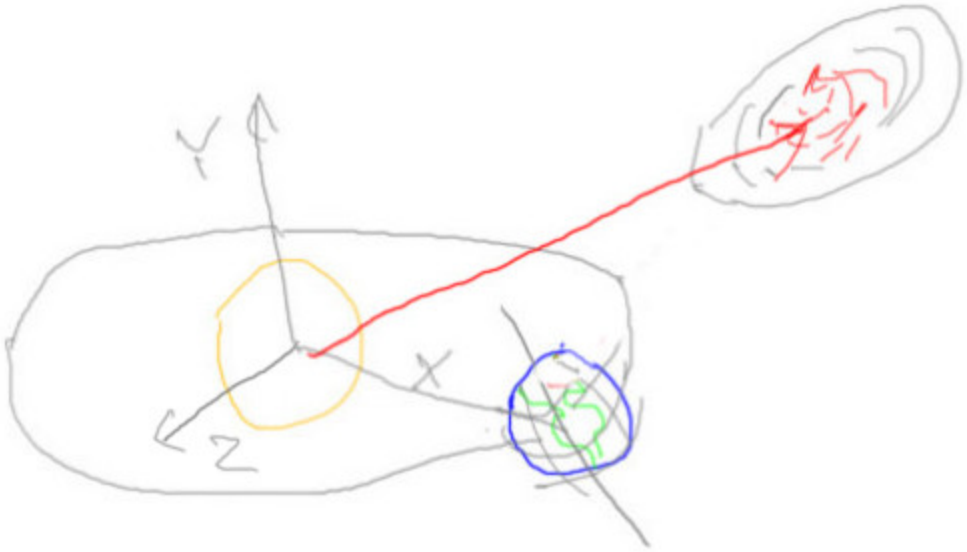


Points and vectors don't require any numbers to be associated with them. For example, I can say "define a vector from the Andromeda Galaxy to Chicago". That vector now exists, but doesn't have any coordinates until I define a frame of reference of some sort.

[draw reference frame for earth]

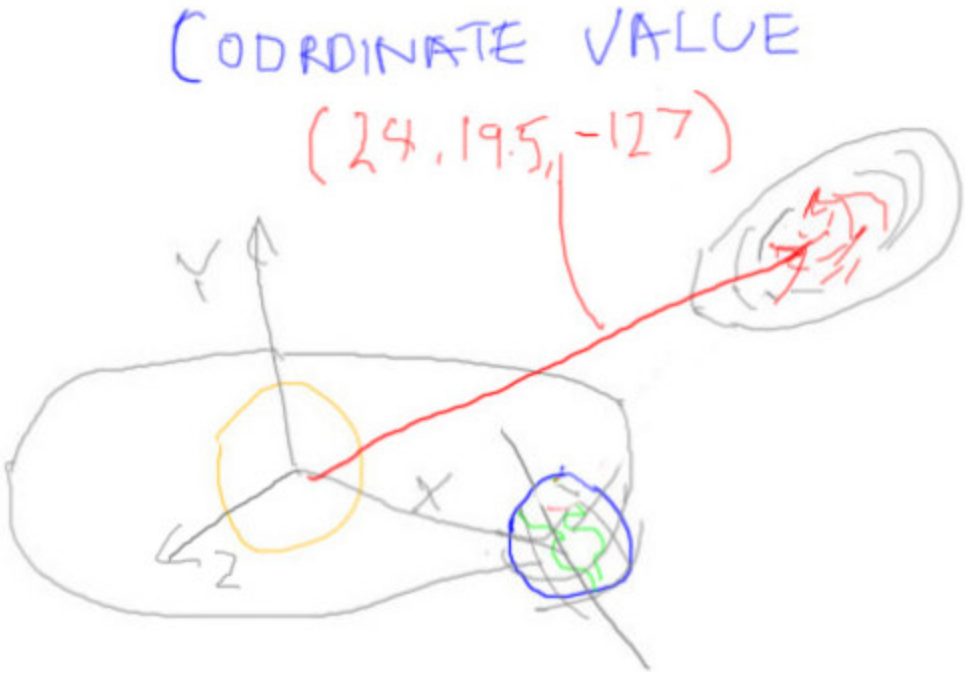
I could ask, "what are the vector's coordinates in terms of the Earth's latitude, longitude, and altitude". That gives one set of coordinates, though ones that change every second as the earth rotates.

[draw sun ref]



Or I could say “what are the coordinates in terms of the sun’s location and the earth’s path?” and get coordinates that change more slowly.

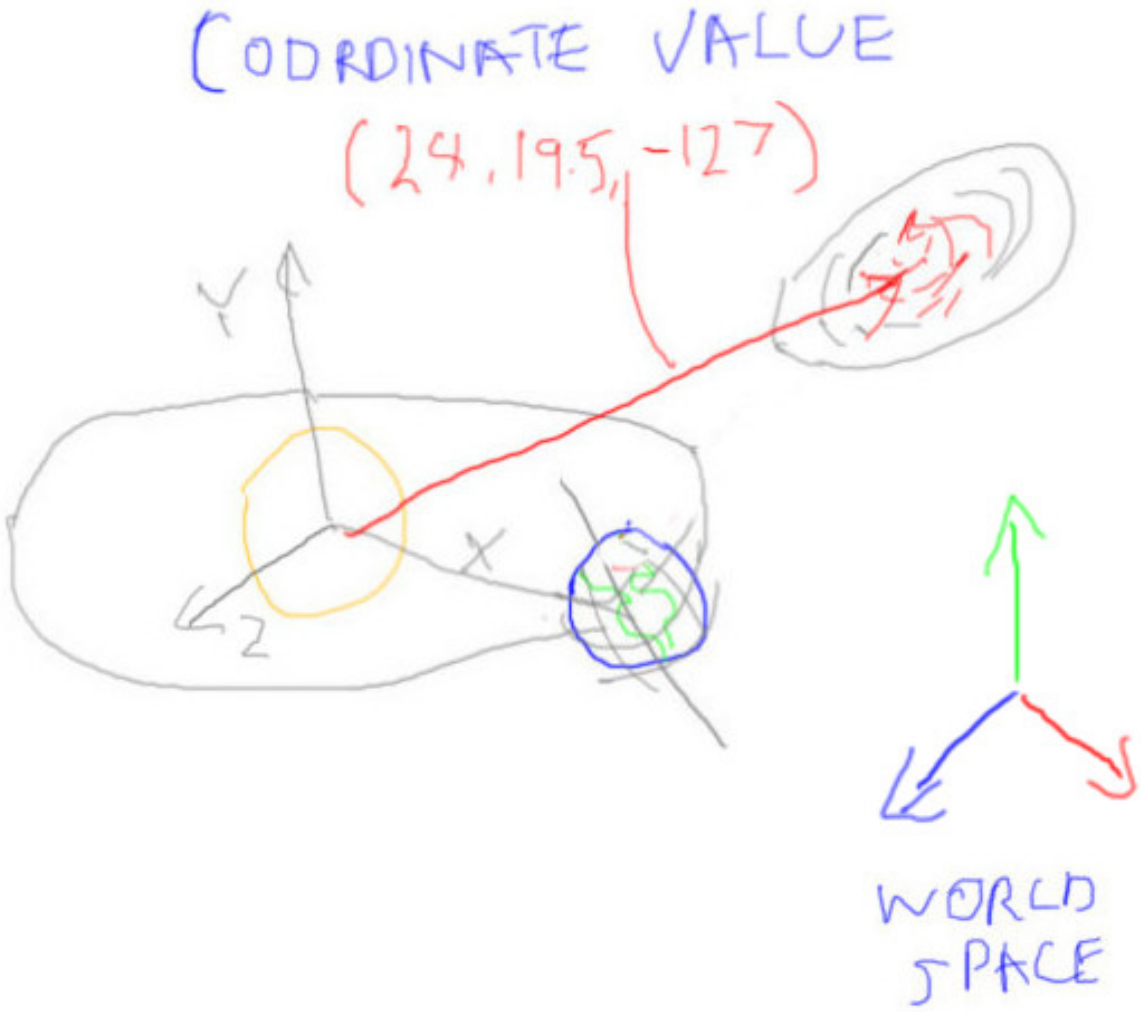
[add ***coordinate value*** and numbers]



Whenever we assign a point or vector a ***coordinate value***, it’s always with respect to some

frame of reference we've defined.

[add world space figure]



In computer graphics, we usually talk about the location and movement of objects, lights, and cameras in terms of ***world space***. This frame of reference is usually right-handed. You can define units for it if you like, such as 1 unit equals 1 meter, or it can be unitless if there's no need for them.

Question: Point and Vector Sum

In world space coordinates, given

[leave room at bottom for me to write the answer, in the next video]

P is a point (-7, -4, 14)
E is a vector (-4, 19, 18)
S is a vector (-5, 6, -10)

What is the new location of point Q = P + E - S

Point Q is at _____, _____, _____

Answer

The equation is:

[write to the right of the question]

Q = (-7, -4, 14) + (-4, 19, 18) - (-5, 6, -10)

When I do vector math, I don't not like double negatives, so I usually fold this minus sign into the coordinates themselves:

Q = (-7, -4, 14) + (-4, 19, 18) + (5, -6, 10) = (-6, 9, 42)

[in-joke 6*9 = 42
http://en.wikipedia.org/wiki/Answer_to_The_Ultimate_Question_of_Life,_the_Universe,_and_Everything#Answer_to_the_Ultimate_Question_of_Life.2C_the_Universe.2C_and_Everything_.2842.29]

Question: Vector Operation

[Go to documentation http://mrdoob.github.com/three.js/docs/55/#Reference/Math/Vector3 and slowly scroll through while talking]

Vector3

3D vector.

Example

```
var a = new THREE.Vector3( 1, 0, 0 );
var b = new THREE.Vector3( 0, 1, 0 );

var c = new THREE.Vector3();
c.crossVectors( a, b );
```

Constructor

Vector3(x, y, z)

Properties

.x

.y

.z

Methods

.set(x, y, z) Vector3

Sets value of this vector.

Three.js has many vector operations available. See the link to the documentation, which is given in the additional course materials http://mrdoob.github.com/three.js/docs/55/#Reference/Math/Vector3.

For the Vector3 class, which method changes a vector so that it points the same direction but it has a length of one? Here's a hint: this operation was used to make surface normals and the light direction vector each be one unit long.

Vector3._____()

Answer

.normalize() Vector3

Normalizes this vector.

The method's name is "normalize".

Note that this method changes the vector itself, in place. Some libraries will return a separate, new, normalized vector and leave the original vector alone.

Lesson: Translation

```
sphere = new THREE.Mesh(
new THREE.SphereGeometry( 104/2, 32, 16 ), sphereMaterial );
sphere.position.x = 0;
sphere.position.y = 540;
sphere.position.z = 0;
scene.add( sphere );
```

"***Translation***" is just a way to say "move something to a new position". In fact, in three.js we've been using translation for positioning parts of the drinking bird model.

Three.js has translation and other transforms built into every object. In this code we're moving the center of the sphere to the position ***(0, 540, 0)***, which is a vector. That is, any point on the sphere is first to be moved by this vector to a new position, then is rendered.

This code does the same thing:

```
sphere = new THREE.Mesh(
new THREE.SphereGeometry( 104/2, 32, 16 ), sphereMaterial );
sphere.position.x = 0;
sphere.position.y = 160 + 390;
sphere.position.z = 0;
scene.add( sphere );
```

[write below ***(0,160,0)*** ***(0,390,0)***]

It's pretty obvious that this y coordinate could be the sum or product of any set of numbers. My point here is that adding values together like this is equivalent to adding the coordinates of vectors. So here a vector (0,160,0) and a vector (0,390,0) are added together to get (0,540,0).

Translations can be treated somewhat like numbers in this respect. You can add them together in any order and you'll obtain a final translation vector that sums up all the translations and tells you the final position.

Lesson: Rotation

Three.js also has some built-in support for object rotation. The call is like this:

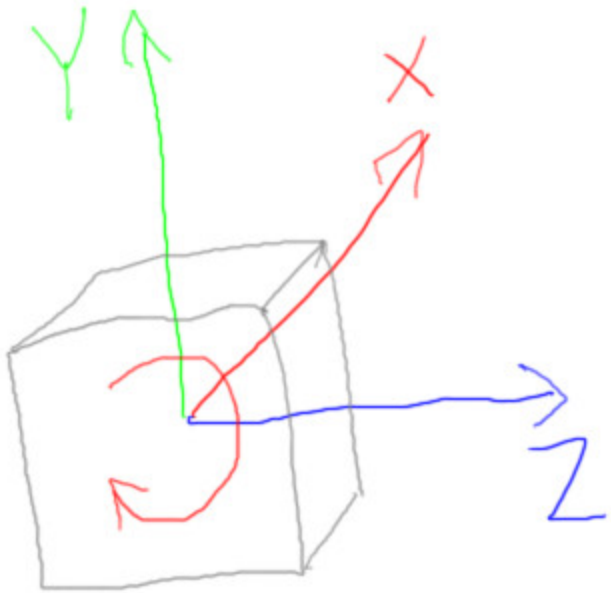
```
cube = new THREE.Mesh(
    new THREE.CubeGeometry( 64, 64, 64 ), cubeMaterial );
cube.rotation.x = 70 * Math.PI/180 ;
```

You can specify X, Y, or Z for the axis of rotation. The object will rotate around its center along that axis, in a counterclockwise direction. Remember that we're using a right-handed coordinate system, so the right-hand rule shows us the direction of the axis and direction of rotation.

[gesture with hand on screen the right hand rule]

The angle is specified in radians, which computers like. As a human, I like degrees, so here I'm specifying 70 degrees, then converting this number to radians by multiplying by pi over 180.

[XYZ system and a cube centered, show rotation direction.]



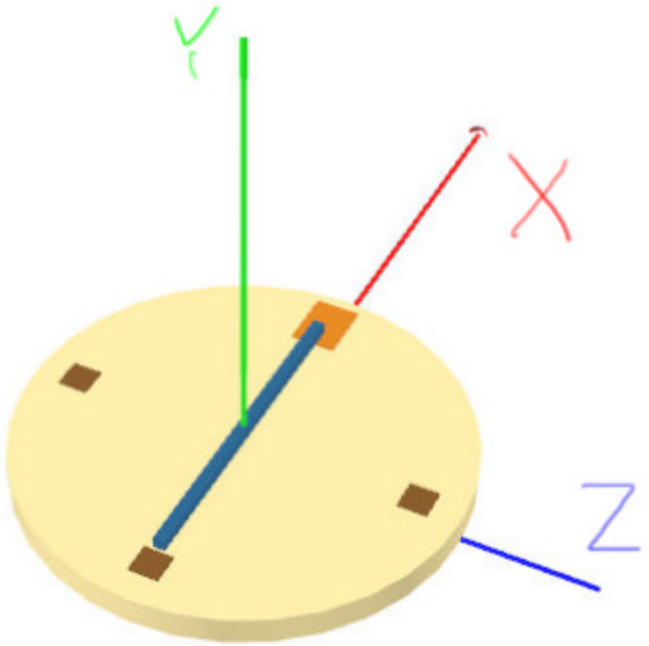
Here you can see the direction is counterclockwise if you were to point the X axis towards you. A way to remember this direction is the right-hand rule.

[Put hand in frame, pointing thumb into screen along X]

Point your thumb along the axis and the rotation angle will wrap in the direction of your fingers. This is the direction of rotation.

Exercise: Rotate a Block

[put this image to left (load the solution below in advance and turn off layer, so that we can the the margins right.]



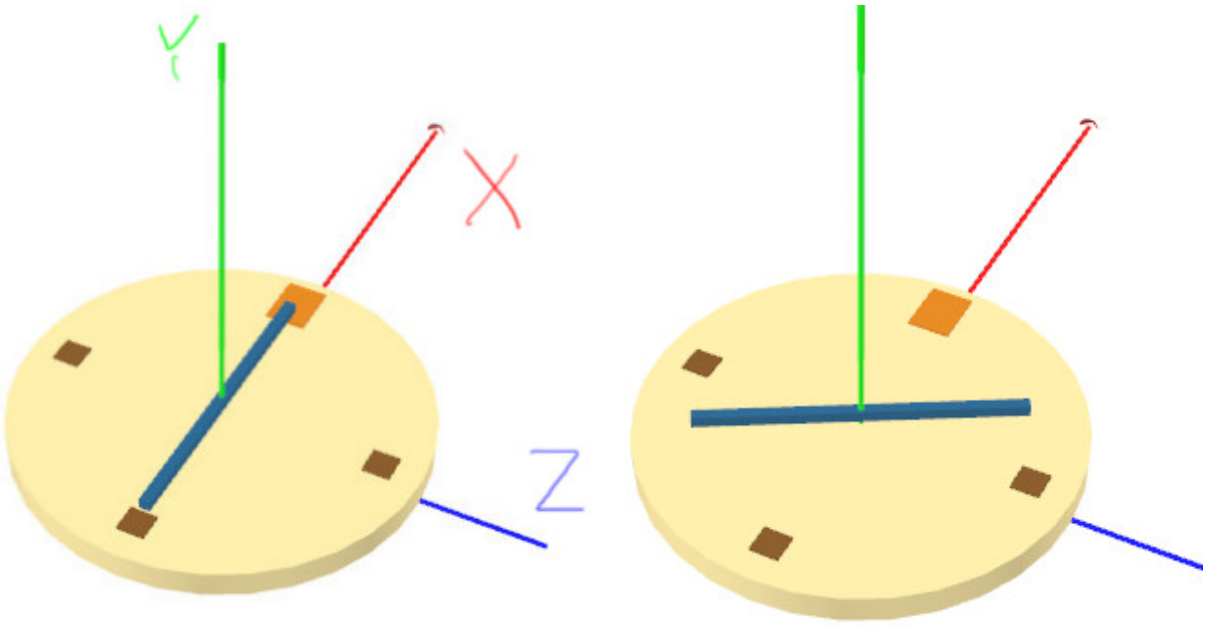
You'll start with a program that gives a model like this.

Notice that the X,Y, and Z axes are shown, given as the colors red, green, and blue, respectively [gesture].

Think of this as a clock hand, sort of. It points to 12 and 6 o'clock currently. We'll fix that later.

Rotate the hand of the clock so that it points to 2 and 8 o'clock, as shown here.

[add this one to right of original image]

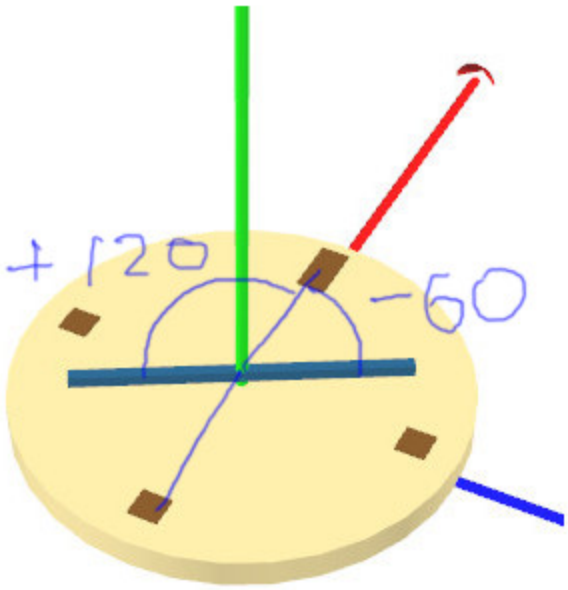


[exercise is at http://www.realtimerendering.com/udacity/?load=unit4/unit4-rotation_exercise.js]

Answer

[solution is at *redacted*]

[show answer above and then add information - the picture below is a bit out of sync, but I'll be writing on top of the correct one.]



The three elements you'd need to determine are:

[write just left column, then drawn in bits of info as I talk, to right of solution image]

axis	***Y***	
angle of rotation	***60***	***120***
sign of angle	***-***	***+***

The green axis is the one we want to rotate around, and this is the Y axis.

There are 360 degrees in a circle, we want to rotate 2/12ths of the way around this circle, so that's 60 degrees.

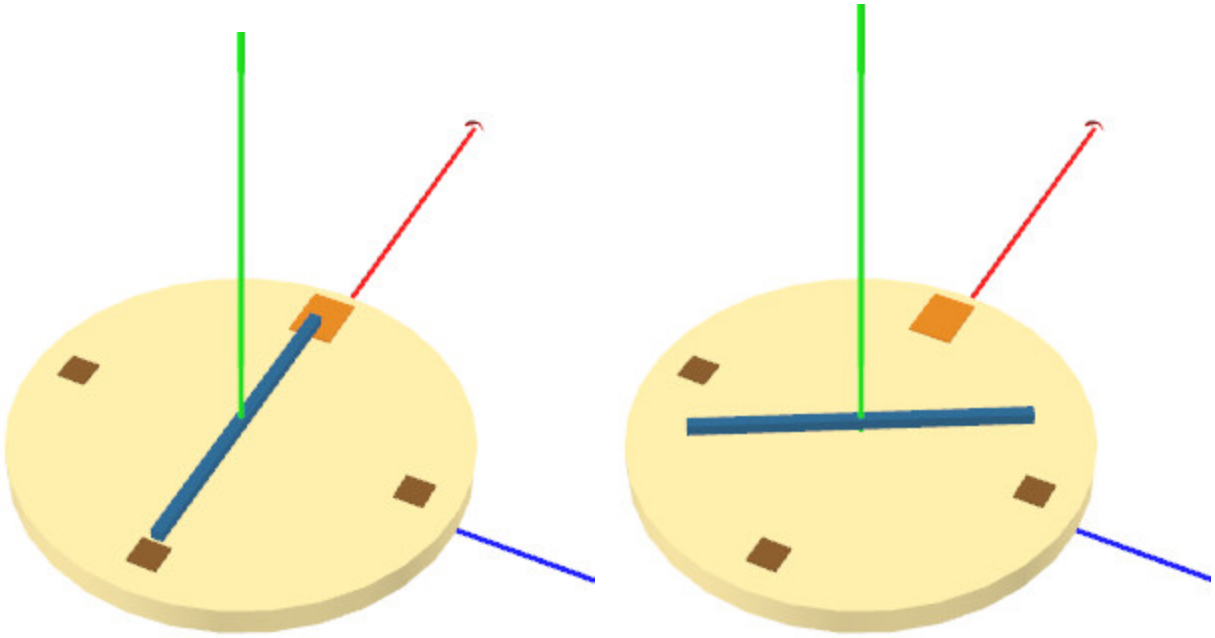
We also need to rotate in a negative direction, since the positive direction is counterclockwise.

```
cube.rotation.y = -60 * Math.PI/180;
```

This information can be saved in a line of code like this.

Since we can't tell which end is which, we could instead rotate 120 degrees counterclockwise and get the same answer, which would let us put in a positive angle.

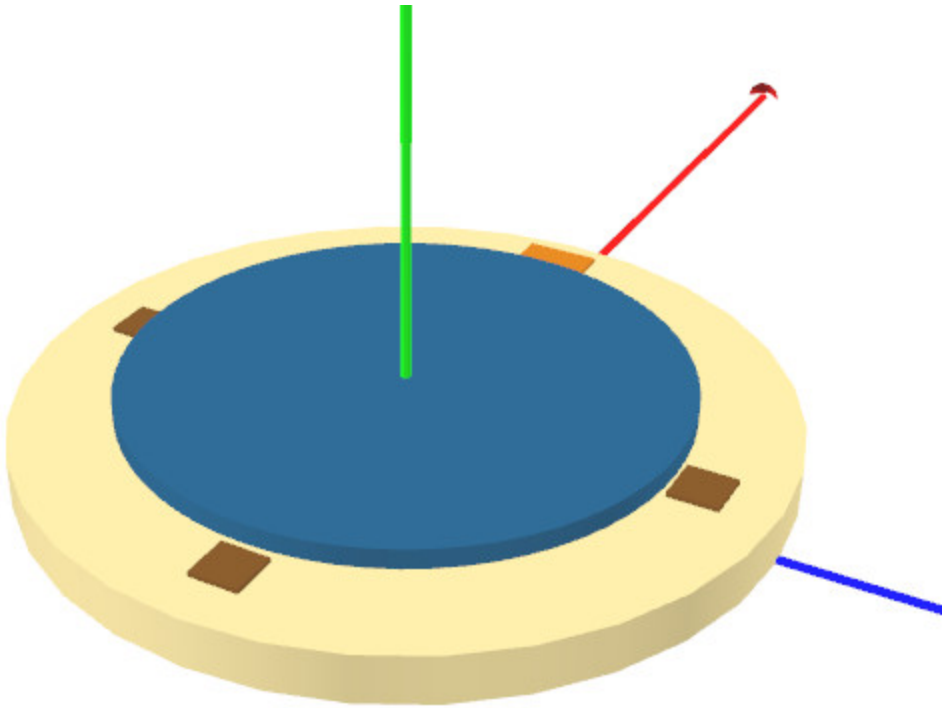
Question: Name That Object.



You saw the hand rotate from one position to another. Say you rotated the hand from 0 to 360 degrees. What sort of volume would it have swept through? That is, if you made a million hands, each one at a slightly different angle, what would they form?

() A sphere
() A disc
() A hemisphere
() A doughnut

Answer

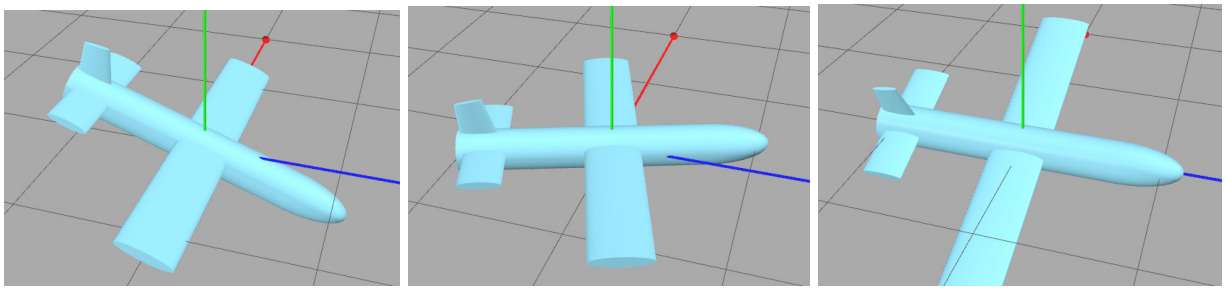


The answer is a disc-shaped volume, a pretty flat cylinder. Whatever axis you rotate around, each point will trace out a circle perpendicular to this axis, with a center at the origin.

It's useful to think about the axis of rotation this way. For example, if you have a location and want to rotate it to another location, find the circle that has its center at the origin and contains these two points. The circle's corresponding axis is the one to rotate around.

Lesson: Euler Angles

[Label each underneath, as shown: pitch/x-roll, yaw/head/y-roll, roll/z-roll. Put x-roll, etc. in different color beneath]



pitch
x-roll

yaw/head
y-roll

roll
z-roll

```
plane.rotation.x =  20 * Math.PI / 180;
plane.rotation.y = -12 * Math.PI / 180;
plane.rotation.z =  71 * Math.PI / 180;
```

Notice that there are three rotation angles on the three.js object. You can rotate around the X, Y, or Z axis, all together. If you're controlling a plane, the X rotation is known as the pitch angle, since it pitches the plane up and down.

[use pen to gesture, aligning it with the plane's body. Better yet, make a little plane out of a pen and some paper and use it.]

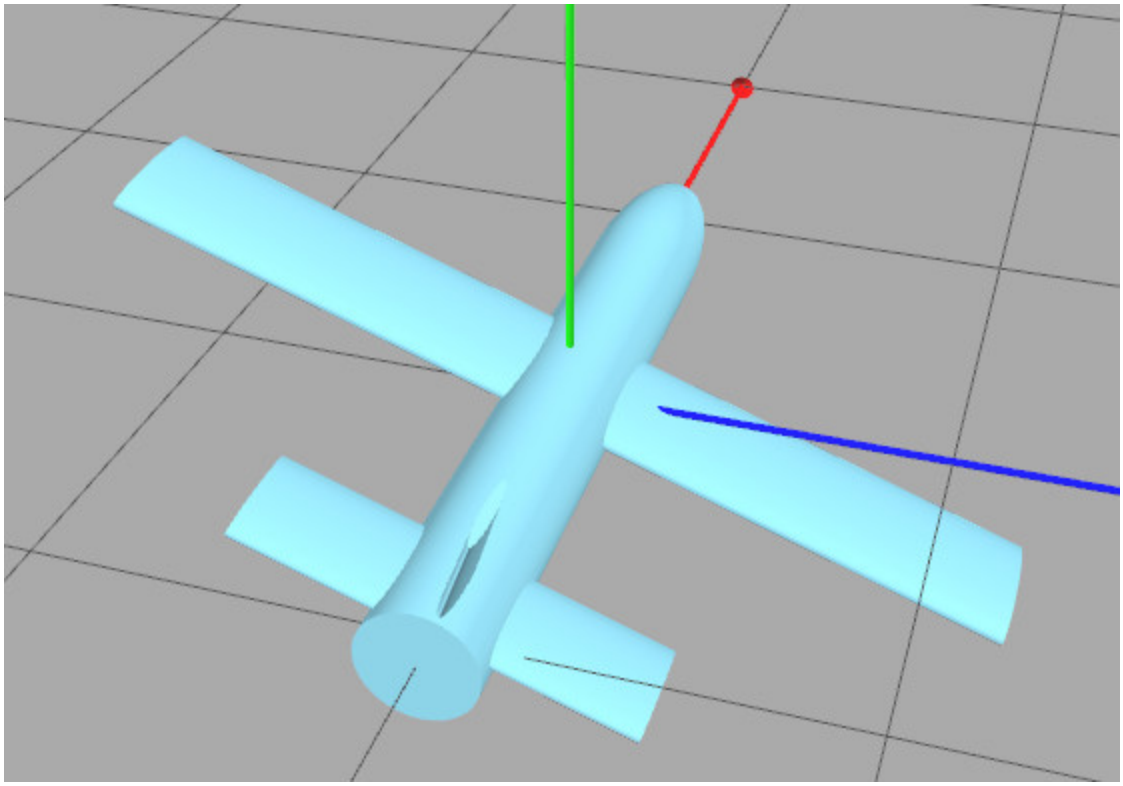
The Y is known as the yaw or head direction, since that's where the plane is head. Z is the roll, how much you're rotating along the axis of the plane itself. If you're using these angles in another context, they're sometimes called the x-roll, y-roll, and z-roll.

Leonhard Euler - e, 2.71828... is known as Euler's number. Invented Σ symbol.

These angles are called Euler angles when used together in this way. They are named after

Leonhard Euler, a Swiss mathematician and physicist who lived in the 18th century. He was one of the most prolific mathematicians of all time. If you have ever run across the number “e”, that’s called Euler’s number. Euler invented using the sigma sign for summation. Just listing out his accomplishments would take a whole unit at least, but you get the idea.

[start to run demo of Euler angles in action:
http://www.realtimerendering.com/udacity/?load=demo/unit4-euler_angles.js - this airplane could get remade for sure, right now it looks like a UAV drone, which doesn’t have the friendliest connotations!]



In three.js the rotations are applied to the model in the order Z, Y, and X. I should mention that this order can be different in different applications. For example, the order Z, X, Z is seen in robotics.

What happens if you rotate around two or more of these axes? The short answer is that the frame of reference changes for each rotation. This is actually easier to see than to explain.

[run demo while talking - that’ll be fun...]

Here I’m changing the X rotation, and you can see that the plane pitches up and down. Now applying the Y rotation, notice that we’re not rotating around the world’s Y axis, but rather around a plane that is defined by the X rotation. In other words, the Y rotation is applied with respect to

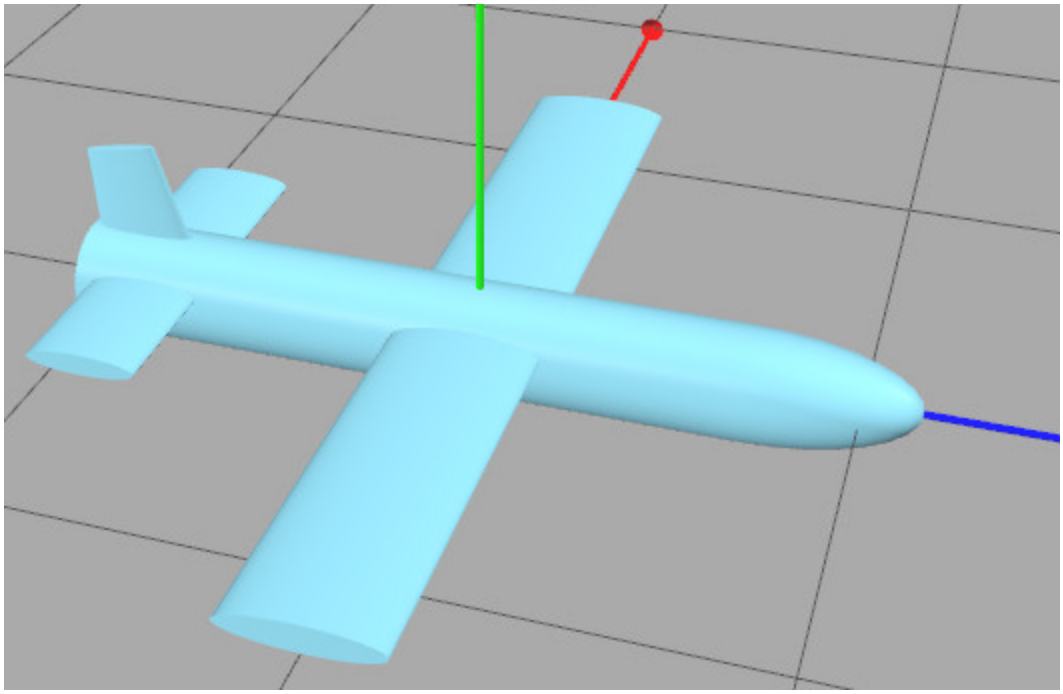
the plane's frame of reference, not the world's. In fact, all rotations are applied with respect to the plane. The initial X rotation just happened to be aligned with the world.

The Z rotation is also clearly changing with respect to the plane's orientation, rolling along its axis.

Euler angles are handy for flight simulators, robotics applications, and even mobile applications, as they can be used to describe the orientation of the mobile device itself.

However, Euler angles also can run into limitations, such as the problem of gimbal lock. For example, say I set the Y angle to 90 degrees. You can now see that the X rotation and Z rotation have exactly the same effect. We've lost a degree of freedom under these conditions, limiting how we can move away from this orientation.

[Euler angle demo put on screen:
http://www.realtimerendering.com/udacity/?load=demo/unit4-euler_angles.js]



I encourage you to play with the Euler Angles demo that follows and get a feel for the strengths and limitations of this method of setting rotations. Definitely try setting the Y angle to about 90 degrees to see gimbal lock in action.

[Additional Course Information: There's a lot more about Euler angles on Wikipedia, http://en.wikipedia.org/wiki/Euler_angle. If you want to play with Euler angles more and see how

they affect various objects, you might try the three.js interactive editor http://mrdoob.github.com/three.js/editor/ The Euler quote is from here http://en.wikipedia.org/wiki/Leonhard_Euler]

Demo: Euler Angles

[demo on screen: http://www.realtimerendering.com/udacity/?load=demo/unit4-euler_angles.js]

Lesson: Scaling

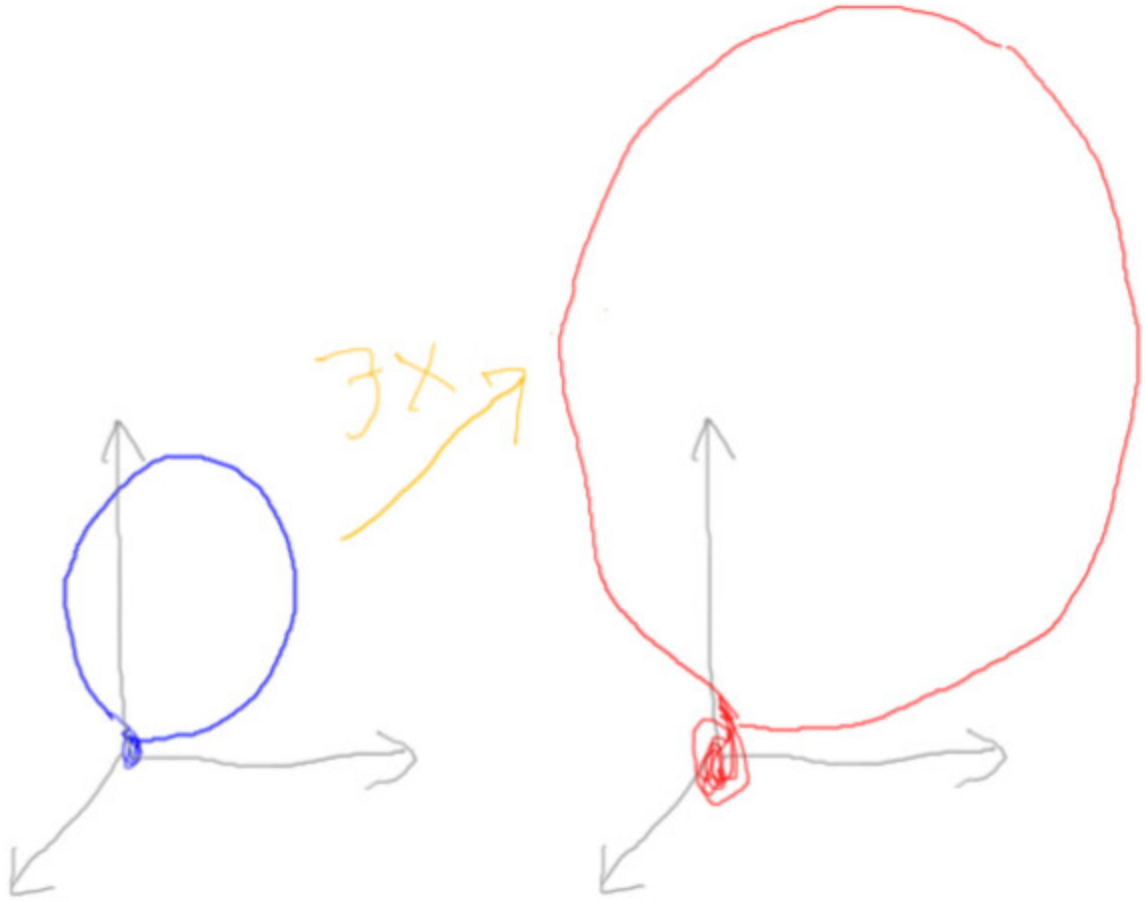
[***translate, rotate, scale***
rigid body
]

To scale something is to make it larger or smaller. This operation is somewhat different than translation and rotation. When we translate or rotate an object, we don't change its shape or volume. These two operations are called "rigid-body transforms".

The name "rigid-body" is just what it says: if you have an object and apply any number of rotations and translations to it, you won't change its shape. It's as if the object is made out of something, well, rigid, and will keep its form.

Scaling does change an object's overall size, so is not a rigid-body transform.

[draw balloon and 3x balloon]

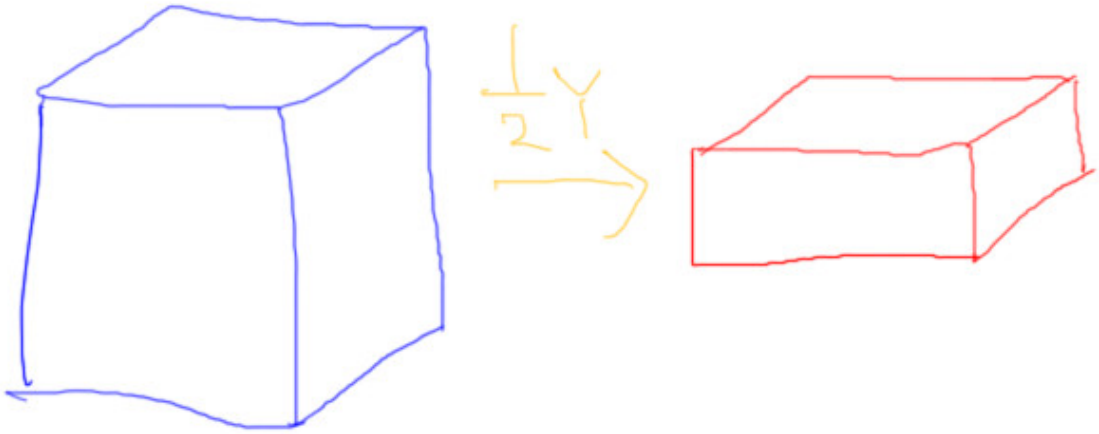



```
balloon.scale = new THREE.Vector3( 3,3,3 );
```

Scaling in three.js is simple enough. Just as there's a translation and rotation parameter on the object, there is also a scaling parameter.

Here's the code for to make a balloon object 3 times as large as it was before. Note that, unlike real life, this makes the balloon 3 times larger in all ways: the thickness of the skin, the part where you seal it off, and so on. Like rotation, scaling is done with respect to an origin. Since the valve of the balloon is at the origin, the valve will stay still and the balloon will expand upwards.

[draw cube squished vertically]

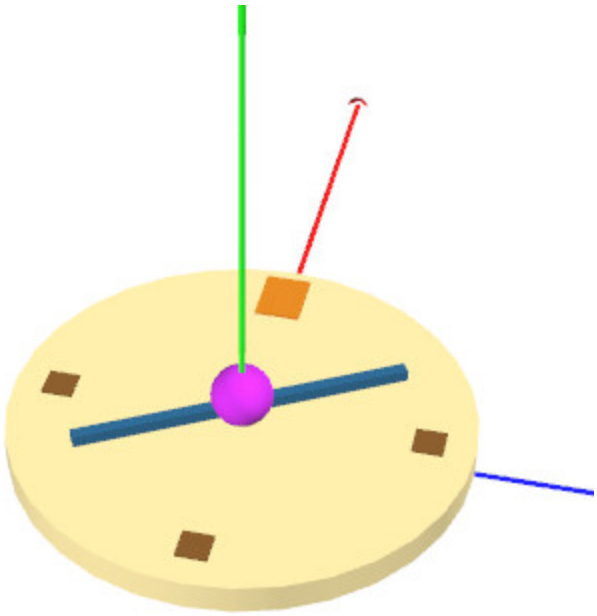



```
cube.scale.x = 1;
cube.scale.y = 0.5;
cube.scale.z = 1;
```

In fact, you can scale an object differently along each of the three axes, X, Y, and Z. This code is scaling the object along the Y axis but not the other two. A number less than 1 means that you're making the object smaller. The scale shown here has the effect of compressing it along this axis.

When you scale an object the same in all directions, this is called ***uniform scaling***. If the scale varies it's called ***non-uniform scaling***. Uniform scaling does not change any angles within the model itself. Non-uniform scaling can modify angles; in other words, the shape of the model itself is changed.

Exercise: Scale a Sphere

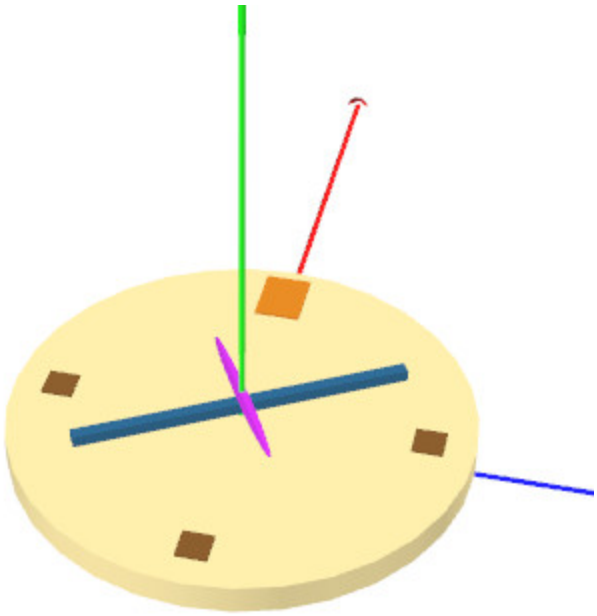


I want to add another indicator to my watch-like object. I've started by putting a magenta sphere in the scene, of ***radius 10***. I've positioned it correctly, but it's not looking at all like a hand.

Your job is to stretch and squish this sphere into something that looks kind of like a clock hand, but centered at the origin like the other hand. I want the hand to be

hand: 60 units long
4 units high and wide

Use the scale parameter to modify the sphere. Once you've properly scaled this sort-of-an-hour hand, rotate it so that it points at 11 and 5 o'clock.



When you're done things should look like this.

[exercise at http://www.realtimerendering.com/udacity/?load=unit4/unit4-scale_exercise.js]

Answer

[solution at *redacted*]

```
sphere.scale.x = 3.0;    // 60 / (2 * radius 10 ) -> 3
sphere.scale.y = 0.2;    //  4 / (2 * radius 10 ) -> 0.2
sphere.scale.z = 0.2;
sphere.rotation.y = 30 * Math.PI/180;
```

The axis to stretch along is red, which is the X axis.

The one tricky bit is to realize a sphere of radius 10 is 20 units across. If we want to go from 20 to 60 units, the scale along the X axis needs to be multiplied by 3. That is, 60 divided by 20 gives 3.

To go from 20 units to 4 units, the other two axes must be multiplied by 0.2.

For the rotation, we do as before, rotating around the Y axis. 30 degrees corresponds to one hour of movement.

Lesson: Translation, Rotation, and Scale

Up to this point we have been mostly ignoring the order of operations. That is, we've been merrily performing ***translations, rotations,*** and ***scales*** without worrying about whether one needs to be done before the other. After all, we found that translations can be done in any order and these still add up to a single translation. What could go wrong?

[list order below here: ***scale, rotate, translate***]

The answer is, plenty. Order matters when rotations or scales are involved. Three.js uses the following order to apply the transforms on an object:

scale, rotate, translate

```
var sphere = new THREE.Mesh(
new THREE.SphereGeometry( 10, 32, 16 ), hourHandMaterial );
sphere.scale.x = 3.0; // 60 / (2 * radius 10 ) -> 3
sphere.scale.y = 0.2; //  4 / (2 * radius 10 ) -> 0.2
sphere.scale.z = 0.2;
sphere.rotation.y = 30 * Math.PI/180;
sphere.position.y = 18;         // move the hand above the other hand
```

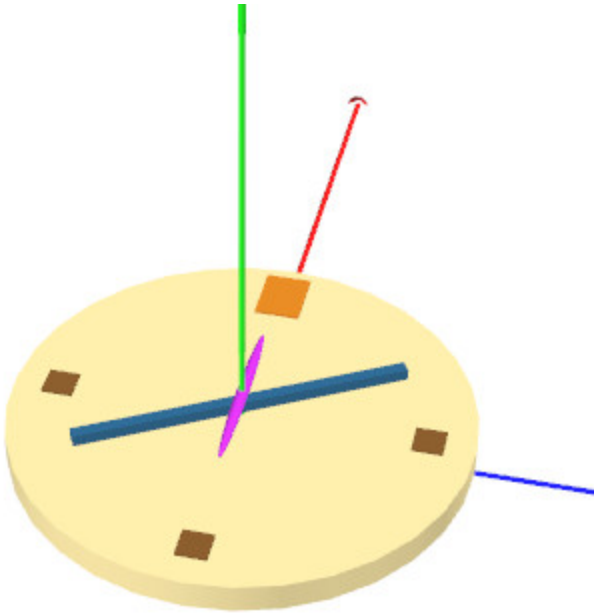
```
var sphere = new THREE.Mesh(
new THREE.SphereGeometry( 10, 32, 16 ), hourHandMaterial );
sphere.scale.x = 3.0;    // 60 / (2 * radius 10 ) -> 3
sphere.scale.y = 0.2;    //  4 / (2 * radius 10 ) -> 0.2
sphere.scale.z = 0.2;
sphere.rotation.y = 30 * Math.PI/180;
sphere.position.y = 18; // move the hand above the other hand
```

In the scaling exercise three.js first scaled the clock hand made out of a sphere, then rotated it to position. Finally, it positioned this hand to be above the other hand. It doesn't matter what order you set these parameters; three.js always evaluates them in the order scale, rotation, position.

This is the default because it's often the easiest way to produce the results you want. If you have something special in mind, you may find it best to transform in a different order. That's possible, and in fact we'll later see how we need to use a different order to make our clock example look really good. For now, though, we'll stick with scaling, then rotation, then translation, in that order.

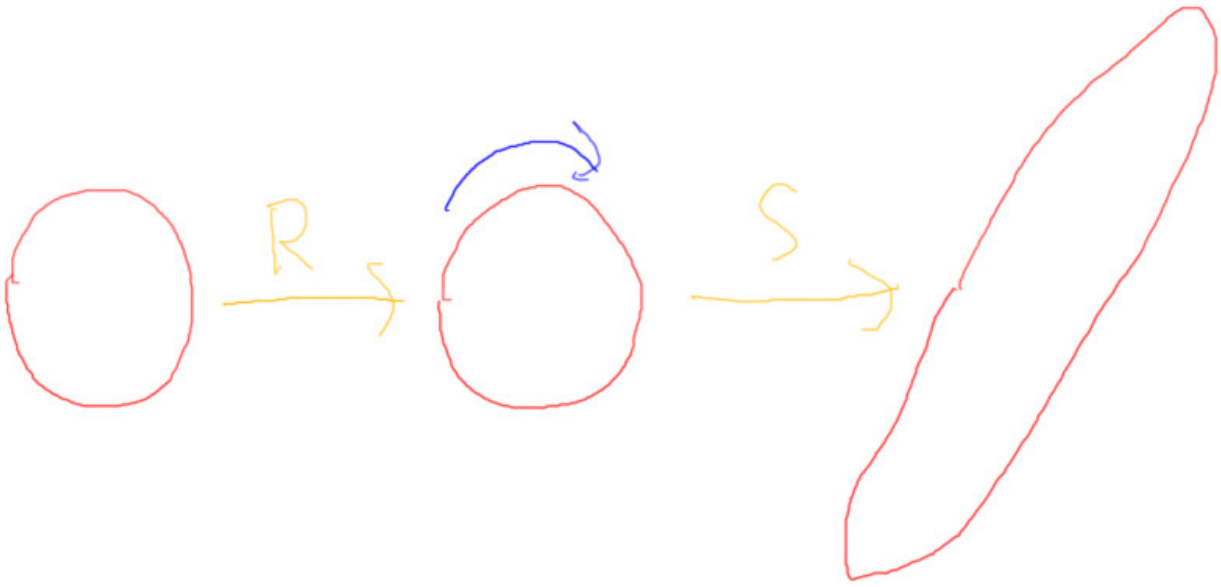
Let's see what would happen if the order of rotation and scaling were reversed. That is, first we'd rotate, then we'd scale.

[made with http://www.realtimerendering.com/udacity/?load=demo/unit4-rotate_then_scale.js]



For our clock exercise, the result would look like this. The rotation seems to have had no effect!

[draw sphere, rotated sphere, stretched sphere]

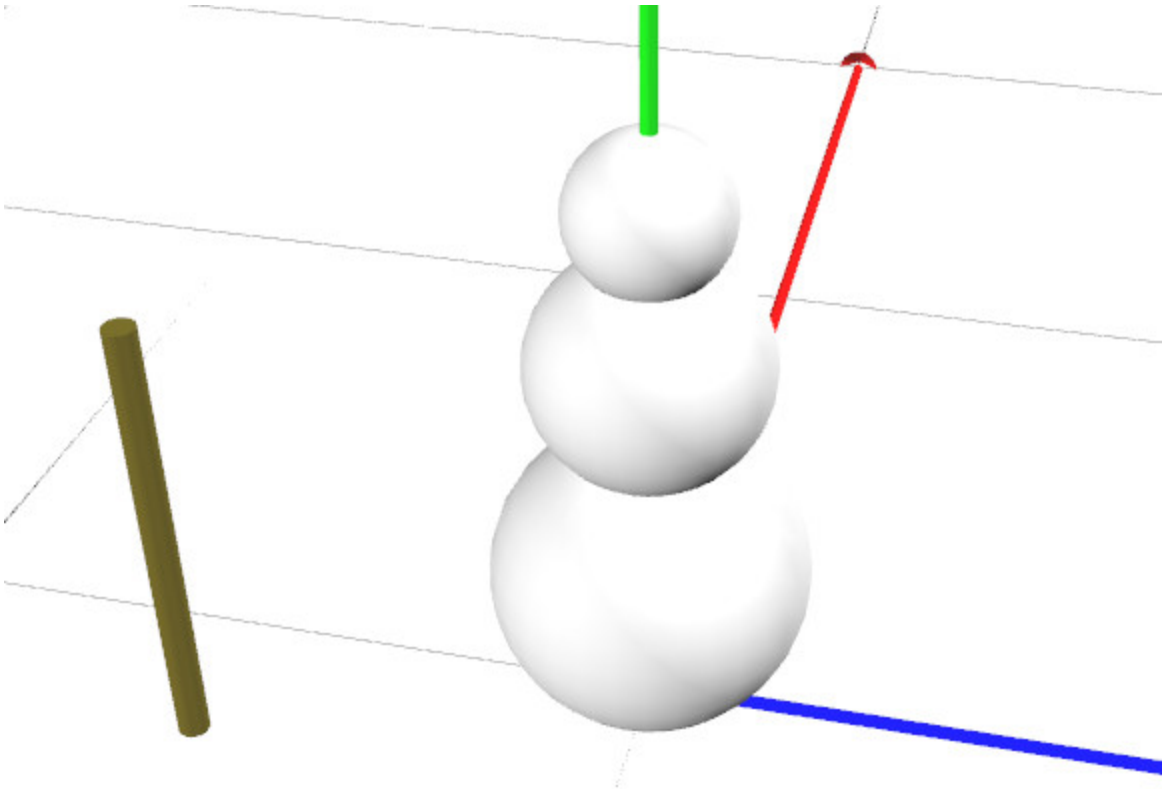


Think about what the order of operations actually does to the object. First we rotate the sphere. This has little effect on what we see on the screen - a sphere looks pretty much the same from any angle. So by doing the rotation first, we've essentially lost its effect. Stretching the sphere afterward gives us the shape we want, but the rotation's already been done.

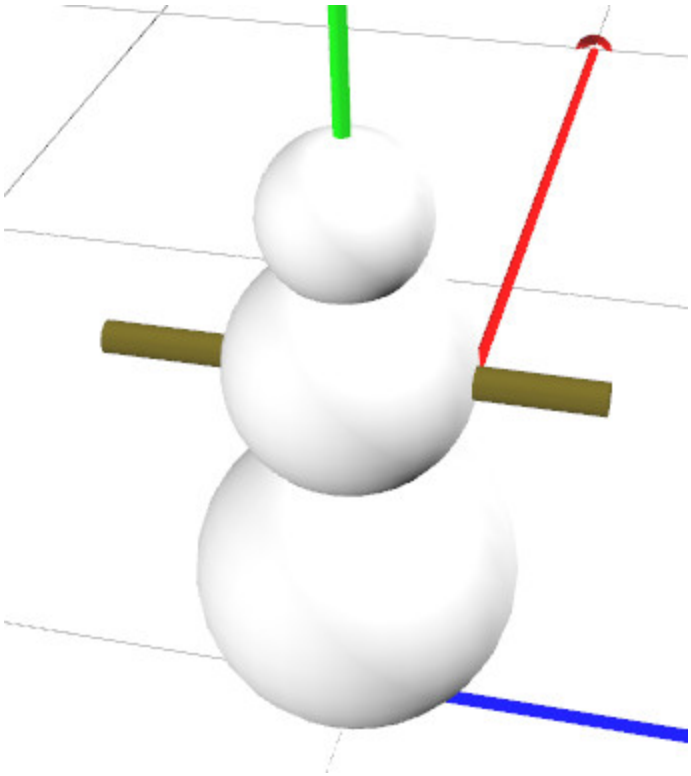
At this point I should show you a great program for experimentation. See the additional course materials for a pointer to the

[Additional Course materials: The three.js interactive scene editor is at http://mrdoob.github.com/three.js/editor/. First define a directional light, then add an object to the scene.]

Exercise: Build a Snowman



Let's build a snowman using rotation and translation. I did the first part, stacking up the snow.



Your job is to position the pole through the snowman and give him arms. The center of the pole should be placed 50 units above the ground. When you're done you should see something like this.

[Exercise at http://www.realtimerendering.com/udacity/?load=unit4/unit4-snowman_exercise.js]

Answer

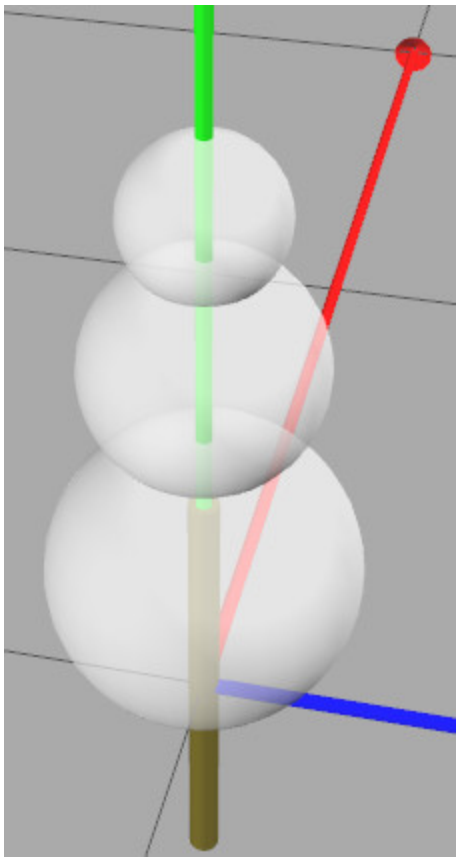
[Solution at *redacted*]

[Keep solution image on?]

The answer is that you rotate the pole to be horizontal by rotating it along the X axis 90 degrees, then move it up 50 units along the Y axis.

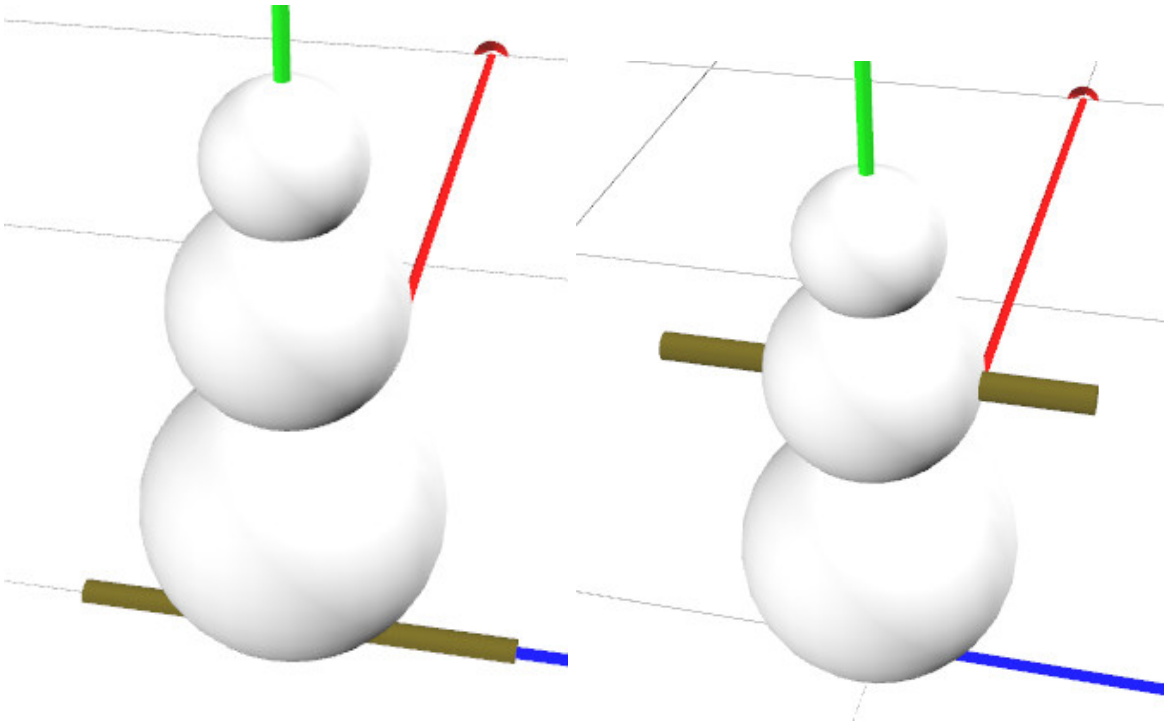
```
var cylinder = new THREE.Mesh(
    new THREE.CylinderGeometry( 2, 2, 60, 32 ), woodMaterial );
cylinder.rotation.x = 90 * Math.PI/180;
cylinder.position.y = 50;
```

Lesson: Rotate, then Translate



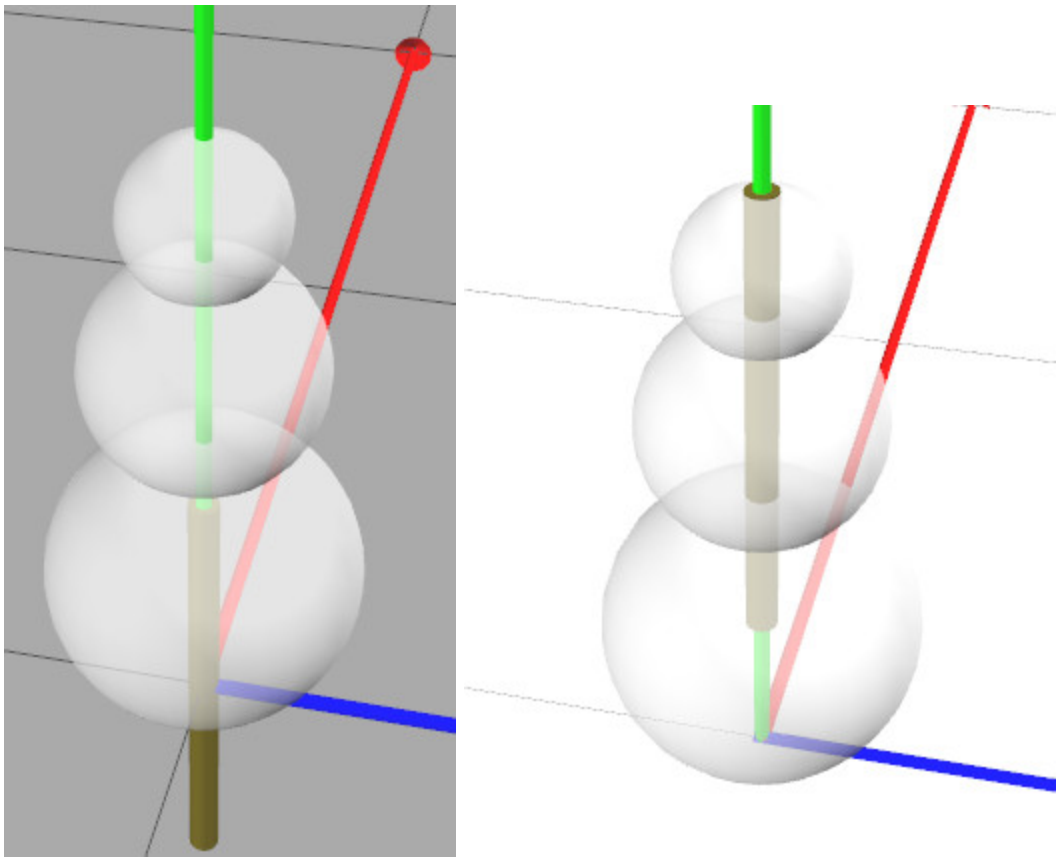
[Image generated by unit4-snowman_half_bad.js - the students will not use this program at all]

We saw earlier how scaling, then rotating is often the most convenient order. With our snowman the arms were positioned by rotating, then translating. Here's the stick without any transforms on it. I've made the snowman transparent and removed the ground plane so we can see the stick's position. It's actually inside the snowman, halfway through the ground. Whenever you create most geometric objects in three.js the object is centered around the origin.

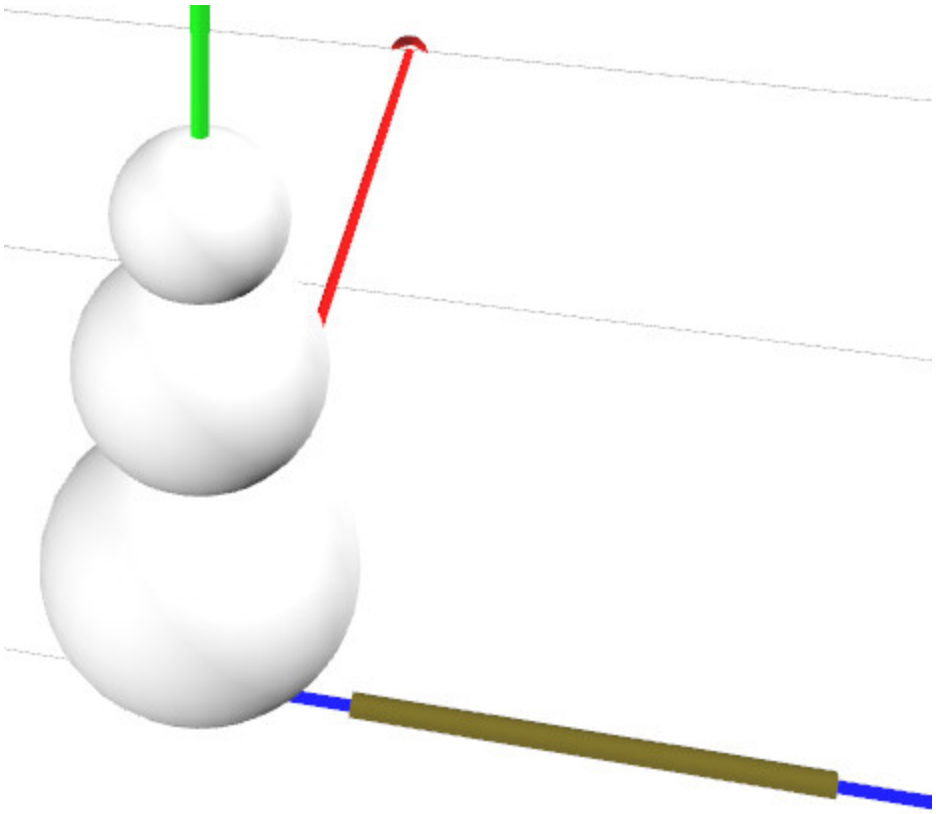


The rotation along the X axis put the stick into its proper orientation. Then the translation moved it upwards in world coordinates to the proper location.

Let’s see what would happen if we first translated, then rotated.



First we move the stick upwards 50 units. The center of the stick is now in place in the middle of the body - so far, so good.

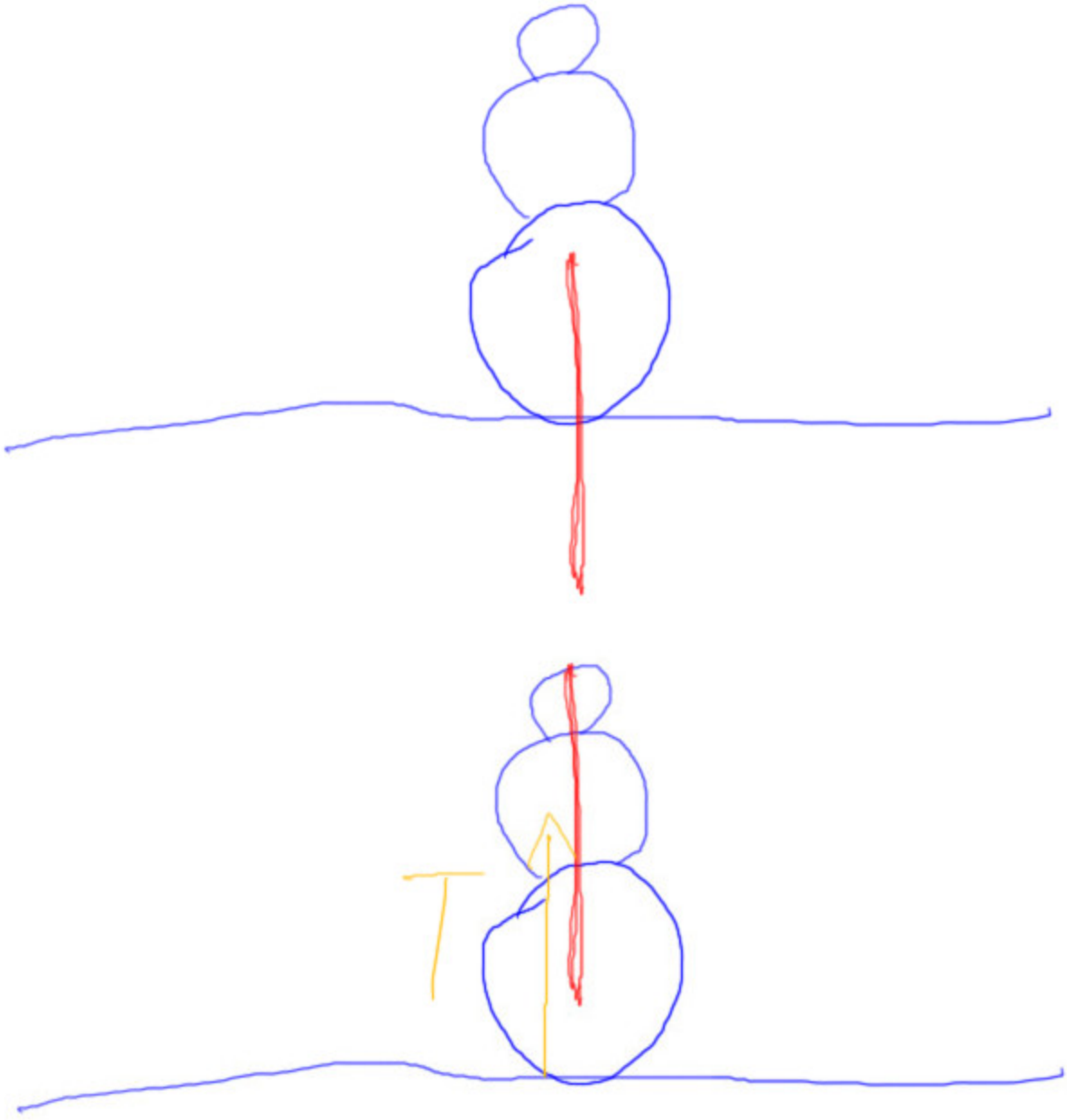


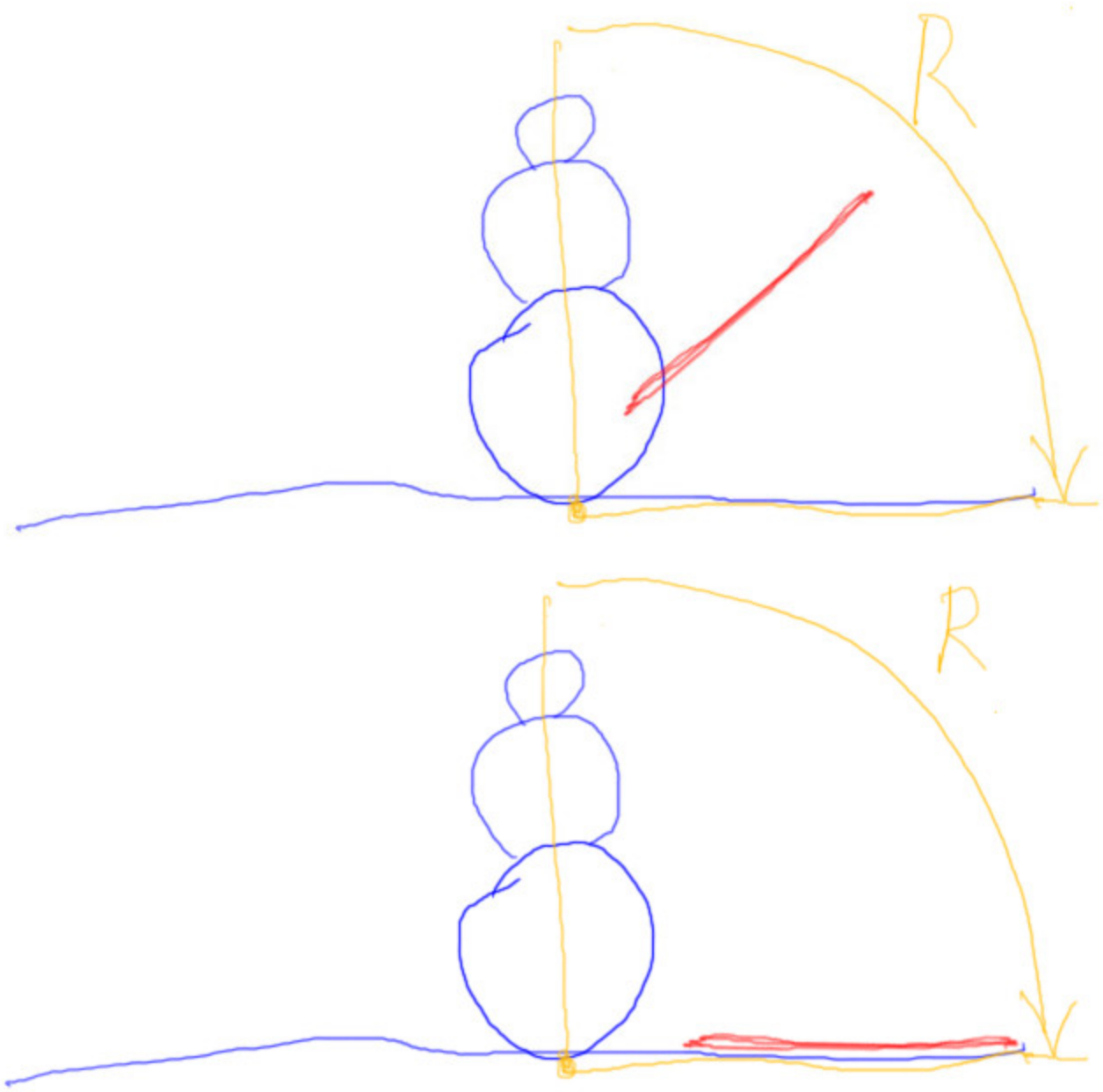
[Generated by unit4-snowman_whole_bad.js - not a program the student will use.]

Now if we rotate the stick after positioning it, we suddenly see it lying on the ground. What happened?

The problem is that rotation takes place with respect to the origin. In the first step we moved our stick up above the origin. In the second we rotated the stick, but it rotated around the origin.

[Draw side views, USE LAYERS FOR EACH ELEMENT: stick in place; stick moving upwards so its center is at 50 units above; stick rotating to its final position. Label each!]


T



Let's watch this disaster in slow motion. The stick starts centered at the origin. We move it upwards 50 units. Now its center is at the right height. However, when we then apply a rotation, the object rotates around the origin down here instead of rotating around its center.

Another way to think about it is that translating an object moves its center. You're moving the object up. Relative to the stick, you're moving the point it will rotate around down.

We've established that scaling before rotating is usually what we want, and rotating before translating is also generally more useful. This is why three.js uses this order: ***scale, rotate,***

translate, when dealing with a single object.

Question: How To Position a Clock Hand

[draw a simple clockface, circle, hour points, and a single hand; put hand in two positions.]

Think about our boxy clock hand. We want it to rotate around something else than the center point of the hand itself - we want it to rotate around the axis through center of the clockface. One end of the hand wants to be near the origin.

What's the order of operations that will work and is simplest to use?

( ) Rotate, then translate
( ) Translate, then rotate
( ) Scale, rotate, translate
( ) Translate, rotate, translate

Answer

The first and third answers are how three.js normally performs its operations: scale, rotate, translate. However, this order doesn't work for the clock hand: we in fact want to translate to position first. So three.js's basic methods won't work.

The fourth answer will work, but it's not the simplest to use. The second translation is not needed.

The second answer of "translate, then rotate" is correct. We want to translate the clock hand so that the point we rotate around is towards the end of the hand. Once the object is translated to this position, we can correctly rotate around its origin in a sensible way.

Lesson: Object3D

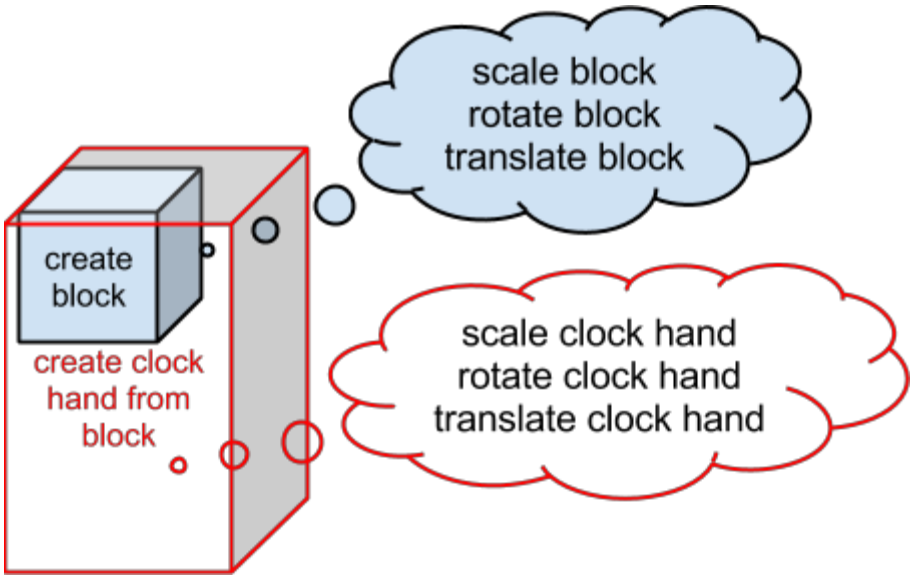
We've seen that three.js can't easily let us position and then rotate the hand of a clock in place. The problem is that three.js rotates, then positions, when we'd like to do these in the opposite order.

One simple solution that three.js provides is to use ***Object3D*** to make a new object that contains our clock hand. Here's the code for how to do this for a clock hand:

```
var block = new THREE.Mesh(
    new THREE.CubeGeometry( 100, 4, 4 ), clockHandMaterial );
block.position.x = 40;

var clockHand = new THREE.Object3D();
clockHand.add( block );

clockHand.rotation.y = -70 * Math.PI/180;
scene.add( clockHand );
```

What is happening here is that the block is nested inside the clockHand object. The translation moves the block so that one end is over the center of the clockface, so that the hand will rotate around the clock properly. By putting this object into the clockHand, I'm saying I want to use the block in its current position. I can then apply an additional scale, rotate, and translate. In this case I just want to rotate. The final line of code adds the clock hand object to the scene.

That's one way to look at Object3D: it adds more transforms into the list that you can then use. We now have some six transforms that we could set in this system. Here I've written it out:



scale block
rotate block
translate block
create clock hand from block
scale clock hand
rotate clock hand
translate clock hand

Here's a more compact way to write out the order of transforms:

Th Rh Sh Tb Rb Sb O
< ----------
[that is, draw arrow underneath showing order]

This looks a bit crazy, I've put the order of the transforms from right to left. There's a reason for that, and we'll talk about it when we get to matrices. For now, just believe me that this is how most computer graphics texts show a series of transforms.

O is the object we're transforming, which in this case happens to be a box. Reading from right to left, we apply the box's scale, rotation, and translation. By inserting an Object3D in our code, we get an additional three transforms we can apply in turn. That is, we can use the clock hand's scale, rotation, and translation.

[***TRaSh]***

The way I remember this order is by the word "***TRaSh***". There's TRS in that word, in the proper *notational* order. I guess I could use the word "Trees" instead and not have to think about the "h". But I like "trash" as it reminds me of the TRS-80, called a "trash 80" for short, one of the first personal computers, from back when programs were stored on cassette tapes. Once upon a time this computer ruled the earth: in 1979 it had the largest selection of software for any PC.

TRS, then, is the order to remember. In our case, just the box's translation and the clock hand's rotation are used:

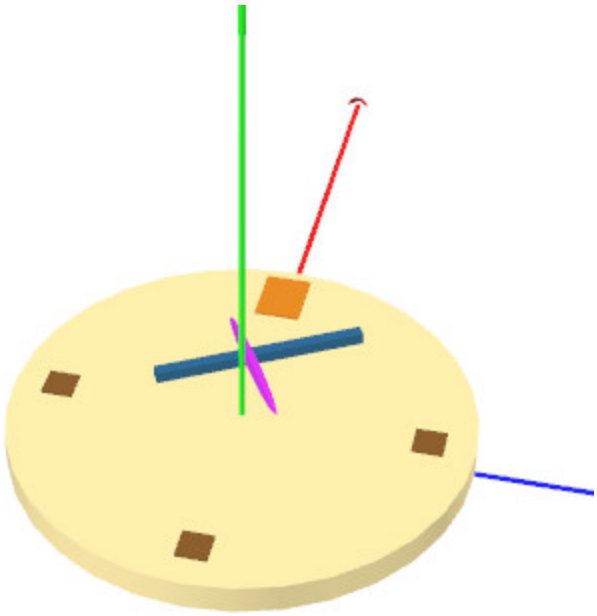
[erase the unused transforms]

Th ***Rh*** *Sh* ***Tb*** *Rb Sb* ***O***

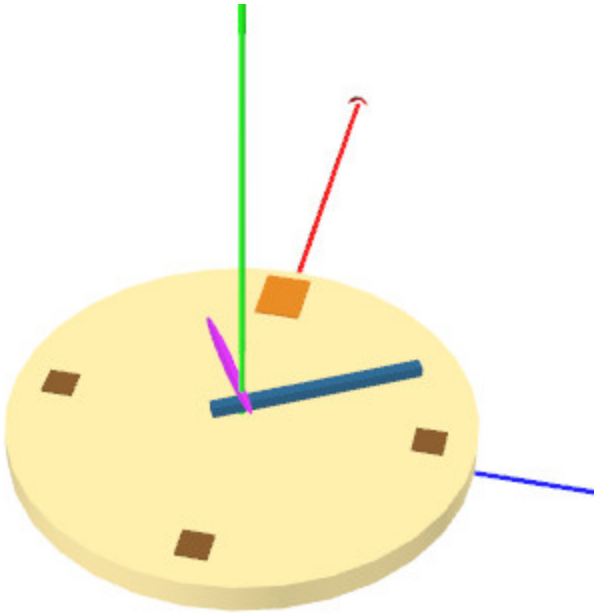
This gets us the combination we want: translate so there's a new origin, then rotate around this origin. Adding a whole object3D to get a single additional transform seems like overkill, but really it doesn't add much code.

[Additional Course Materials: the TRS-80 is described on Wikipedia http://en.wikipedia.org/wiki/TRS-80, and here http://oldcomputers.net/trs80i.html.]

Exercise: Two Clock Hands



If you try to take a box or sphere clock hand from the previous exercises and try to position and rotate it, you'll get something like this. Three.js is doing the standard rotate, then translate series of operations. The hands are rotated to the proper angles, but the translations don't move them along their axes but rather move them in world space.



The positions and scales in the exercise are all correct, you shouldn't need to change those. You need to figure out how to rotate the hands so they are properly oriented. When you're done you should see the hands as shown here.

[exercise at http://www.realtimerendering.com/udacity/?load=unit4/unit4-clock_exercise.js]

[**TODO** Gundega: what might be really cool is to add two sliders here to the code. Namely, one for each hand, of the rotation angle (in degrees). People then can see the wacky things that go on when you rotate, then translate in this way, and can test their own answers. We can also test their answer by giving different angles in our test, to make sure they're not cheating. Easy to add, I just need to move on...]

Answer

[solution at *redacted*]

You'll need to use an Object3D to let you rotation after translation. Here's my solution for the minute hand:

```
cube = new THREE.Mesh(
    new THREE.CubeGeometry( 70, 4, 4 ), minuteHandMaterial );
cube.position.y = 14;
cube.position.x = 70/2 - 10;

var minuteHand = new THREE.Object3D();
minuteHand.add( cube );

minuteHand.rotation.y = -60 * Math.PI/180;
scene.add( minuteHand );
```

The cube is translated to the correct pivot point by the position - that doesn't change. A minuteHand is then added and rotated into place. This new object is then added to the scene. It's this minuteHand object that undergoes the rotation and gives us what we want. Similar code is added for the hour hand.

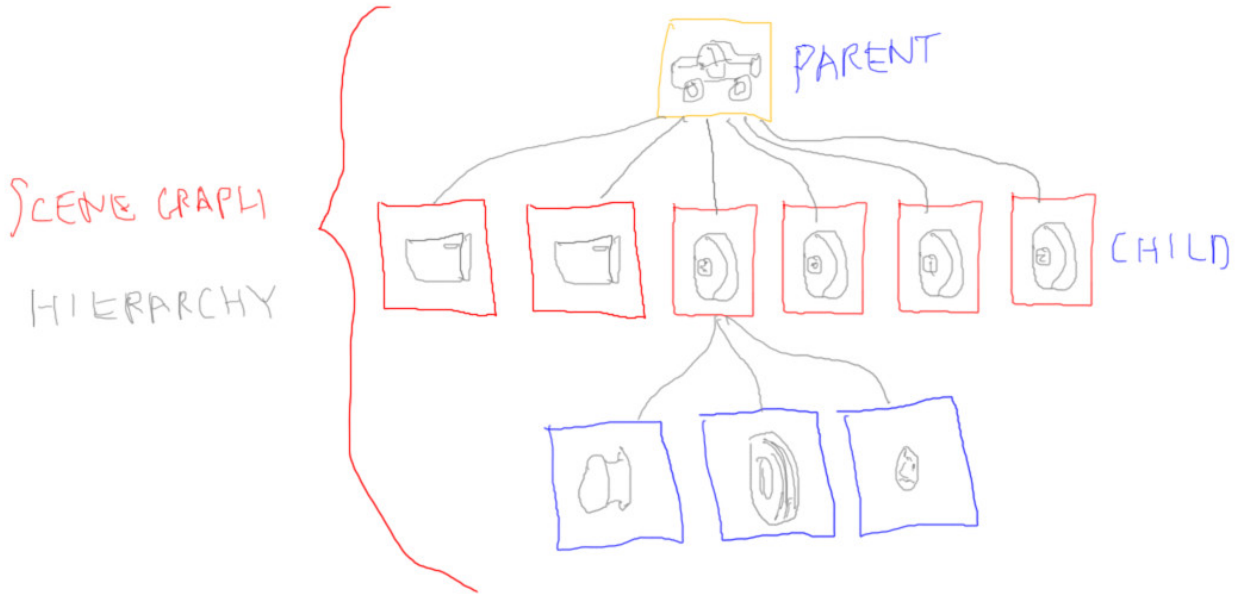
Lesson: Hierarchy of Objects

[***parent-child*** Draw two boxes, one above another, different colors each level, gray connector. See drawing below, start with them]

We've used Object3D to give us access to a few extra transforms in the chain. However, Object3D is designed for another purpose that is extremely useful. What Object3D does is create a parent-child relationship between two objects. Once an object is a child of another object, that child is affected by whatever is done to the parent.

[***scene graph*** ***hierarchy*** add in the rest: drawing of scene graph and real object, e.g. a car, two doors, four wheels, within a wheel is rim, tire, hubcap. Use different colors at each level, connect boxes with gray]

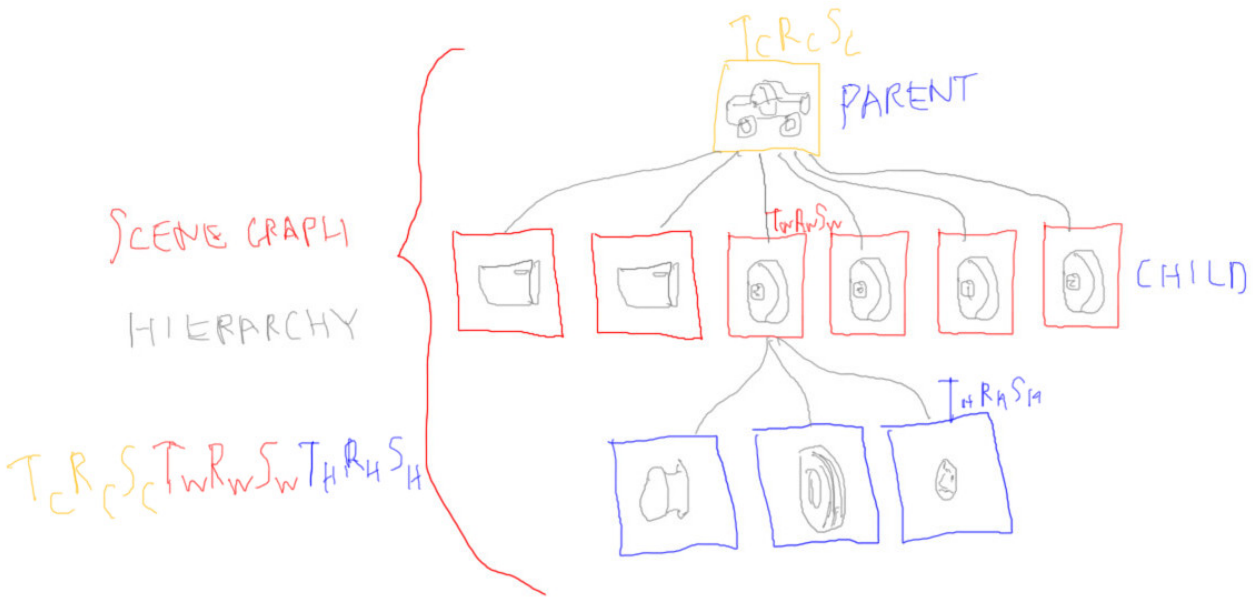
[WHEN I DRAW THIS FOR REAL: put hubcap under wheel instead of tire, as we'll reuse the three objects one over the other.]



Here's a more fleshed-out set of parent-child relationships. The car has two doors and four wheels. Each wheel has a rim, a hubcap, and a tire. This is usually called a "scene graph", as it defines a graph of relationships of objects in the scene. This whole tree structure is called the "hierarchy" (another frequently misspelled word!).

With this hierarchy we can do a lot. For example, we can move the whole car by applying a transform to the body. The other objects in the hierarchy will automatically be updated with this transform and follow along with the body. We can also affect objects within the hierarchy. If I change the wheel's rotation, the rim, hubcap and tire will also be updated and move along.

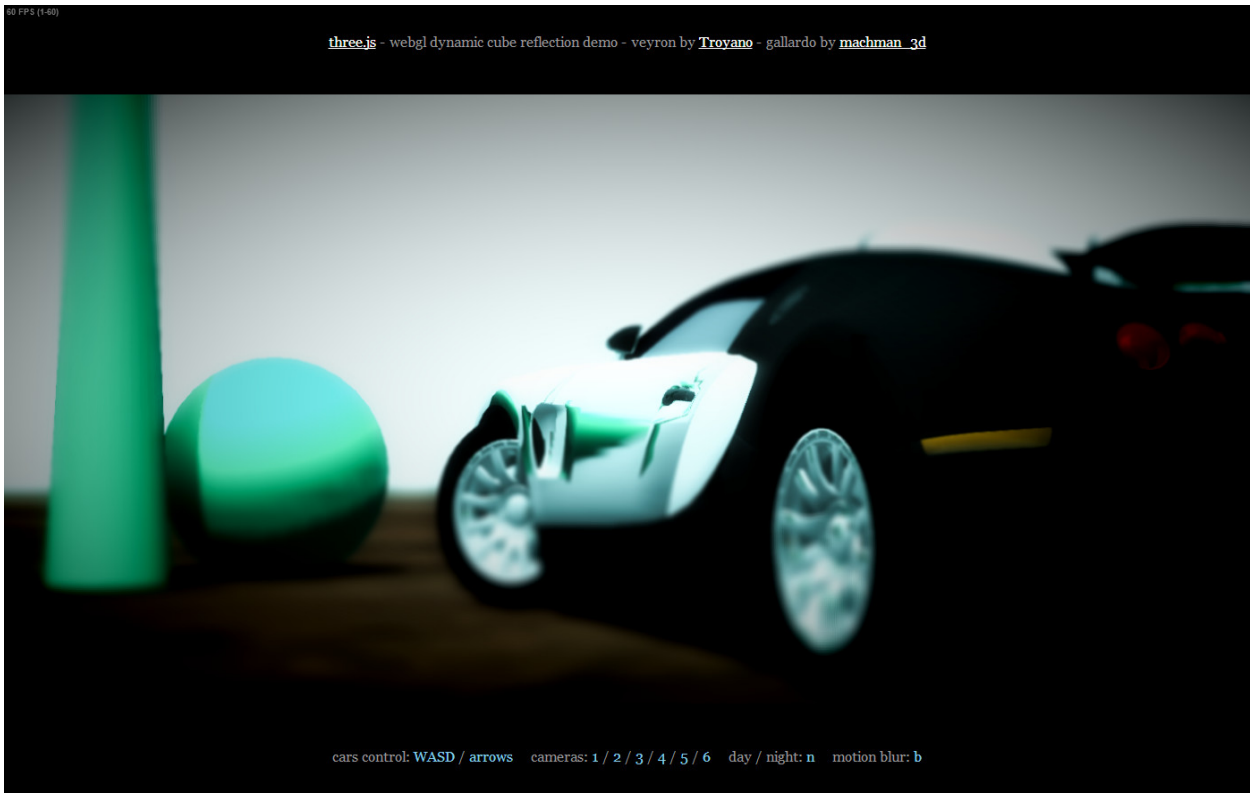
[Add TcRcScTwRwSwThRhSh series in colors, correlate to hierarchy]



Under the hood what is happening is that a series of transforms are being maintained. For example, for the hubcap here's the series: there's the scale/rotate/translate set for the hubcap itself, then another SRT series from the wheel, and finally an SRT set from the car as a whole. In practice the scale matrices are often not used, as many model creators make their models to scale.

The good news is that these complex series of transforms can be compressed into a single transform that does it all: it holds all scales, rotations, and translations, plus anything else done. We'll show how this works in the upcoming lessons on matrices.

[car demo here. run it for awhile and then talk over it. Use camera 6, WASD
http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_dynamic.html]



That said, fixed hierarchies and the parent-child relationship they establish are extremely useful. Hierarchies allow most objects to be controlled in a natural way. I believe I've said this three times, so I'll invoke Lewis Carroll's line from *The Hunting of the Snark*, "what I tell you three times is true!"

[On screen, before saying each one
**** Work step by step***
**** Once you apply a transform, forget about it.***
**** Draw a picture!***
**** Undo if you're not moving towards your goal.***
]

Personally, when I'm thinking through what series of transforms I want to apply, I work step by step. Perhaps the most important rule I can offer you is this, "once you've applied a transform, forget about it entirely!" That transform's applied, it's history, and you now have some new object in a new position, possibly with some new location at the origin. Draw a picture if it helps - I usually do.

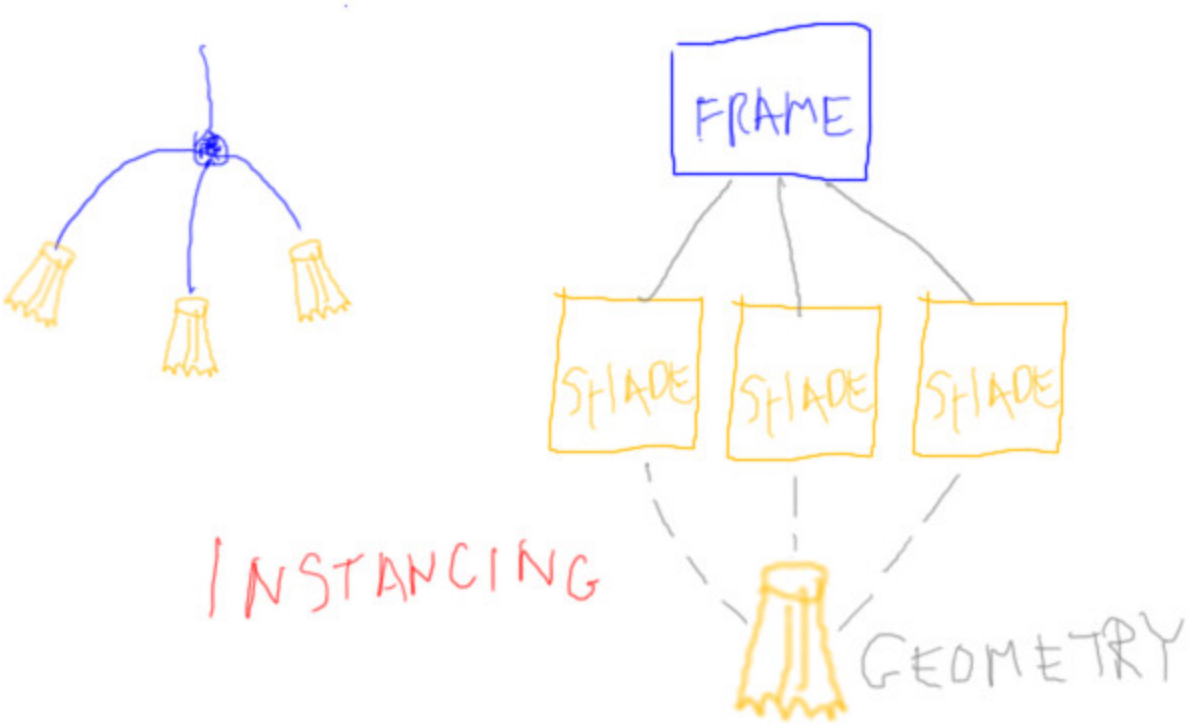
Once a transform is applied, where does your new object want to move from there? If you find you've messed up and the new object is not moving towards your goal, undo and try again.

[Additional Course Materials: The Hunting of the Snark is here

http://www.literature.org/authors/carroll-lewis/the-hunting-of-the-snark/chapter-01.html]

Question: Instancing

[show lamp and three shades hierarchy, show shades sharing a single mesh]



“Instancing” is the idea that a single geometric set of triangles can be reused again and again. For example, for a lamp you might make a single piece of geometry for a bulb fixture. If you want three bulb fixtures, you set up three different transforms for where the objects go, but reuse the same set of triangles for each one. In three.js the set of triangles is called its geometry, though often you’ll hear it called the object’s mesh. A “mesh” in three.js is a type of object.

```
var lamp = new THREE.Object3D();
var cylinderGeometry = new THREE.CylinderGeometry( 20, 20, 100, 32 );
for ( i = 0; i < 10; i++ )
{
        var cylinder = new THREE.Mesh( cylinderGeometry, cylinderMaterial );
        cylinder.rotation.x = 20 * i * Math.PI/180;
        lamp.add( cylinder );
```

```
    }

var lamp = new THREE.Object3D();
var cylinderGeometry = new THREE.CylinderGeometry( 20, 20, 100, 32 );
for ( i = 0; i < 10; i++ )
{
    var cylinder = new THREE.Mesh( cylinderGeometry, cylinderMaterial );
    cylinder.rotation.x = 20 * i * Math.PI/180;
    lamp.add( cylinder );
}
```

This is easy enough to do in three.js: if the geometry doesn't change, just reuse it. Here's a simple example: I create the geometry for a cylinder and then reuse it again and again for each mesh object that I make. Notice that an object is make from some geometry and a material. This means I could have given each cylinder a different material while reusing the same geometry.

[--- new page ---]

Say you have a car model with four tires. The question to you is,

Which of these are valid reasons for using instancing for tires?

[ ] It allows an individual tire to be shown as out of air and flat.
[ ] It reduces the number of transforms needed.
[ ] It can use less memory.
[ ] It allows some tires to be drawn with less geometry than others.

[Additional Course Materials: See this demo http://mrdoob.github.com/three.js/examples/webgl_performance_doublesided.html for 5000 instanced spheres on the screen. More in-depth coverage of instancing can be found in this technical article: http://web.archive.org/web/20120426210216/http://developer.nvidia.com/node/20]

Answer

The first reason is false. Since a single set of triangles is shared by all four tires, each tire must look exactly the same, at least as far as geometry is concerned. A flat tire doesn't look the same as the others. The last reason can also be rejected, as each tire is drawn with the same geometry.

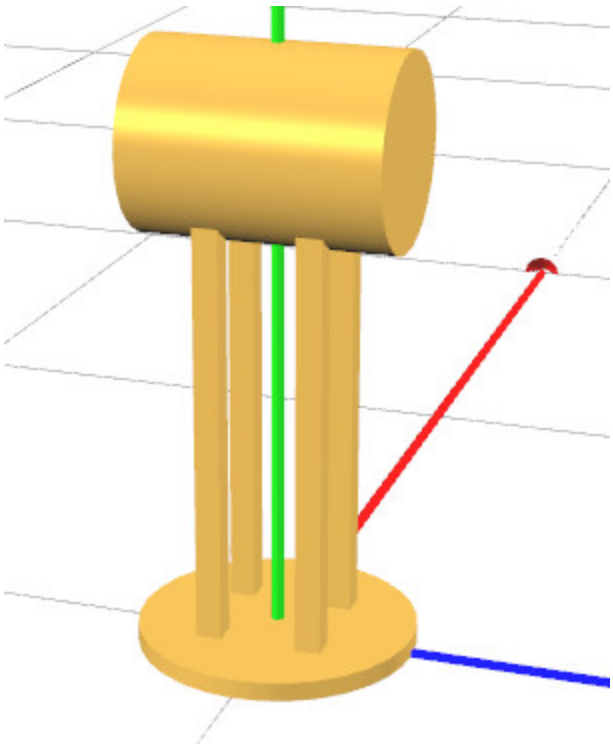
[yes, reviewers, this is not entirely true, geometry or vertex shader blah blah, but really it's not a correct answer.]

The second reason is not true, as each tire needs to have its own transform.

The third reason is definitely true. Instead of storing four separate triangle sets for each tire, we store just one. The savings can be huge when there are many identical parts. This reuse can also increase performance on the GPU, depending on how the data is passed in.

Lesson: Robot Arm

[Show forearm test image.]



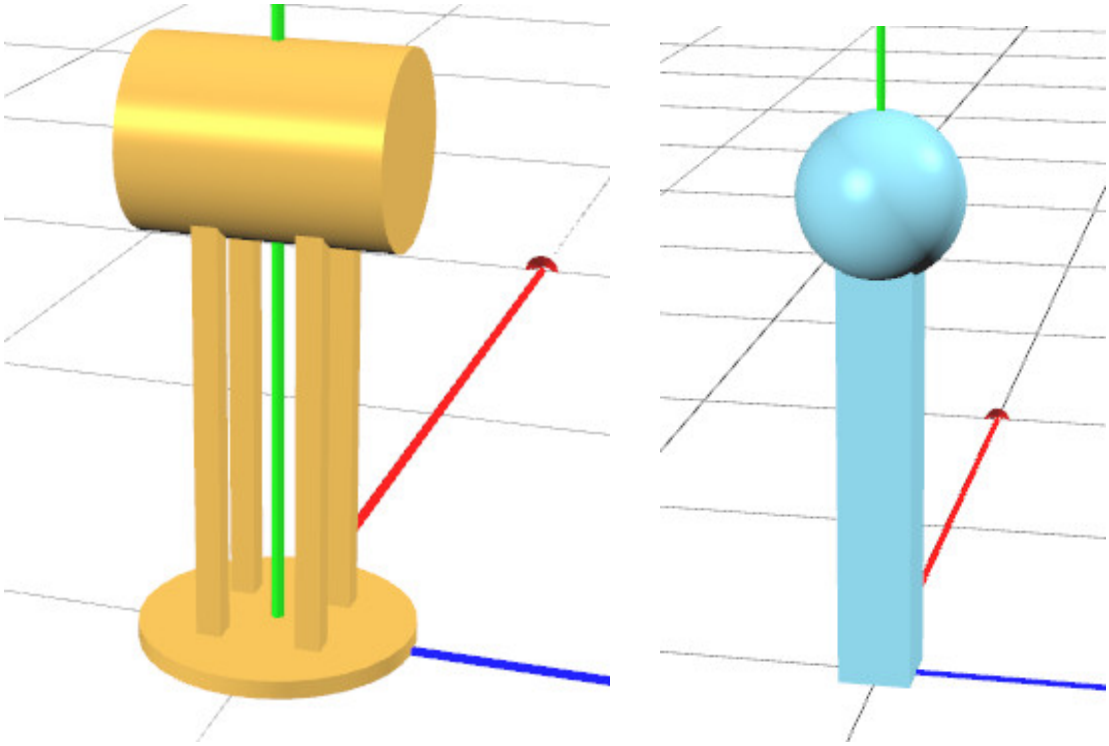
Let's see how a simple hierarchy works in practice. I'll make a robot with two parts to its arm: a forearm and an upper arm.

The forearm is this piece, made of six objects. I run some tests on it and it looks good. I've made it so that the point where I'll attach it to the upper arm is at the origin. I do this because I know that when a rotation is applied, the rotation happens with respect to the point at the origin.

```
forearm = new THREE.Object3D();
var faLength = 80;
createRobotExtender( forearm, faLength, robotForearmMaterial );
scene.add( forearm );
```

The code to create my forearm is this. There's not a lot to it: I create an Object3D, I call createRobotExtender, and then I add the forearm to the scene so I can see it. My createRobotExtender function add a bunch of geometric objects to the forearm object. "faLength" is passed in to let this function know how tall to make the extender.

It doesn't really matter what I add to the forearm object. The only rule is that, whatever I put into this object, I need to remember that this object itself will rotate around its origin.



Make some room to the right, so we can add a drawing, as shown]

```
forearm = new THREE.Object3D();
var faLength = 80;
createRobotExtender( forearm, faLength, robotForearmMaterial );
scene.add( forearm );
```

```
upperArm = new THREE.Object3D();
var uaLength = 120;
createRobotCrane( upperArm, uaLength, robotUpperArmMaterial );
scene.add( upperArm );
```

I design my upper arm in the same way and code it up about the same. Again, it rotates around the origin. The bit we're interested in is how these two objects get hooked together, so that the forearm is a child of the upper arm.

```
forearm = new THREE.Object3D();
var faLength = 80;
createRobotExtender( forearm, faLength, robotForearmMaterial );

arm = new THREE.Object3D();
var uaLength = 120;
createRobotCrane( arm, uaLength, robotUpperArmMaterial );

// Move the forearm itself to the end of the upper arm.
forearm.position.y = uaLength;
arm.add( forearm );

scene.add( arm );
```

Here's the code that attaches the two objects together. We make the two arm pieces the same as before. The key difference is that, instead of adding the forearm to the scene itself, we add the forearm to the upper arm. We also move the forearm to be at the top of the upper arm.

At this point the upper arm can be moved around, rotated, scaled, and all of its parts will be transformed in the same way. This is why I renamed this object to be the "arm" instead of upper arm, since in fact it holds both the upper arm and the forearm.

[add below code above]

```
arm.rotation.y = 30 * Math.PI/180;
```

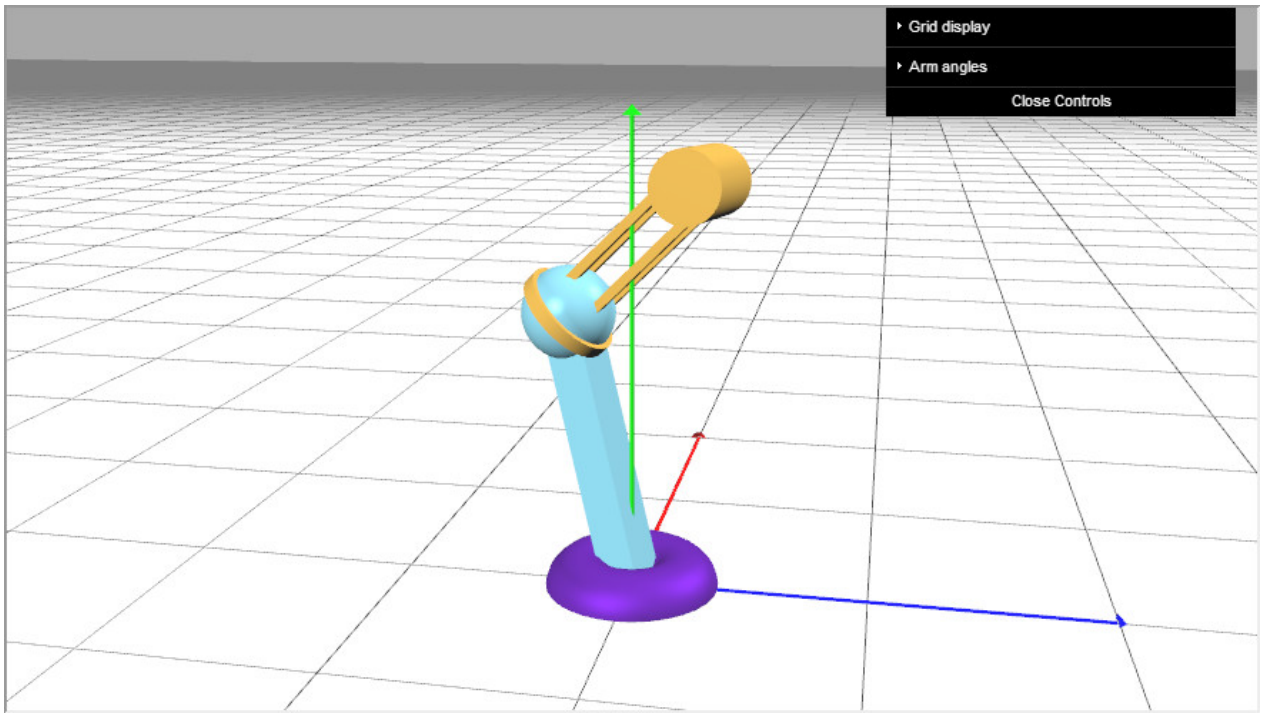
For example, if we set a rotation on the arm, the whole object will rotate. What's important to realize here is that the arm doesn't particularly know or care about what's inside any of its children. When I add the forearm to the arm, the forearm becomes just another part of the arm, no different than adding a simple sphere or block.

```
forearm.rotation.z = -20 * Math.PI/180;
```

[put to right of this code ***Ta Ra*** *Tf Rf* O]

However, since we can manipulate any object by changing its transforms, we can also change the position of the forearm itself. So changing the angle with this line of code will change the angle of the forearm, without affecting anything else. The transform order is the forearm's rotation and translation followed by the arm's rotation and translation. This means the forearm can be transformed however you wish, without knowing about the arm as a whole. The arm's transforms are applied after.

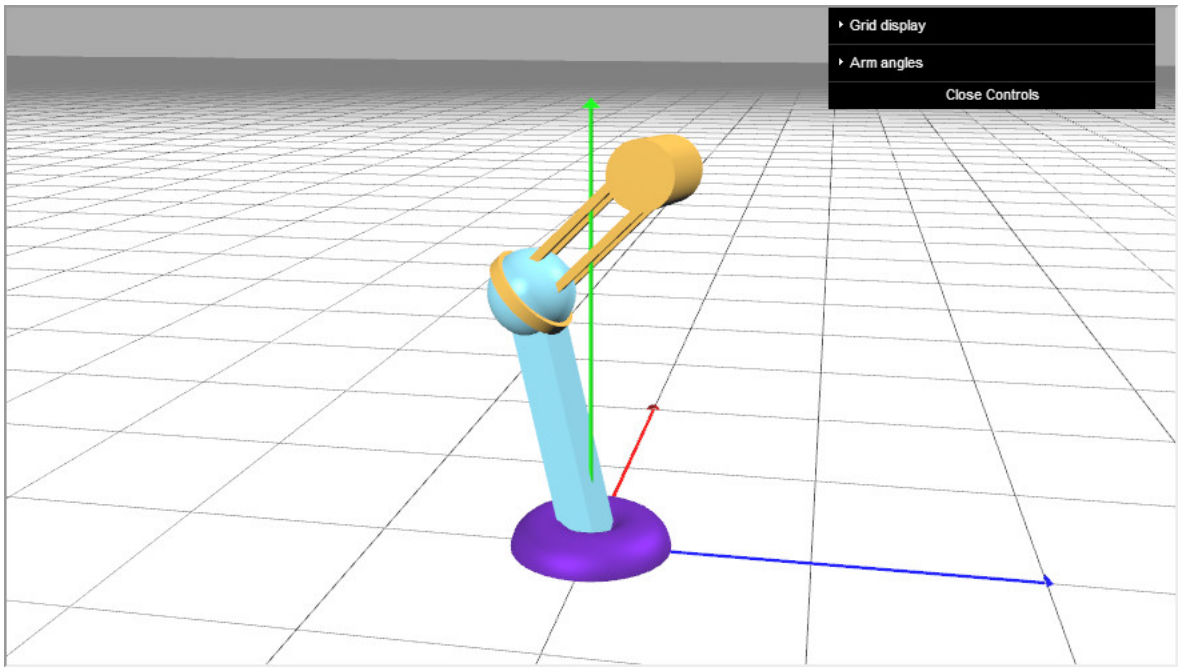
[start demo running:
http://www.realtimerendering.com/udacity/?load=unit4/unit4-robot_arm_extended_exercise.js]



Give the demo a try, to see how the two parts interact with their separate controls.

Demo: Robot Arm

[http://www.realtimerendering.com/udacity/?load=unit4/unit4-robot_arm_extended_exercise.js]



[NOTE: no conclusion needed, as the problem set naturally flows from the previous lesson, and Unit 5 is closely tied to Unit 4. If I desperately need one, talk over http://mrdoob.github.com/three.js/examples/webgl_lines_colors.html]

Problem Set

Problem 4.1: Extended Robot Arm

[show the final thing here, then maybe the robot body part in action, by itself, similar to how we see the forearm in action by itself: http://www.realtimerendering.com/udacity/?load=unit4/unit4-robot_arm_extended_exercise.js]

In this exercise you'll add an upper-upper arm to the robot, let's call it the body. The body is created by the ***createRobotBody*** function you'll see in the code. This body can tilt and it has the upper arm attached to its top. Your job is to add this body to the robot's design.

You might want to turn off display of the other two arm parts while you're testing out the body itself. Simply comment out the line of code that adds the arm to the scene.

When you're done, your robot arm should look like this.

If you really want to get fancy, you can add a Y axis control for the body itself. It's pretty easy to do this, just three lines of code - take a look in the user interface "setupGUI" method and in the render() method.

Answer

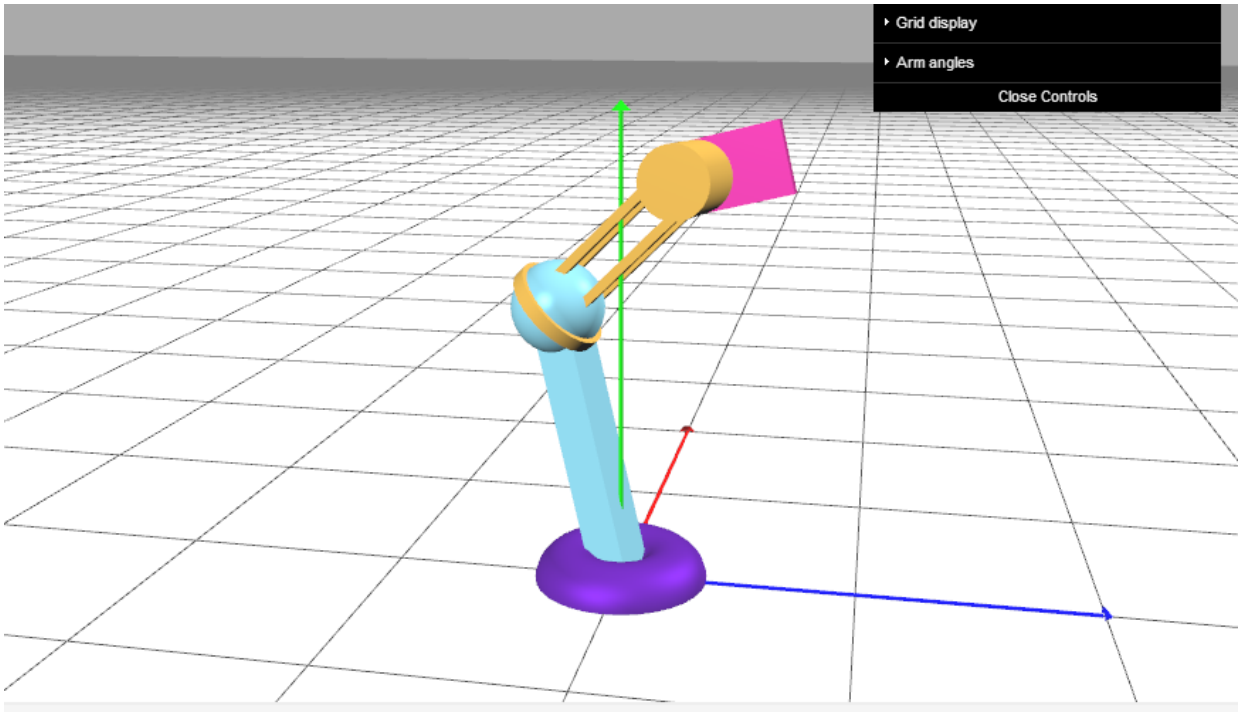
This exercise extends the hierarchy by adding the body object so that the arm is attached to its top. Instead of adding the arm to the scene itself, create the body and attach the arm to the top of it.

```
body = new THREE.Object3D();
var bodyLength = 60;

createRobotBody( body, bodyLength, robotBodyMaterial );

// Move the upper arm itself to the end of the body.
arm.position.y = bodyLength;
body.add( arm );

scene.add( body );
```

```
body = new THREE.Object3D();
var bodyLength = 60;

createRobotBody( body, bodyLength, robotBodyMaterial );

// Move the upper arm itself to the end of the body.
arm.position.y = bodyLength;
body.add( arm );

scene.add( body );
```

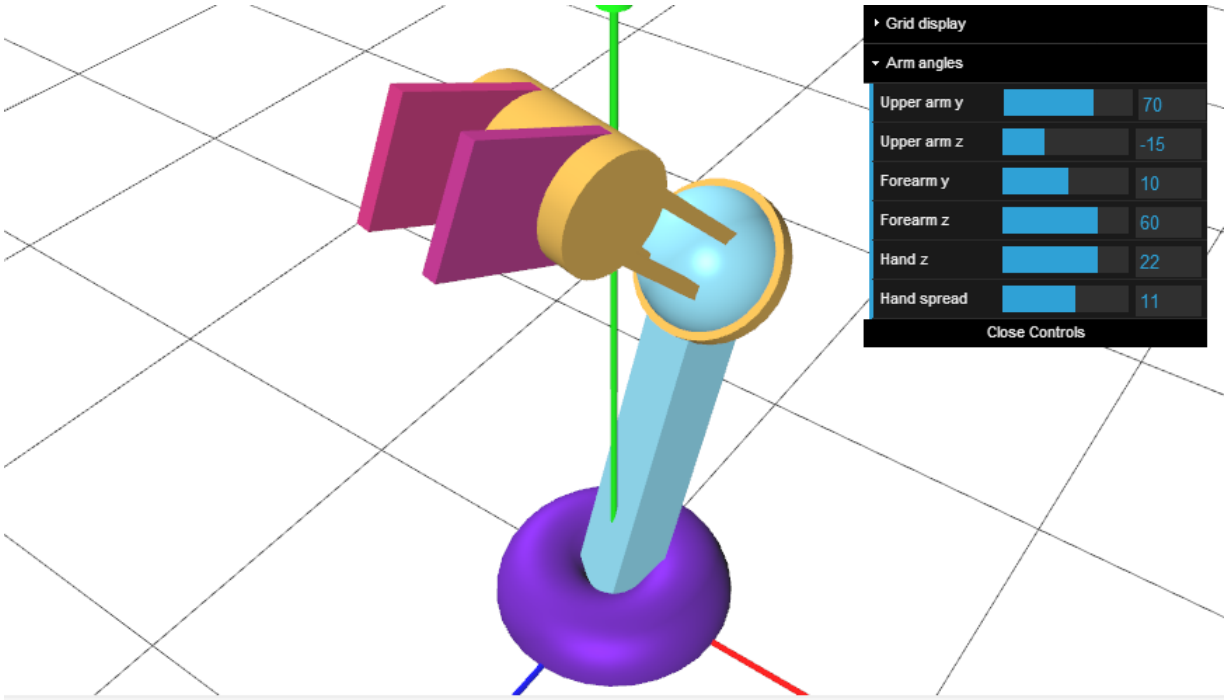
Problem 4.2: Robot Hand

[Show initial exercise starting conditions
http://www.realtimerendering.com/udacity/?load=unit4/unit4-robot_hand_exercise.js
]



In this exercise you'll finish adding a hand to the original robot. You'll see in the initial code I've created one grabber for the hand by using the createRobotGrabber function. It starts like this, with just this one piece of the hand.

[Show solution *redacted*]



Your goal is to create a hand that has two grabbers that slide along the cylinder and rotate together. I have provided sliders for both of these controls.

When you're done, the hand should look like this. The "hand z" control rotates it as a whole, the "hand spread" slider opens and closes the hand.

Note that for this exercise you'll have to modify both the fillScene() method and the render() method.

Answer

The code to make the right grabber is essentially the same as for the left grabber.

```
handRight = new THREE.Object3D();
createRobotGrabber( handRight, handLength, robotHandRightMaterial );
// Move the hand part to the end of the forearm.
handRight.position.y = faLength;
forearm.add( handRight );
```

```
handRight = new THREE.Object3D();
createRobotGrabber( handRight, handLength, robotHandRightMaterial );
// Move the hand part to the end of the forearm.
handRight.position.y = faLength;
forearm.add( handRight );
```

The slightly tricky bit is that you need to add to the render() method. You have to move the grabber in the *negative* Z direction when the slider is slid:

```
handRight.rotation.z = effectController.hz * Math.PI/180;   // yaw
// negate to go the other direction
handRight.position.z = -effectController.htz;// translate
```

```
handRight.rotation.z = effectController.hz * Math.PI/180;    // yaw
// negate to go the other direction
handRight.position.z = -effectController.htz;    // translate
```

Problem 4.3: Series of Transforms

[***Ta Tb Tc Td O***
Tb Td Ta Tc O]

Normally matrix order matters. If I rotate and then translate an object, I'll usually get a different result than if I translate and then rotate. However, there are exceptions.

Say I give you a bunch of the same type of transforms, such as four different translation matrices. If I mixed up the order of these transforms, like on the second line here, would any object still be transformed in exactly the same way by this new order, or does the ordering matter?

Yes No
() () Can a series of translations be done in any order?
() () Series of rotations?
() () Of scales?
() () Of rotations and translations all along the same axis?

Ignore precision errors; assume you have infinite precision. Also assume the scales and mirrors are nothing tricky, just the typical transforms generated by three.js.

Answer

Translations represent movements. Just like vectors can be summed together to make a single movement vector, translations can be multiplied together to get a single translation. The order that we add the vectors doesn't matter. Likewise, translation order among translations doesn't matter. So the answer is "yes".

Rotation order certainly does matter, reversing the order will normally change the effect. So this second question's answer is "no".

Each axis scale is independent of the others. If you stretch an object along the Y axis, its X and Z coordinates do not change. Because each is independent, there is no danger of the scales interacting with each other. Each axes' scale factors can simply be multiplied together. Multiplication is commutative, so the scales can be done in any order. "Yes."

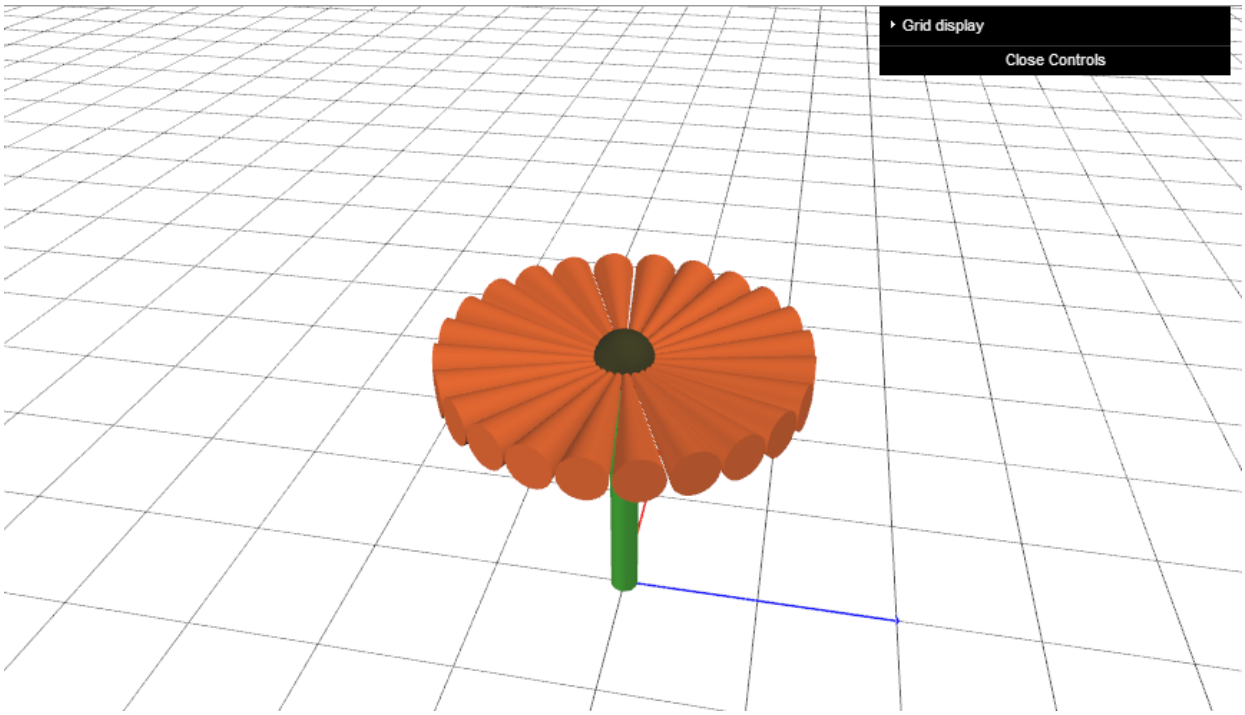
For this last question, say you rotate along the Y axis, and translate along this same axis. The rotations can be done in any order, since each is just an angle change around the Y axis. The rotation affects the X and Z coordinates of any points transformed, translation affects only the Y coordinates. Since these are independent values, rotations and translations along an axis can be evaluated in any order. So the answer is "yes".

Problem 4.5: Make a Flower

First, and this is important! Save your code around for this exercise. You'll use it as a basis for the next exercise. We don't start the next exercise with the solution for this one. At this point you should be using a text editor for these exercises (I like Notepad++, others like Sublime Text). By saving code snippets around you can look these over in future exercises and see what you did.

In this exercise you will make a flower. You start with this, a stem and stamen of the flower, with a single petal part - a cone - sort of stuck in the stem. I start you off with this cone, which is a child of a petal, which is the child of the flower as a whole.

[exercise at http://www.realtimerendering.com/udacity/?load=unit4/unit4-flower_exercise.js]



Your job is to make 24 instances of this petal and form a flower. You'll need to use rotation Euler angles, applied in the proper order, and translations, and you'll need to apply these to the cone object and the petal object which is its parent. Notice that all the tips of the cones meet in the middle of the sphere, the stamen.

[Additional Course Materials: Save your code from this exercise! Any text editor is fine, I like Notepad++ http://notepad-plus-plus.org/, others prefer Sublime Text http://www.sublimetext.com/. For searching through text files, I like Agent Ransack http://www.mythicsoft.com/page.aspx?type=agentransack&page=home]

Answer

Here's my solution:

```
for ( var i = 0; i < 24 ; i++ )
{
        var cylinder = new THREE.Mesh( cylGeom, petalMaterial );
        cylinder.position.y = petalLength / 2;

        var petal = new THREE.Object3D();
        petal.add( cylinder );
        petal.rotation.z = 90 * Math.PI/180;
```

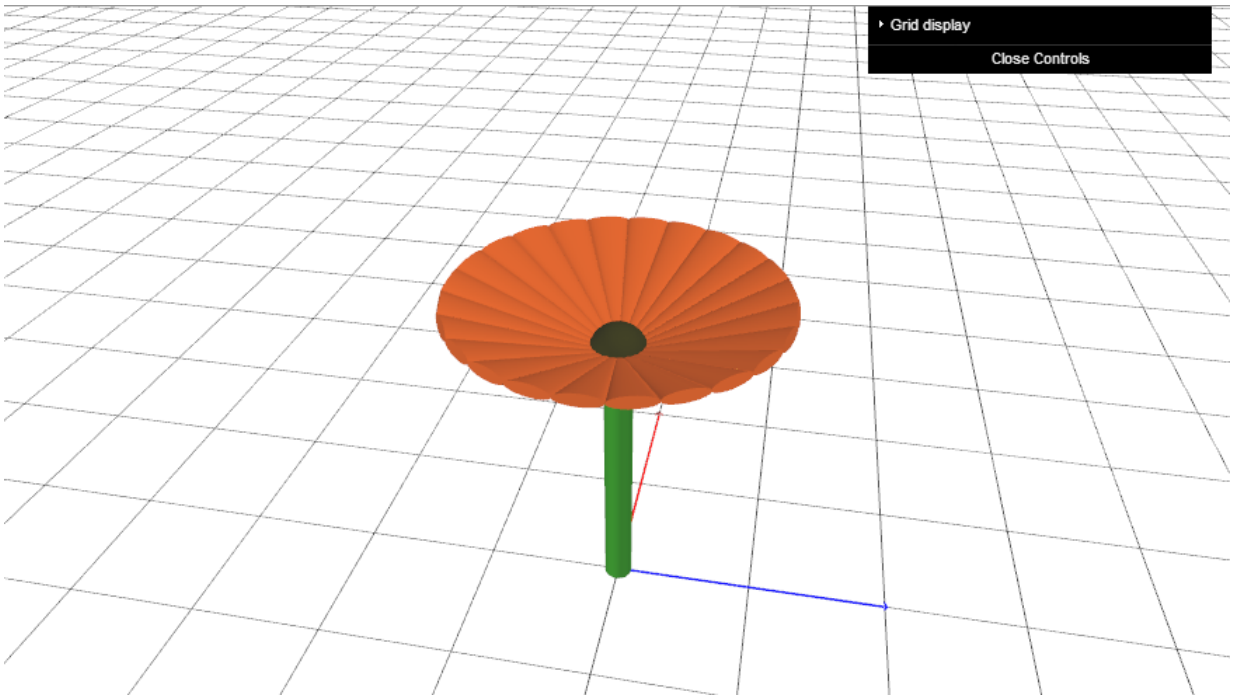
```
            petal.rotation.y = 15*i * Math.PI/180;
            petal.position.y = flowerHeight;

            flower.add( petal );
    }
```

```
for ( var i = 0; i < 24 ; i++ )
{
    var cylinder = new THREE.Mesh( cylGeom, petalMaterial );
    cylinder.position.y = petalLength / 2;

    var petal = new THREE.Object3D();
    petal.add( cylinder );
    petal.rotation.z = 90 * Math.PI/180;
    petal.rotation.y = 15*i * Math.PI/180;
    petal.position.y = flowerHeight;

    flower.add( petal );
}
```

The loop creates 24 petals. For the cylinder, I position it so that its tip is at the origin. I then know any rotations I do to the petal will keep this tip there. The trickiest bit is that I first want to rotate the petal so it is horizontal, then rotate it increments of 15 degrees around the origin. That's why I use the Z axis for rotation, then the Y axis. Remember that Euler angles are applied in the order Z, Y, X.

After the rotation is done, I then move the petal up to its final position at the flower's height.

Problem 4.6: Improved Petals



Starting with your solution to the previous problem, make two small changes that will make the petals look more realistic. The first is to squish the petals: make them one quarter as tall as they were before. The second is to tilt the petals up by 20 degrees. When you're done you should get something like this, which looks at least a little bit more like a flower.

Notice that all the points of the petals should still be in the center of the stamen, this sphere at the top of the stem. When I first solved this problem my petals were pointing below the stamen, which isn't what I wanted. I expect you can do better than I first did.

[exercise at
http://www.realtimerendering.com/udacity/?load=unit4/unit4-flowersquish_exercise.js]

Answer

The code changes are pretty small, though they took me a bit to get correct.

```
for ( var i = 0; i < 24 ; i++ )
{
        var cylinder = new THREE.Mesh( cylGeom, petalMaterial );
        cylinder.scale.x = 0.25;
        cylinder.position.y = petalLength / 2;
        //cylinder.rotation.z = -20 * Math.PI/180;
```

```
            var petal = new THREE.Object3D();
            petal.add( cylinder );
            petal.rotation.y = 15*i * Math.PI/180;
            petal.rotation.z = (90-20) * Math.PI/180;
            petal.position.y = flowerHeight;

            flower.add( petal );
        }
```

```
for ( var i = 0; i < 24 ; i++ )
{
    var cylinder = new THREE.Mesh( cylGeom, petalMaterial );
    cylinder.scale.x = 0.25;
    cylinder.position.y = petalLength / 2;
    //cylinder.rotation.z = -20 * Math.PI/180;

    var petal = new THREE.Object3D();
    petal.add( cylinder );
    petal.rotation.y = 15*i * Math.PI/180;
    petal.rotation.z = (90-20) * Math.PI/180;
    petal.position.y = flowerHeight;

    flower.add( petal );
}
```

The scale was easy enough, I modified the cylinder's X scale to be one-quarter of what it was before.

The Z rotation I initially applied to the cylinder, but that was not correct. That rotated the cylinder around its own center. This gave an interesting result - try it out sometime! - but wasn't what I was aiming for. The proper rotation was to simply not rotate the petal itself as far along the Z axis, rotating 70 degrees instead of 90.

Problem 4.7: Scale vs. Translate

Just one quick question. Say I apply a uniform scale to an object centered at the origin and then translate it. That's case A. I start again and apply the translation first to the object, then scale it - case B. Assume that the scale and translation transforms actually *do* something - I'm not scaling by 1, or moving by 0 feet, or anything silly like that.

[***A: scale, then translate B: translate, then scale***]

Mark any of the following that are always true.

True
[ ] Neither object's center has moved from the origin.
[ ] The object is the same size in both cases.
[ ] The object is in the same position in both cases.
[ ] The object's center has moved farther in case B than case A.

Answer

This first answer is simply false. Both objects are translated by some amount from the origin.

This second answer is true. If you scale by the same scaling matrix, the objects are the same size. They may be in different positions, but they'll have the same dimensions.

The object is not in the same position. If you scale up by a million, then move a foot, you've moved a foot for case A. If you move an object a foot, then scale it up by a million, its origin will essentially move a million feet for case B.

This next one could be true or false, it depends on the scale factor. If the object is growing larger, then it's true - B would move farther than A. If the object is shrinking, A would move farther than B.