Lesson 10: Interaction and Animation

Lesson: Introduction

[recorded 3/30, parts 1 (no name, really) & 2 screenflick]

[http://maxogden.github.com/voxel-engine/ - select brick “3”, then start taping: start by walking. Then start chopping (left click), then build by control + left click. Then move and bounce around, space bar.
]

It’s easy in three.js to hook up the mouse, keyboard, and other devices as interactive controls for

objects in a scene. We've been doing this all along with demos such as the robot arm. There's much javascript code to draw upon for getting inputs connected to programs, such as the dat GUI library for sliders.

[chop and build for a bit]

However, up to this point we've had little *direct* interaction with our virtual worlds. By "direct interaction" I mean using the mouse to click on the world itself and change its appearance. The main direct interaction so far has been using the mouse to move the camera. As an example of another direct interaction, in this demo by Max Ogden and James Halliday I'm clicking to remove blocks or add them. Programs can seem limiting without these sorts of direct interaction with what the user sees.

[move and bounce around]

Even when moving around the scene with the mouse, in many demos it's as if we're a ghost: we can walk through walls or drop through the floor, there's nothing stopping us. By instead performing what is called "collision detection" between objects, and having the program respond to those collisions, we can create a more believable world. We can also use physics to determine how our actions and the movements of other objects change what we see.

If you've played any computer game, you know how various button pushes and so on are tightly tied to what you see on the screen. How the world reacts to inputs such as these is often shown through animation techniques of one sort or another. You push the thumb stick forward and you see your character run. You bump a car in a race game and the other driver turns his steering wheel to avoid you, or bumps you back.

[http://djazz.mine.nu/lab/minecraft_items/]

Up to this point we've mostly been ignoring this fourth dimension: time. Once we start changing elements in our scene over time, we're performing animation. Animations can also be run in their own right: streams flow down a hillside, people walk past you in a marketplace, the sun sets over ocean waves. Animation, such as shown in this demo by Daniel Hede, gives a scene life and a sense of engagement, even when there is little interaction.

What I'm going to cover in this unit is a sampling of interaction and animation techniques. So, let's get going!

[Additional Course Materials:

The demos shown are by [Max Ogden and James Halliday](http://maxogden.github.com/voxel-engine/), (also see [the related blog](http://maxogden.com/)), and [Daniel Hede](http://djazz.mine.nu/lab/minecraft_items).

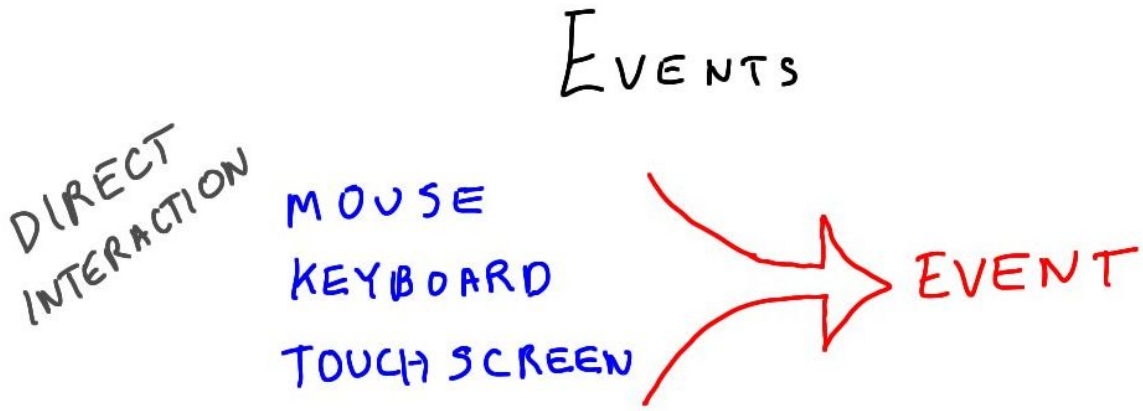
In case you haven't looked at the documentation, [the dat.GUI library](https://code.google.com/p/dat-gui/) is how we've been adding sliders to programs.

[This tutorial](http://stemkoski.github.com/Three.js/Chase-Camera.html) by Lee Stemkoski shows how to set a camera to follow behind a moving object; use WASDQE to move. Switching between cameras is shown in [this demo](http://stemkoski.github.com/Three.js/Multiple-Cameras.html), hit the 1 and 2 key.

]

Lesson: Events

[draw ***mouse, keyboard,*** leave room for ***touch screen*** -> ***event***]



Direct interaction, where for example you use the mouse to control or influence what is happening in the scene, is handled by using events. An event is generated by the system whenever the user presses a key or uses the mouse. There are other kinds of events, but these two - keyboard and mouse input - are the main ones we care about in three.js.

[add touch screen]

Well, I should mention there's also a touch interface that's somewhat similar to the mouse interface, so that you can interact with scenes on a smartphone or other touch-sensitive display.

You can ignore any and all events or can listen for them. It's easy to listen to an event.

```
document.addEventListener( 'mousedown', onDocumentMouseDown, false );
```

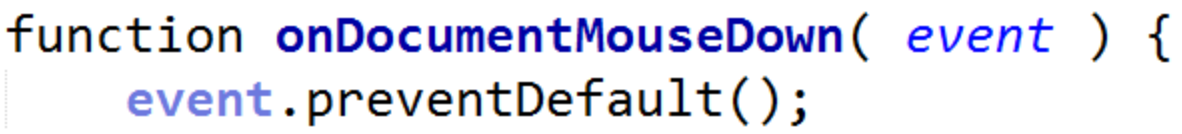
```
document.addEventListener( 'mousedown', onDocumentMouseDown, false );
```

Here's code for listening to a mouse-click event, specifically "mouse down", which means when the mouse is first clicked. If you wanted to wait until the mouse button was released, you would listen for "mouse up".

The addEventListener takes a function name as an input. Whenever the mouse button is clicked,

this function is called.

```
function onDocumentMouseDown( event ) {
    event.preventDefault();
```



This code defines the function called on mouse down. This “preventDefault()” line is optional, it makes your program not pass the mouse event on up to the page running your javascript program. I usually don’t use this call. The rest of this function is up to you: a mouse or key click could modify an object, start an animation, you name it.

Enough talk; what follows is an example of using the mouse down event to perform picking. You are orbiting around a bunch of blocks. Try clicking on a block as you move past them.

[Additional Course Materials:
See [this page](http://www.w3schools.com/jsref/dom_obj_event.asp) for a good run through of events.

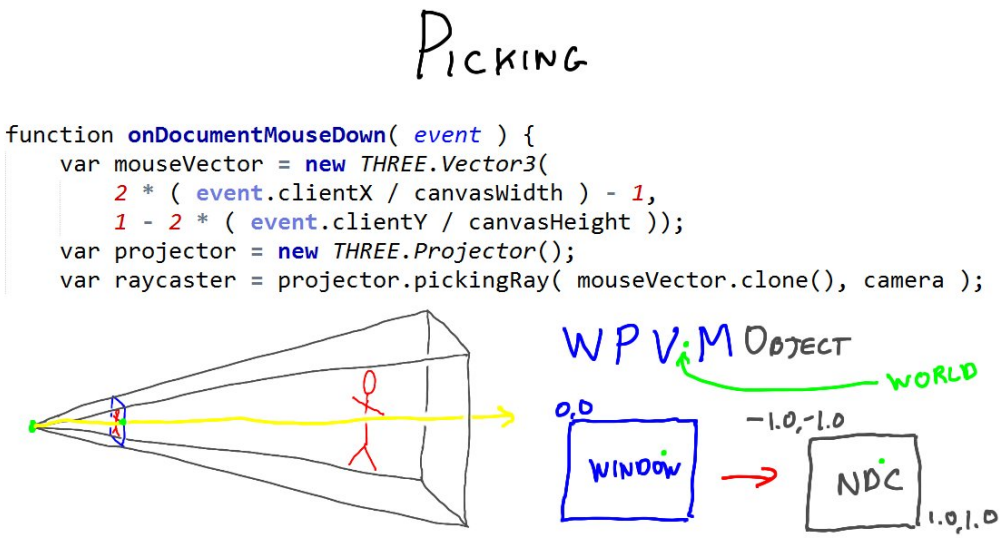
For examples of the various events and how to use them, I recommend looking through the [three.js code examples](https://github.com/mrdoob/three.js/) themselves, [direct download](http://github.com/mrdoob/three.js/zipball/master). Search the HTML code examples for “addEventListener”. For mass file searching on Windows, I like the free [Agent Ransack](http://www.mythicsoft.com/page.aspx?type=agentransack&page=home) utility.
]

Demo: Picking Demo

[unit10-picking_demo.js]

[Additional Course Material:
This demo is based on [a sample from three.js](http://mrdoob.github.com/three.js/examples/canvas_interactive_cubes.html). Note that the sample works without using WebGL at all! Three.js has limited support for non-WebGL 3D rendering.
]

Lesson: Picking




```
function onDocumentMouseDown( event ) {
        event.preventDefault();

        var mouseVector = new THREE.Vector3(
                2 * ( event.clientX / canvasWidth ) - 1,
                1 - 2 * ( event.clientY / canvasHeight ));
        var projector = new THREE.Projector();
        var raycaster = projector.pickingRay( mouseVector.clone(), camera );
```

```
function onDocumentMouseDown( event ) {
    var mouseVector = new THREE.Vector3(
        2 * ( event.clientX / canvasWidth ) - 1,
        1 - 2 * ( event.clientY / canvasHeight ));
    var projector = new THREE.Projector();
    var raycaster = projector.pickingRay( mouseVector.clone(), camera );
```

The demo performed two separate actions: selecting the object clicked on, and if any object was found, then the object itself was modified and the point clicked on was highlighted.

Picking objects on the screen with the mouse is quite simple in three.js. Here's the first half of

the mouse down function, which sets up to perform picking.

The bulk of the code fires a ray from the eye into the world. This is a common way to perform picking. WebGL itself is focused on rendering, so has no real support for picking and other forms of interaction. It's up to the program to figure this out.

[draw frustum and show the two points.]

To form a ray from the eye we need two points. One's the eye's position in world space, that's easy. The other is formed using a point on the screen, that point being located wherever the user clicked the mouse. What we want to know is this point's location in world space. To do so we need to travel upstream, back along the chain of transformations: window coordinates to normalized device coordinates to view to world coordinates.. Happily, three.js makes this easy (how many times have I said that sentence during this course?).

[draw screen coordinates, upper left is XY origin. Correlate code with actions.]

The 2D point from the mouse down event is a document object model screen coordinate. This is like a WebGL window coordinate, only Y = 0 is at the top edge of the window instead of the bottom.

The mouseVector code here converts from this screen coordinate system into Normalized Device Coordinates, which go from -1 to 1 in X and Y. The Z value doesn't really matter - any value from Z equals -1 to 1 will form a point on the ray.

[Then draw to NDC, then all the way back to world.]

We next create a projector object. Really, it's better to create this object once during initialization and reuse it here, but I wanted to show the code all in one piece. The final line of code does the heavy lifting. It takes the NDC coordinate and forms a raycaster object from it. Ray casting is like ray tracing, except that you only trace rays from the eye into the scene and no further, no new rays are spawned from intersections.

---- new page ---

```
var intersects = raycaster.intersectObjects( objects );
if ( intersects.length > 0 ) {
        intersects[ 0 ].object.material.color.setRGB(
                Math.random(), Math.random(), Math.random() );

        var sphere = new THREE.Mesh( sphereGeom, sphereMaterial );
        sphere.position = intersects[ 0 ].point;
        scene.add( sphere );
```

```
        }

    var intersects = raycaster.intersectObjects( objects );
    if ( intersects.length > 0 ) {
        intersects[ 0 ].object.material.color.setRGB(
            Math.random(), Math.random(), Math.random() );

        var sphere = new THREE.Mesh( sphereGeom, sphereMaterial );
        sphere.position = intersects[ 0 ].point;
        scene.add( sphere );
    }
```

Once we have this raycaster object we can perform picking, also called "***hit testing***". It's one line of code to pick. You feed a list of objects into the raycaster and it returns an array of intersections. The "if" statement checks if the array has anything in it - if it does, then one or more intersections have been detected. These intersections are sorted front to back, so the first intersection on the list is the closest, which is usually the one we want.

[Note data structure for intersects[0]: ***object, face, faceIndex, distance, point***]

This first intersection has information about what the ray hit, including the object, face, and the face's index in the object. We use the object parameter in the next line to change the material to a random color. The record also includes the distance and the point in space. To show where the object was hit, we add a small sphere to the scene using this intersection point for its location. These are just examples - you can do whatever you like with the intersection information you get back.

[Additional Course Materials:
Note: in this lesson I set only the X and Y values when creating the mouseVector Vector3. This is valid in JavaScript, the third value defaults to 0. In retrospect it would have been clearer to simply put the 0.

For a more thorough tutorial on picking in three.js see [this page](http://soledadpenades.com/articles/three-js-tutorials/object-picking/).

Look at the TrackballControllers.js code in three.js if you want to see how touch events are handled.

For a picking demo in WebGL, see [this code](http://voxelent.com/html/beginners-guide/1727_08/ch8_Picking_Scene_Final.html) from the book "[WebGL Beginner's Guide](http://www.amazon.com/WebGL-Beginners-Guide-ebook/dp/B008CEMBPI?tag=realtime

renderin)".

One method I don't discuss in this lesson is the idea of first rendering each object with a unique color. This "ID image" can then be accessed and the color used to find what object is visible at a given pixel.
]

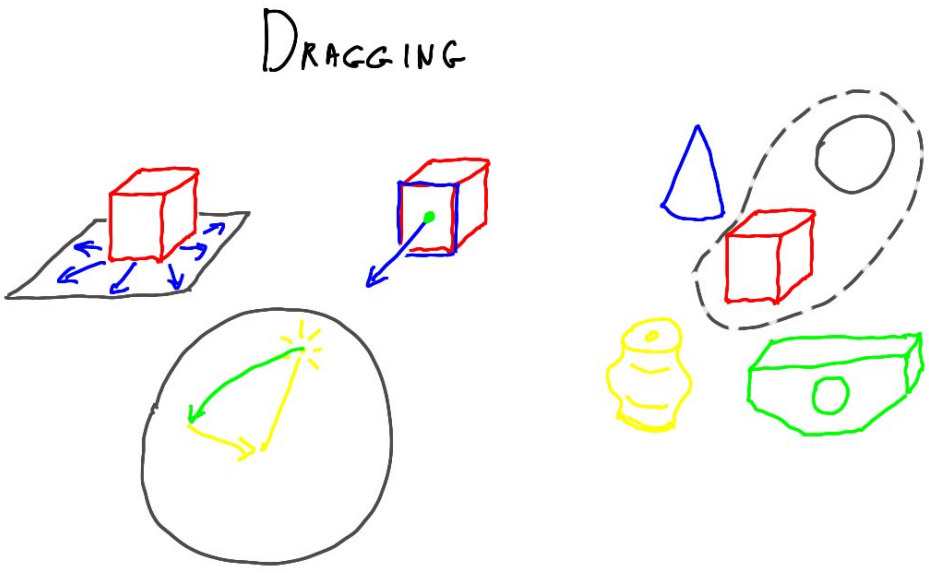
Lesson: Dragging

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

Once you've picked an object, you might want to drag it to a new position. Here's a demo of dragging in action. It's a bit more involved, since you also have to track and respond to mouse move and mouse up events. I'm not going to walk through the code, since you can take a look yourself.

The main question with dragging in 3D space is how to constrain the third dimension. There are usually two choices. One is to constrain the movement to be parallel to the screen's surface, as seen in this demo.

[draw plane and cube, arrows for movement. PUT CUBE IN LAYER and reuse.]



The other is to limit movement of the object to be on a world plane or other geometry in the scene.

[draw another cube, select a face, draw just that arrow.]

For example, if you drag a handle on the face of a box to resize it, you limit interaction to be along the face's normal direction.

[selection types: rectangle]

Beyond the various mouse and keyboard combinations - double-click, shift and control keys - there are also different forms of selection. In areas as diverse as real-time strategy games and computer-aided design programs, you can select an entire set of objects by drawing a rectangle around them on the screen. This sort of selection is done by finding which objects are fully or partially inside what is essentially a smaller frustum.

[selection type: lasso]

Lasso selection lets you draw an arbitrary shape on the screen for choosing a set of objects.

[show light in scene and a dashed sphere, for selection, show curved arrow of light moving.]

Keep in mind that you don't have to make a selectable object a visible object. For example, for positioning a directional light you might put an invisible sphere in a scene and only make it selectable. Wherever the user clicks on the sphere determines a point in space, which in turn determines the direction the light comes from.

This is as far as we're going with selection and interaction. The rest of this unit will be about animation, which is a huge topic.

Lesson: The Rendering Loop

renderer.render(scene, camera);

```
renderer.render(scene, camera);
```

You've initialized your scene with lights, objects, and a camera. Naturally, the next step is to

render. If you've looked through a whole three.js program you've seen this line, telling the renderer class to render the scene. It's possible to call the renderer once during initialization and be done, but usually we want to have the scene be updated as needed.

```
function animate() {
	window.requestAnimationFrame(animate);
	render();
}
```

```
function animate() {
    window.requestAnimationFrame(animate);
    render();
}
```

Once a scene is set up, calling the animate function gets things rolling. The key bit of magic is the window requestAnimationFrame call, which in turn uses the animate method itself. This method will start up a rendering whenever it makes sense to do so. The other thing that requestAnimationFrame does for you is help lower the temperature of your computer. No, really. Say you have a WebGL program running in a browser window. If you switch to another tab, the browser will know that the program is no longer visible and it will not keep it running.

```
function render() {
	var delta = clock.getDelta();
	cameraControls.update(delta);
	renderer.render(scene, camera);
}
```

```
function render() {
    var delta = clock.getDelta();
    cameraControls.update(delta);
    renderer.render(scene, camera);
}
```

The animate method calls the render method to, need I say it?, render the scene. Actually, it can do much more than that. I like to separate out the render method itself, just to keep track of what's doing what. This method is where you want to put code that performs changes *before*

rendering the scene. For example, if the user modifies a slider, the slider's value is used here to modify the scene - we've seen that in previous programs. If you have a bouncing ball animation, you change the position of the ball here. This is where animation gets going.

[Additional Course Materials:
See [this article](http://paulirish.com/2011/requestanimationframe-for-smart-animating/) for a bit more about requestAnimationFrame.
]

Lesson: Incremental Animation

```
var bodyhead = new THREE.Object3D();
bodyhead.add(body);
bodyhead.add(head);

// add field for animated part, for simplicity
bbird.animated = bodyhead;

bbird.add(support);
bbird.add(bodyhead);
```


```
var bodyhead = new THREE.Object3D();
bodyhead.add(body);
bodyhead.add(head);

// add field for animated part, for simplicity
bbird.animated = bodyhead;

bbird.add(support);
bbird.add(bodyhead);
```


Let's get animating, putting something in the render method to at last make our drinking bird move. I've gotten things ready by structuring the geometry a bit. The bird has two parts, the support, which is the legs and feet, and the animated part, which is the body and head. I've put the body and head into a new Object3D, called bodyhead.

Next I do a thing that JavaScript allows: I create a new parameter for the bird, called animated,

which I set to bodyhead. In this case I want to access the animated part of the bird in the render function, so for convenience I give it a separate parameter.

Adding parameters at any time to an object is a feature of JavaScript. That said, it can also be a bit annoying at times. For example, if I try to set a parameter and make a small spelling error, I'm not told that the parameter didn't previously exist. I could also create what I think is a new name, only to find later that I'm actually using an existing parameter. Long and short: be careful.

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

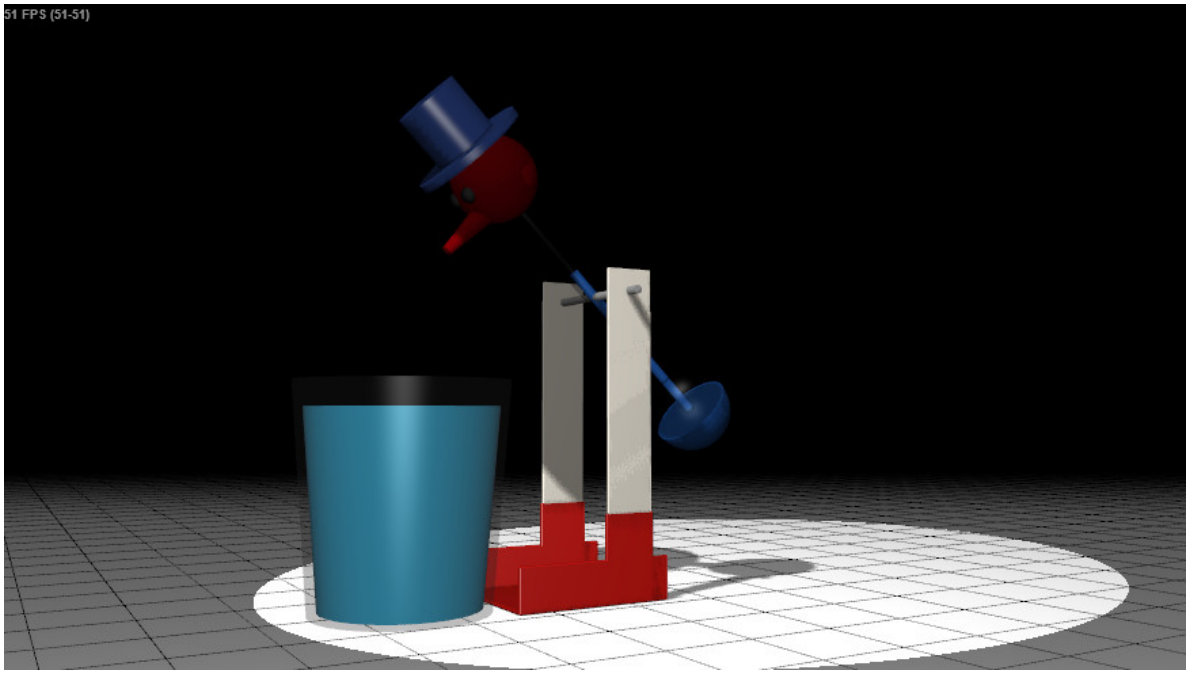
```
var tiltDirection = 1;

function render() {
	bird.animated.rotation.z += tiltDirection * 0.5 * Math.PI/180;
	if ( bird.animated.rotation.z > 103 * Math.PI/180 ) {
		tiltDirection = -1;
		bird.animated.rotation.z = 2*(103 * Math.PI/180) - bird.animated.rotation.z;
	} else if ( bird.animated.rotation.z < -22 * Math.PI/180 ) {
		tiltDirection = 1;
		bird.animated.rotation.z = 2*(-22 * Math.PI/180) - bird.animated.rotation.z;
	}

	renderer.render(scene, camera);
}
```

```
var tiltDirection = 1;

function render() {
    bird.animated.rotation.z += tiltDirection * 0.5 * Math.PI/180;
    if ( bird.animated.rotation.z > 103 * Math.PI/180 ) {
        tiltDirection = -1;
        bird.animated.rotation.z = 2*(103 * Math.PI/180) - bird.animated.rotation.z;
    } else if ( bird.animated.rotation.z < -22 * Math.PI/180 ) {
        tiltDirection = 1;
        bird.animated.rotation.z = 2*(-22 * Math.PI/180) - bird.animated.rotation.z;
    }

    renderer.render(scene, camera);
}
```

Here's where the animation happens. We want to rotate the body around its Z axis, the axis of the crossbar. There is a variable called tiltDirection that is either 1 or negative 1. If it's 1, then the rotation will increase by a small increment, half a degree per render. When the angle reaches a maximum of 103 degrees, the object swings back the other direction by changing tiltDirection to -1. Whatever amount the bird has moved past 103 degrees is used to move it the other way.

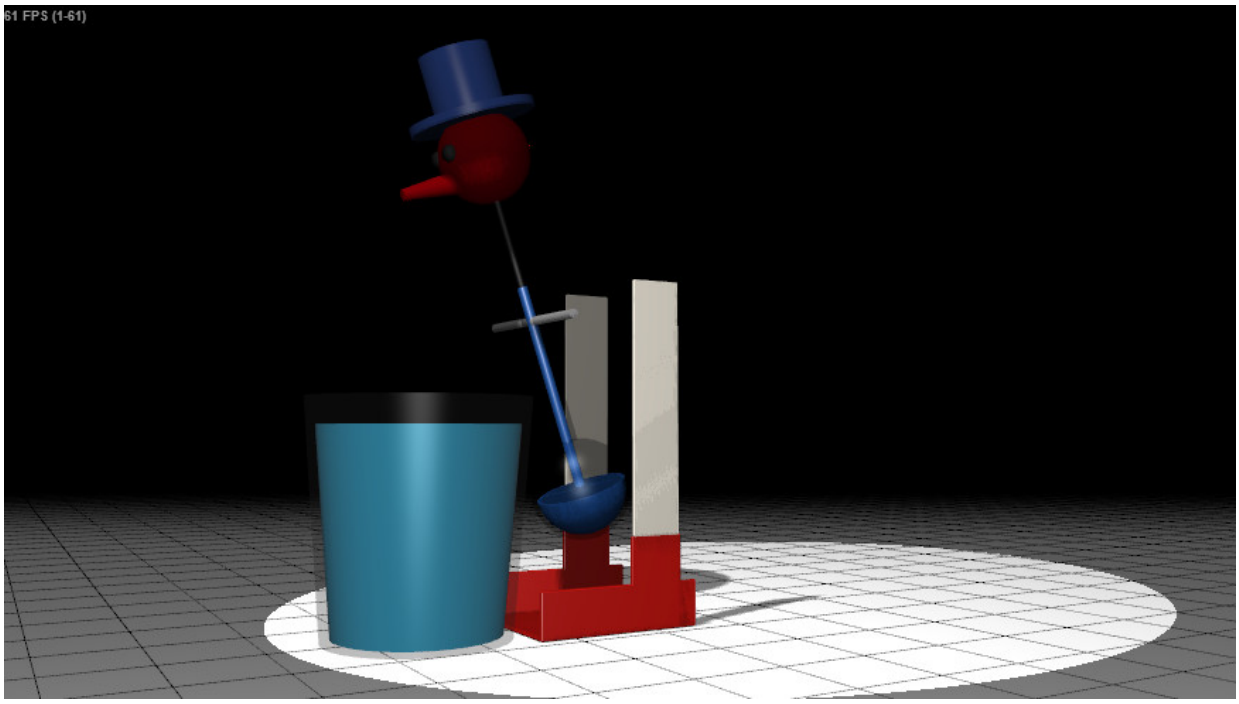
Now the rotation angle is decremented by half a degree, until a limit of -22 degrees is reached, which causes it to reverse again.

[show working bird picture, demo is here unit10-db_animation_solution.js]



What we would like to see is this result, with the drinking bird rotating around its crossbar, forward and back.

[SHOW PICTURE unit10-db_animation_exercise.js]



We get this instead, with the drinking bird rotating around the origin. The problem is that we haven't set the body of the bird to have a pivot point that is where its crossbar is located.

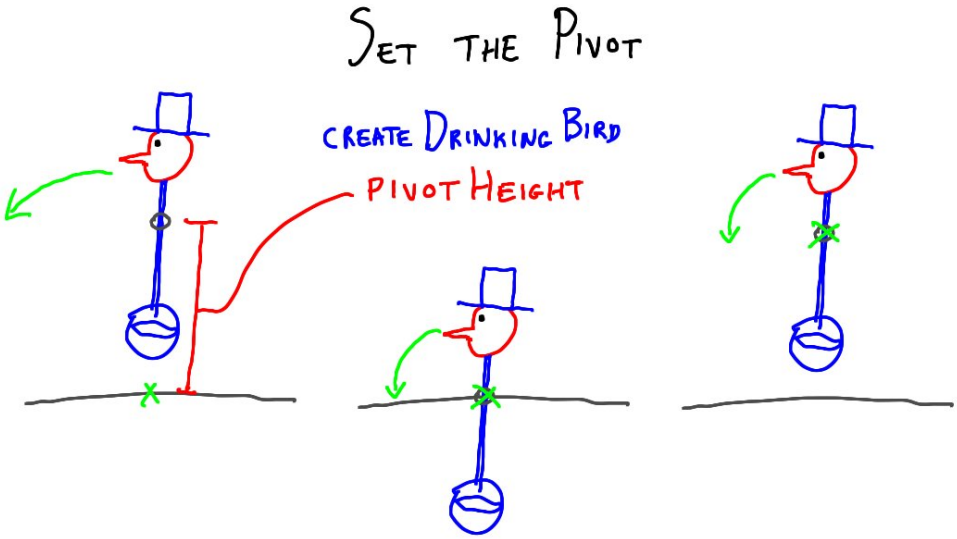
[Additional Course Material:
Adding new parameters to an object can also hurt the performance of accessing properties of that object, [see this presentation](https://developers.google.com/events/io/2012/sessions/gooio2012/224/).
]

Exercise: Set the Pivot

[Draw bird body, show what body is doing currently, translate then rotate. Show a ground plane and origin.]

Your task is to set the pivot point for the body and head to be around the crossbar. Think back to the unit on transforms. We saw how the snowman's arms could go to the wrong place with a bad order of transforms. If we translated, then rotated, the arms would wildly swing in their position, similar to the drinking bird. However, it's a bit trickier than that. The bird's body and head are in the correct spots, so where would we translate them to?

[show body translated down to origin, then rotating, then translating up]



Think about it this way: say we constructed the bird a bit differently. Instead of moving the body and head to their final location, what if we moved them so that the center of the crossbar was at the origin? Then the body and hat would be at the right location for rotating around the crossbar. We could then put both parts into an Object3D, such as the ***bodyhead*** object, and apply the rotation to this parent. We could then move the parent to the proper location after performing the rotation.

Your job is to modify the ***createDrinkingBird*** method so that the pivot is set correctly. I've given you a variable, ***pivotHeight***, which is the height of the crossbar above the origin. I'll also give you a hint: it should take about three lines of code inside just this method to set the pivot.

Answer

[unit10-db_step_anim_solution.js]

```
var pivotHeight = 360;
body.position.y = -pivotHeight;
head.position.y = -pivotHeight;
bodyhead.position.y = pivotHeight;
```

```
var pivotHeight = 360;
body.position.y = -pivotHeight;
head.position.y = -pivotHeight;
bodyhead.position.y = pivotHeight;
```

This problem could be fixed in the body and head code by changing every child, such as the eyes, nose, etc., to be at the right height initially. That's a lot of work! Instead, I use the body and head objects themselves.

[drawing of bird body and head objects, boxes around each, moved down.]

I move the body and head down to the origin by the crossbar's height. This puts the crossbar at the origin. The bodyhead object holds both of these parts, body and head, as children. If I rotate this object around its Z axis, it will properly go around its crossbar.

[Then big box, bodyhead object, moved up.]

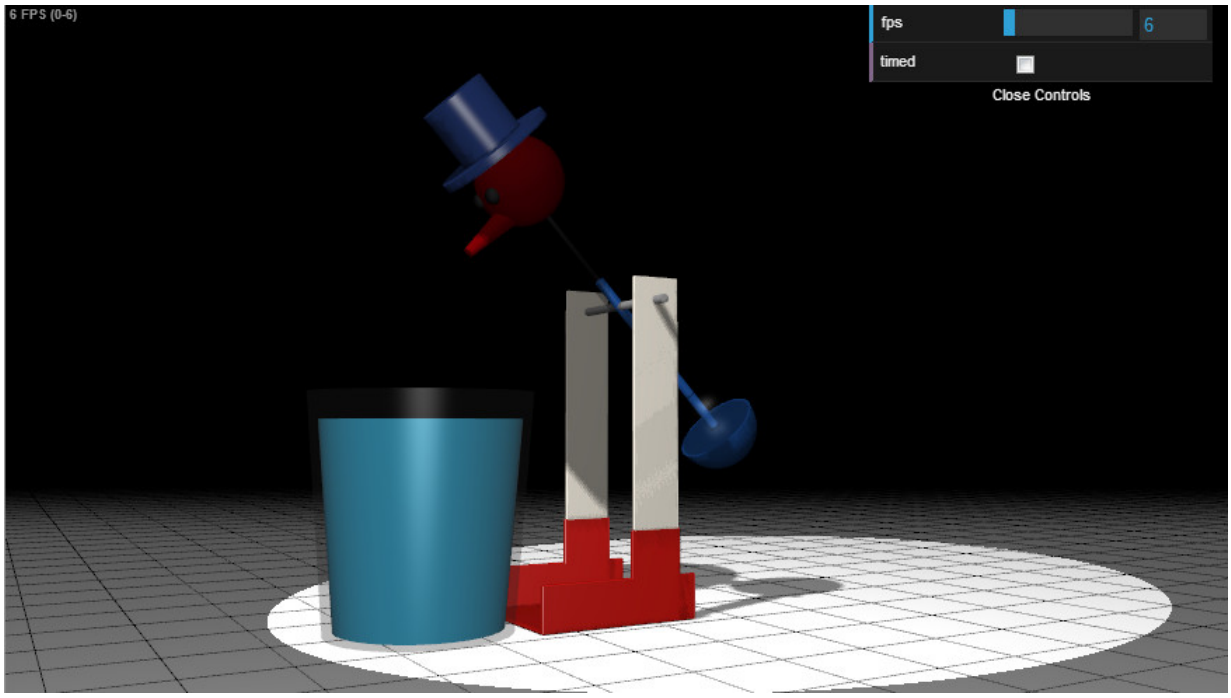
Since translation happens after rotation, I then translate this parent object to its proper location. Now the animation works fine.

[end recording 3/30]

Lesson: Step Size

[recorded 3/31, part 2]
[go to EXT version ext_unit10-db_step_fps_demo.js]

[Show untimed for awhile, sliding FPS bar.]



This animation works, but it has a potential problem. We're basing the animation entirely off of the number of frames rendered. Each frame rotated the drinking bird half a degree. However, what if your machine runs this demo at 60 frames per second, and mine runs it at 30 frames per second? This means your drinking bird is rotating 30 degrees a second, mine is rotating only 15. This might be fine for drinking birds, but if say we had a facial animation play and it was supposed to be synced with a sound file of the character's voice, this would be a disaster.

[Now show timed by checking box, sliding FPS bar - prerecord and then talk.]

What we really want to do is use the real-world time to know how far to rotate before rendering the next frame. For example, let's say our target rate is 60 frames per second, meaning that the bird should rotate 30 degrees per second. We'd like to know how much time has gone by since the last frame. In a moment I'll show you how this is done. As you can see from the demo, once we switch to the timed version of the code, the *rate* of the drinking bird's motion is unaffected by the frames per second shown. Give this demo a try yourself.

Demo: Constant Steps vs. Timed Steps

[unit10-db_step_fps_demo.js]

Lesson: Timed Animation

```
var clock = new THREE.Clock();

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

```
var clock = new THREE.Clock();

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

All it takes is this little bit of code to get the amount of time from the previous frame to this one. When you call clock.getDelta(), it returns a precise floating point value of the number of seconds that have elapsed since the clock was created or from the last time getDelta was called. By calling this at the beginning of each render, we can know exactly how much time we should use when we rotate the drinking bird.

```
bird.animated.rotation.z += tiltDirection * 0.5 * Math.PI/180;
```

```
bird.animated.rotation.z +=
    tiltDirection * 0.5 * Math.PI/180;
```

This first line is how we were rotating the bird, adding a half a degree each frame.

```
bird.animated.rotation.z += tiltDirection * 30 * delta * Math.PI/180;
```

```
bird.animated.rotation.z +=
    tiltDirection * 30 * delta * Math.PI/180;
```

If our goal is to rotate 30 degrees per second, we replace the 0.5 degrees per frame with exactly this. As each frame passes and the delta gets updated, the object is rotated the proper amount.

[do what it says below, try to capture on film: set to 60 FPS, turn on “timed”, don’t have

bulletproof: ext_unit10-db_step_fps_demo.js]

This works well while the browser window is visible, but an interesting thing happens if you start the demo in timed mode, set the frames per second, and then switch to another browser window and hide the program. When you come back in awhile, you're likely to see the drinking bird spinning around and around.

```
bird.animated.rotation.z += tiltDirection * 30 * delta * Math.PI/180;
if ( bird.animated.rotation.z > 103 * Math.PI/180 ) {
        tiltDirection = -1;
        bird.animated.rotation.z = 2*(103 * Math.PI/180) - bird.animated.rotation.z;
} else if ( bird.animated.rotation.z < -22 * Math.PI/180 ) {
        tiltDirection = 1;
        bird.animated.rotation.z = 2*(-22 * Math.PI/180) - bird.animated.rotation.z;
}
```

```
bird.animated.rotation.z += tiltDirection * 30 * delta * Math.PI/180;
if ( bird.animated.rotation.z > 103 * Math.PI/180 ) {
    tiltDirection = -1;
    bird.animated.rotation.z =
        2*(103 * Math.PI/180) - bird.animated.rotation.z;
} else if ( bird.animated.rotation.z < -22 * Math.PI/180 ) {
    tiltDirection = 1;
    bird.animated.rotation.z =
        2*(-22 * Math.PI/180) - bird.animated.rotation.z;
}
```

[point at "if" statements]

The problem is with our function that computes how much to rotate. The assumption in this code is that the rotation angle will never increase much beyond 103 or -22 degrees. However, when the window is hidden, rendering stops. When the window is exposed again, the renderer gets a *huge* delta, which the limit testing code doesn't deal with at all. The bird spins around incredibly fast for awhile as the huge value gets damped out.

```
var angle = (30 * clock.getElapsedTime()) % 250;
if ( angle < 125 ) {
        // go from -22 to 103 degrees
        bird.animated.rotation.z = (angle - 22) * Math.PI/180;
} else {
        // go back from 103 to -22 degrees
        bird.animated.rotation.z = ((250-angle) - 22) * Math.PI/180;
```

```
    }

var angle = (30 * clock.getElapsedTime()) % 250;
if ( angle < 125 ) {
    // go from -22 to 103 degrees
    bird.animated.rotation.z =
        (angle - 22) * Math.PI/180;
} else {
    // go back from 103 to -22 degrees
    bird.animated.rotation.z =
        ((250-angle) - 22) * Math.PI/180;
```

There are a number of solutions. Here's one that locks the drinking bird directly to the time itself instead of the change in time, the delta. The method getElapsedTime gives how many seconds have gone by since the Clock object was created. We take this time and multiply it by 30, the number of degrees per second. We know that the drinking bird sweeps through a 125 degree arc in both directions, for a cycle of 250 degrees total. By taking the modulus, the remainder after division by 250, we get an angle value between 0 and 250. If the angle is less than 125 degrees, we rotate the bird forwards, otherwise we rotate it backwards, adjusting for the starting and ending angles of -22 and 103 degrees. Now if the browser window is exposed later, the time is always going to resolve to a proper angle.

The moral is: when you design any animation driven by time, make sure it's bullet-proof from large time changes between renders.

[go back to demo and show wave - turn FPS to max, and make it around 70 speed, toggle wave on and off after a full cycle: ext_unit10-db_step_fps_demo.js]

By the way, there's one more mode in the demo called "wave". It's my attempt to make the bird swing back from the glass at a varying rate, instead of moving back at a constant number of degrees per second. This avoids a bit of the "jerk" that occurs when the direction is reversed. One nice thing about using the time as an input is that you can then decide what the angle should be at various times. I can't say my attempt is all that convincing, but at least it's a bit nicer.

Lesson: Motion Capture

[Start with b-roll of drinking bird and capture motion.

https://www.udacity.com/course/viewer#!/c-cs291/l-90856898/e-93309729/m-93290771 has a bit. I want the side view, of the whole motion of drink and release.]

The drinking bird has just one degree of freedom, it can rotate around its pivot point. However, the motion along this axis is fairly complex and is difficult to capture in a few simple equations.

One brute force method for creating animations is called motion capture. The idea is to capture the movements of a person or object by using a few strategically-placed cameras. The information recorded is processed to derive the paths of various parts of the body. These motions are then applied to the model in the virtual world.

[photos of the suits, etc.
http://en.wikipedia.org/wiki/File:Motion_capture_facial.jpg
http://commons.wikimedia.org/wiki/File:Joesph_gatt_mocap.jpg
http://commons.wikimedia.org/wiki/File:MotionCapture.jpg
http://commons.wikimedia.org/wiki/File:D%C3%A9monstration_de_%22motion_capture%22_(Futur_en_Seine,_pavillon_de_lArsenal)_(3585399745).jpg
]

To more easily track where each part of the body goes, it's common to attach ping pong balls or lights to a person while filming them. These attached objects are called "markers". There's a move towards markerless systems, where no special suits are needed. In the foreground you can see the actor's pose being captured and applied to a character.

[Show video of Unit9_MotionCapture, Virtual_Production_Autodesk_Maya_Entertainment_Creation_Suite, 42:40 to 43:05 - 25 seconds. Please *do* trim the "Autodesk Maya, Entertainment Creation Suite" words from the frame, just focus on the three screens.

[Depending on how long I talk, make 43:05 the ending spot (motion capture stops happening at this point on the left screen), and just start earlier in this clip.]

In this clip there is a lot going on - you might want to watch it a few times. In the upper right the two actresses are being recorded. The cameraman is also moving around the scene. In the lower right you can see the reconstruction of the actresses' motions. On the left, these motions are being applied to the two computer animation characters in real-time. There are two views here on the left: on the left half is the camera man's view, the right half shows a fixed camera view of the whole scene.

[show head tracker demo Unit9_MotionTracking, HeadTracker.wmv, with three.js - Either run showing the website link and startup, or just immediately go from the head tracks.]

Motion capture is used extensively in the film and games industry, as it is a straightforward way to quickly generate animations and is cost effective. There are some drawbacks. The first is the cost of entry: this process is currently not something you can easily do without some investment. That said, costs are coming down, and inexpensive hardware such as Microsoft's Kinect can be used for capture. You can also use more limited forms of capture. This demo by Jerome Etienne does head-tracking using a webcam. It's using three.js and is free source, so it's something you can instantly use yourself.

Motion capture can only do so much. Systems that use markers can track only those markers. For example, hand and finger gestures will not be caught by a full-body capture system.

Motion tracking is also affected by the limitations of the actors and the world itself. There are real-world constraints on real-world performances, and the data is only as good as your actors can provide. Artists generating animation have unlimited freedom, at least until they run out of time or money.

[End with b-roll of our drinking bird with a ping-pong ball attached to his head]

[recording of drinking birds.]

To get back to our humble drinking bird, the way this process could work is that we'd track the motion over time. The positions tracked could be turned into the amount of rotation of the Z axis for the bird's moving part. We would then have a giant table of values which we could interpolate between: at some given time T the angle of rotation is Z. Apply this rotation at that time and we have an animation matching real-life.

[Additional Course Materials:
Wikipedia's [article on motion capture](http://en.wikipedia.org/wiki/Motion_capture) is a reasonable summary. [Photo used in lesson](http://commons.wikimedia.org/wiki/File:D%C3%A9monstration_de_%22motion_capture%22_(Futur_en_Seine,_pavillon_de_lArsenal)_(3585399745).jpg) from Wikimedia Commons.

The wonderful head-tracker demo in three.js is at [Jerome Etienne's site](http://learningthreejs.com/blog/2013/03/12/move-a-cube-with-your-head/). Try it!

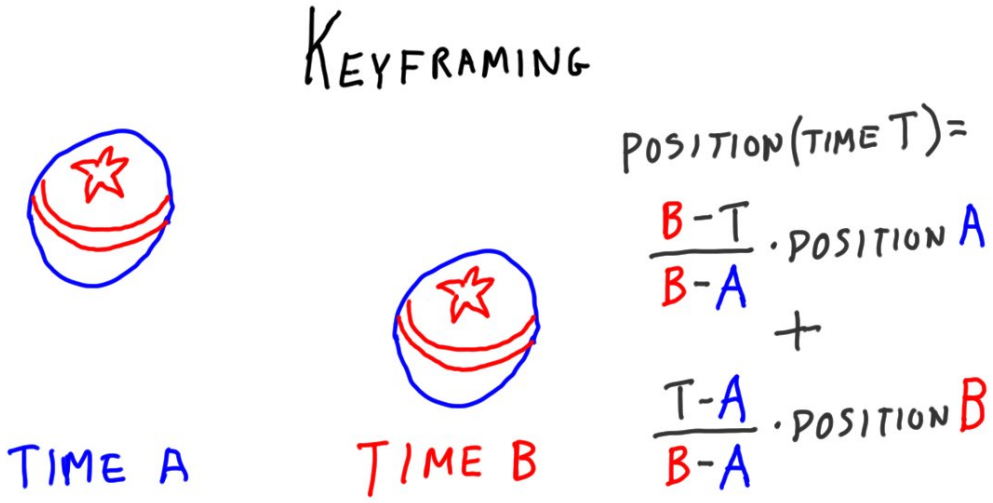
There are motion capture databases out there for experimentation, such as [this](http://www.mpi-inf.mpg.de/resources/HDM05/) and [this](http://accad.osu.edu/research/mocap/mocap_data.htm).

]

Lesson: Keyframing

[Drawing of a box in one position and another, ***time A and time B***. Make separate layers for video guys to do an interpolation.

[A : time: position]
[B : time: position]
]



Keyframing is a basic tool in the animator's box, perhaps the first one learned. The whole idea is this: to animate an object, save its state at various times and interpolate among these. For example, I have an object I want to move from one position to another over time. At time A I record the position and time, at time B I do the same. When I want to run the animation, given a time I can get the position:

position(time T) = (B - T)/(B-A) * position A + (T - A) / (B-A) * position B

This is a linear interpolation between positions A and B. The process of generating these frames in between the keyframes is called "***tweening***".

--- new page ---

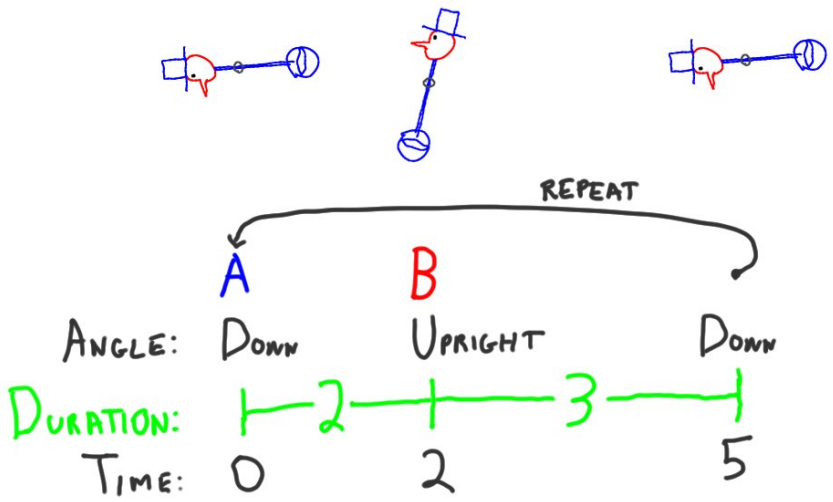
[The drinking bird body & head, show it rotated to two positions.

Label rotations D (drinking) and U (upright)
Show timeline A B horizontally, with D U D underneath at the break points:

A	***B***	***(restart arrow, point to A)***
D	***U***	***D***
0	***2***	***3***

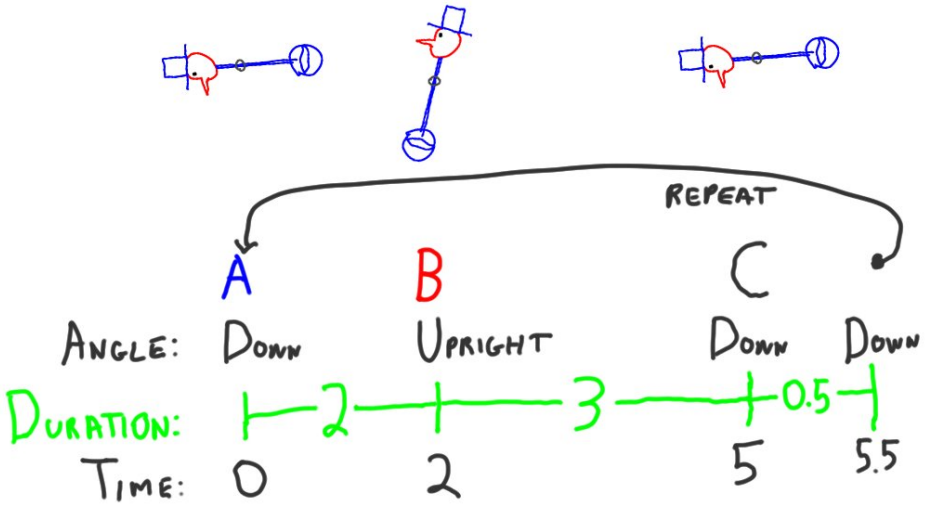
Show A and B a fair bit apart, later B and C close together.
]

[PUT in layers]
[A : duration 2 seconds: rotate to Up]
[B : duration 3 seconds: rotate to Down]
[<leave blank room here>]
[go to A and loop]



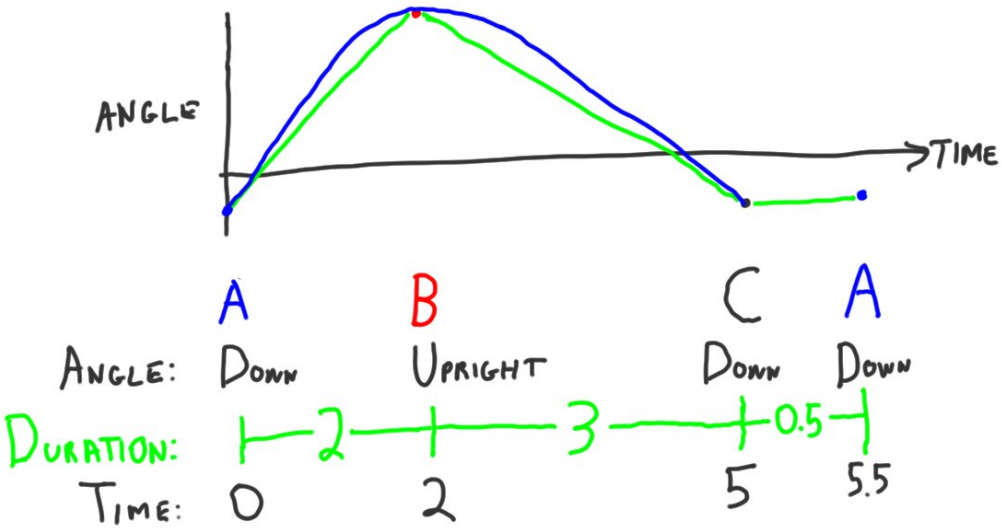
We can do the same thing with the drinking bird, interpolating between its two rotation angles. Down and Upright are the two positions [D and U] . Since we want a loop, we give two keyframes. A is the bird in the down position about to go up, B is the bird in the up position going down to drink. We can use these two frames to form a loop for our animation.

[add a third keyframe at end, call it C]
A B C (restart arrow)
D U D D
0 2 3 0.5
[A : duration 2 seconds: rotate to Up]
[B : duration 3 seconds: rotate to Down]
[C: duration 0.5 seconds: rotate to Down] (stays still)
[go to A and loop]



One improvement we can easily add with keyframes is make the drinking bird pause while it's drinking. We add another keyframe to our timeline, call it C, where the object is held at the down position. C and A are identical, the same "down" position, they're just in different locations on the timeline. The bird is held in place when the time is between C and A.

[draw a linear graph, spike up and down, show line up starting at A, down starting at B, ending at C and going flat. Gesture at point B.]



With our initial drinking bird animation, the end of the swing backwards is particularly unconvincing. The bird abruptly stops and moves forward again. With linear interpolation you get just that, linear motion. As you transition from one pair of keyframes to another, the motion can suddenly change.

[draw on top of spike a fairly tight spline curve around corner.]

To make the transition look smoother, it's common to use spline curves of various sorts to change the motion from keyframe to keyframe. You'll also see the terms "***ease in***" or "***ease out***", which mean to start or end the animation gradually from a stop.

[Run demo unit10-db_tween_demo.js]

Three.js has an add-on library called "***tween.min.js***" that lets you set up a series of keyframes. Here's my attempt to improve the drinking bird by using five keyframes total. I added an extra two keyframes at the end to get a wobble. You'll get to take a closer look and critique it in a minute.

[Show video Unit9_Keyframing, Keyframing_Luxo.wmv - let it run]

Here's a more elaborate keyframe animation by Jason Kadrmas, showing multiple keyframes and different rotations and translations. This was created in Blender, a free modeling program, and exported to a file format three.js can read. I simply dropped this file into Jason's program threeFab, running in a browser window, and can play through the animation at different speeds. Watching the animation play, you can see how just a few keyframes can define some fairly

elaborate motions.

[Additional Course Materials:
[This website](http://learningthreejs.com/blog/2011/08/17/tweenjs-for-smooth-animation/) has a good explanation of tween.js and [a worthwhile demo of easing](http://learningthreejs.com/data/tweenjs_for_smooth_animation/tweenjs_for_smooth_animation.html). More about Blender export [here](http://www.96methods.com/2012/02/three-js-very-basic-animation/). Here's a tutorial of [another tween library, Greensock](http://www.kadrmasconcepts.com/blog/2012/05/29/greensock-three-js/).

[Wikipedia's overview of [keyframing](http://en.wikipedia.org/wiki/Key_frame) is a reasonable start, though is mostly concerned with 2D content. The [article on interpolation for computer graphics](http://en.wikipedia.org/wiki/Interpolation_(computer_programming)) is bare-bones, but talks about some basics. Much more can be found in the [Essential Math for Games Programmers](http://www.essentialmath.com/tutorial.htm) course notes.

[ThreeFab](https://github.com/blackjk3/threefab) lets you animate scenes, it was used for the lamp animation. It's an alpha, and looks inactive, but has some interesting features. Read more about [how to export from Blender and use ThreeFab](http://www.kadrmasconcepts.com/blog/2012/01/24/from-blender-to-threefab-exporting-three-js-morph-animations/). Try it [here](http://blackjk3.github.com/threefab/).

This demo](http://mrdoob.github.com/three.js/examples/webgl_loader_collada_keyframe.html) shows an assembly animation done in three.js.
]

Demo: Three.js Tween Library

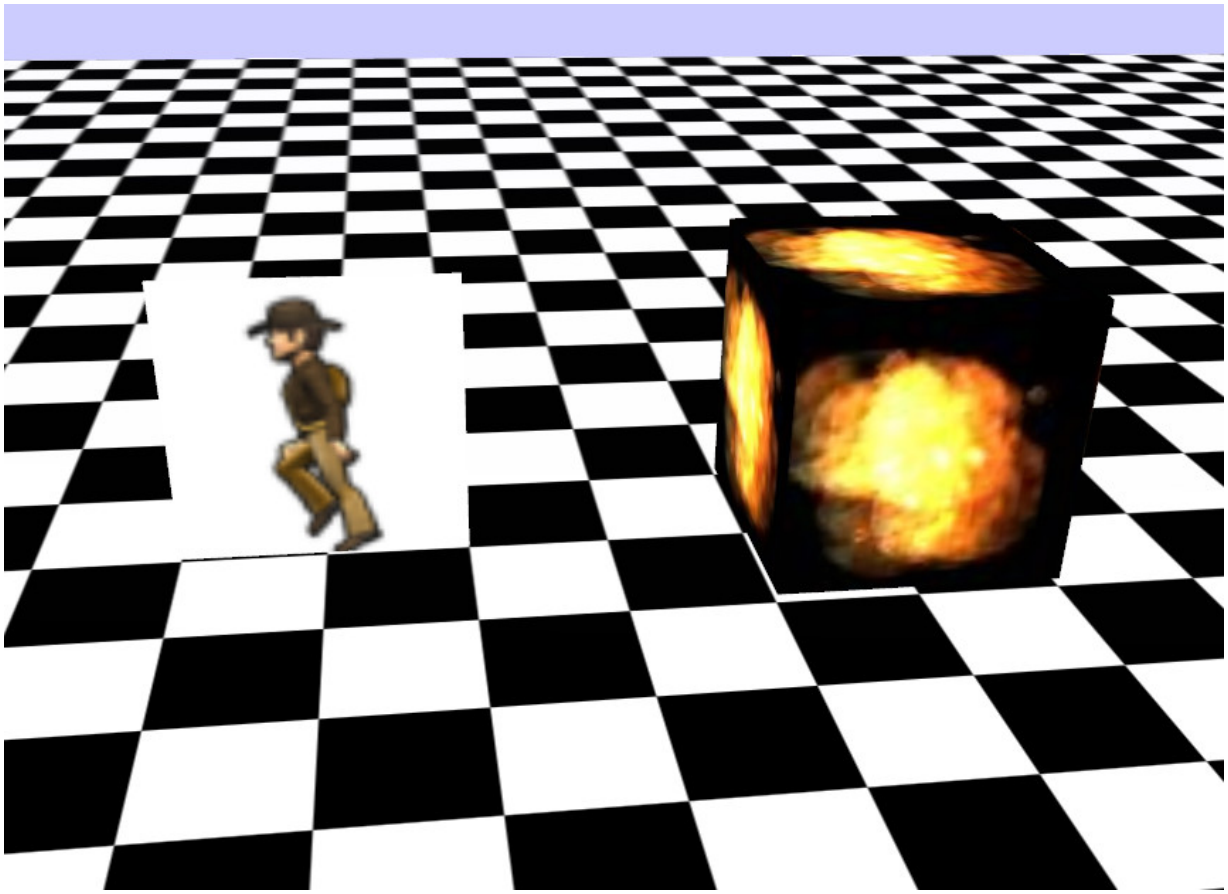
[Run unit10-db_tween_demo.js]

[Additional Course Materials:
I encourage you to examine my example to see how to use the tweening library.
The three.js tweening library is described in [this tutorial](http://learningthreejs.com/blog/2011/08/17/tweenjs-for-smooth-animation/). Try [this demo](http://learningthreejs.com/data/tweenjs_for_smooth_animation/tweenjs_for_smooth_animation.html) to see the effects of the many easing curves supported.
]

Lesson: Texture Animation

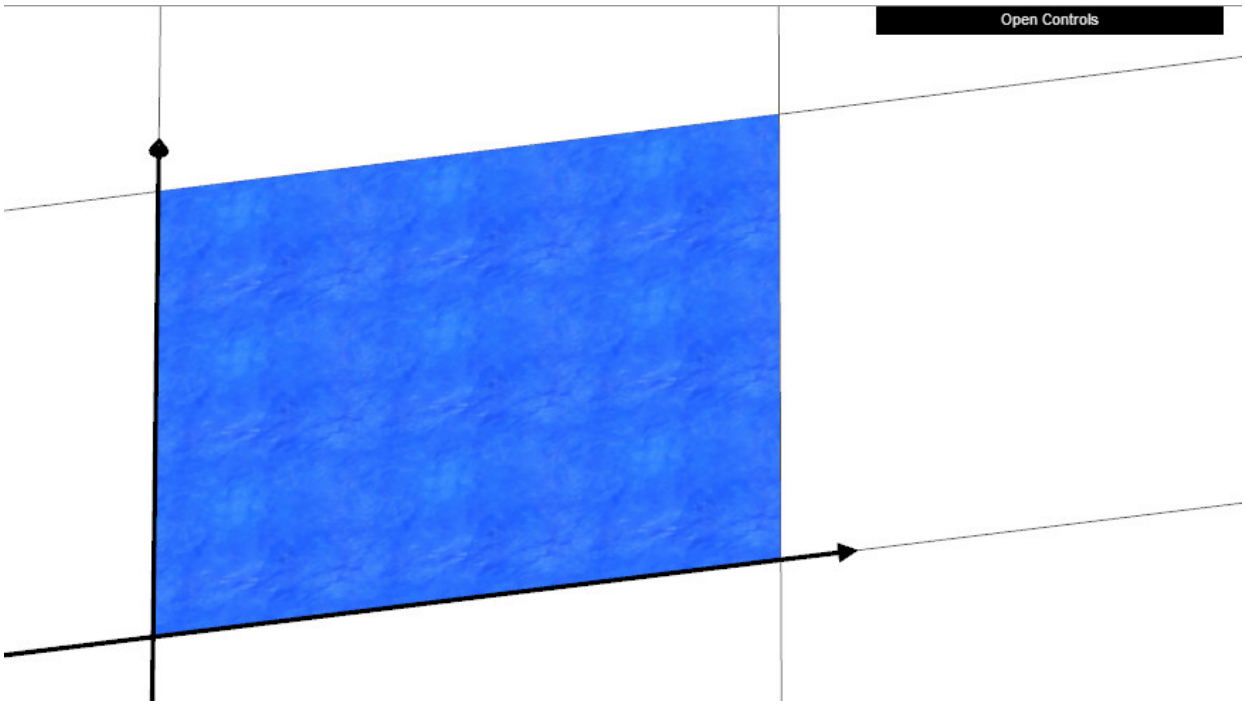
[May want to move this somewhere else.]

[http://stemkoski.github.com/Three.js/Texture-Animation.html - run this one and talk over it. Move around a bit, just to show it's interactive.]



When I see the phrase "texture animation" it brings to mind two separate ideas. The first is shown here, in a tutorial by Lee Stemkoski. The idea is the same as a flipbook, where you draw a different image on each page and then flip through them.

[PROBABLY NEED TO EXT THIS DEMO unit10-txr_wave_solution.js - zoom in and tilt view]



The other form of texture animation is where you move the texture by transforming the texture coordinates, such as shown here.

There's plenty of animation that can be done without a motion capture studio, or even a modeler. Remember that almost *anything* you can set on a model, light, or camera is something you can change in the render loop. You can modify material colors and shininess, transparency and visibility, and just about everything else.

[Additional Course Materials:
The [animated texture tutorial program
http://stemkoski.github.com/Three.js/Texture-Animation.html](is by Lee Stemkoski. He has many other [commented tutorials](http://stemkoski.github.com/Three.js/index.html) available.

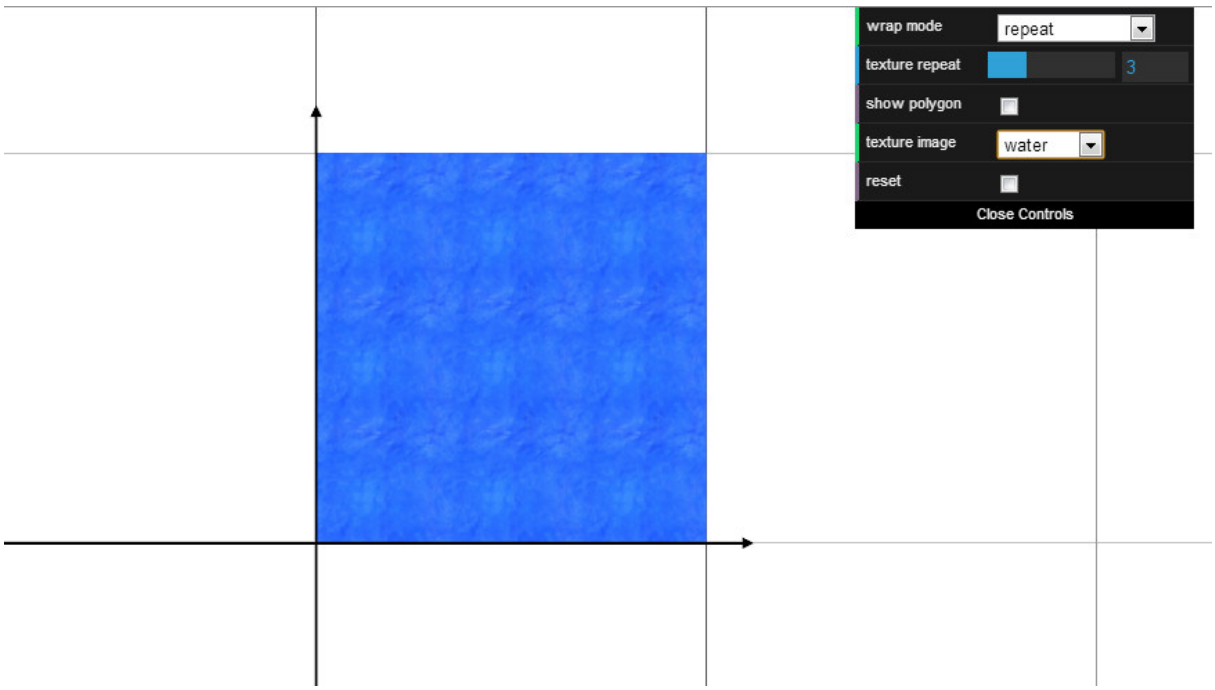
Just in case you don't know what a flipbook is, [read this](http://en.wikipedia.org/wiki/Flip_book) and then [make your own](http://www.benettonplay.com/toys/flipbookdeluxe/guest.php).
]

[end recording session 3/31 part 2]

Exercise: Flowing River

[***Show this demo for a moment, then turn on letterR. Change the repeat mode to 1, so that the texture is shown moving down one per second. TELL VIDEO PERSON TO SYNC SWITCHING TO "R" WITH "when the texture repeat is 3".***

show EXT version ext_unit10-txr_anim_solution.js
]



In this exercise you'll make a river flow. Well, it's more like a square lake instead of river. And really the texture should be more stretched to look like flowing water. And the banks of the river - oh, nevermind! This is an exercise, after all, so I want to keep it simple.

The initial code doesn't have anything moving. Your job is to use the texture transform feature in three.js to animate the water as shown, moving from top to bottom. The rate of movement is one second per copy of the texture. In other words, when the texture repeat is 3, a point at the top of the square will take 3 seconds to move from top to bottom. When the repeat is changed to one, the top of the texture moves down the screen in one second.

For your solution the movement should be tied to the elapsed time in the real world. If you look just above where your code should go, you'll see how the texture repetition parameter is set on

the texture. You'll need similar code in your solution.

[exercise at unit10-txr_anim_exercise.js]

Answer

[unit10-txr_anim_solution.js]

[put code on screen]

```
var time = clock.getElapsedTime();
texture[effectController.mtlName].offset.set( 0, time );
```


```
var time = clock.getElapsedTime();
texture[effectController.mtlName].offset.set( 0, time );
```


The main elements here are figuring out how to get the elapsed time and how to offset the texture. About the only tricky bit is the realization that if you want the texture to move downwards, you *add* the time to the offset. This has the effect of moving the texture's frame of reference up, which in turn moves the display of the texture down.

My solution is actually offsetting the V value a potentially huge amount as the time increases, but it doesn't matter since the texture repeats.

A less efficient solution is:

```
var time = clock.getElapsedTime();
// causes garbage collection
texture[effectController.mtlName].offset = new THREE.Vector2( 0, time );
```


```
var time = clock.getElapsedTime();
// causes garbage collection
texture[effectController.mtlName].offset =
    new THREE.Vector2( 0, time );
```

It's best to avoid creating new objects every frame, such as this vector. Eventually JavaScript has to perform garbage collection, which can cause your animations to hesitate at intervals.

```
var time = clock.getElapsedTime();
// add some wobble
texture[effectController.mtlName].offset.set(
        0.2*Math.sin(2*time), time );
texture[effectController.mtlName].repeat.set(
        effectController.repeat, effectController.repeat/3 );
```

```
var time = clock.getElapsedTime();
// add some wobble
texture[effectController.mtlName].offset.set(
    0.2*Math.sin(2*time), time );
texture[effectController.mtlName].repeat.set(
    effectController.repeat, effectController.repeat/3 );
```

Once I had the basic solution I started to play with the horizontal offset, too. Giving this offset a slight wobble with a sine wave made it a little more appealing. I also changed the repetition length along the vertical axis by dividing the repeat factor by 3. This gives a more stream-like look to the water.

[ext_unit10-txr_wave_solution.js]

It looks a bit like a rushing river or waterfall now.

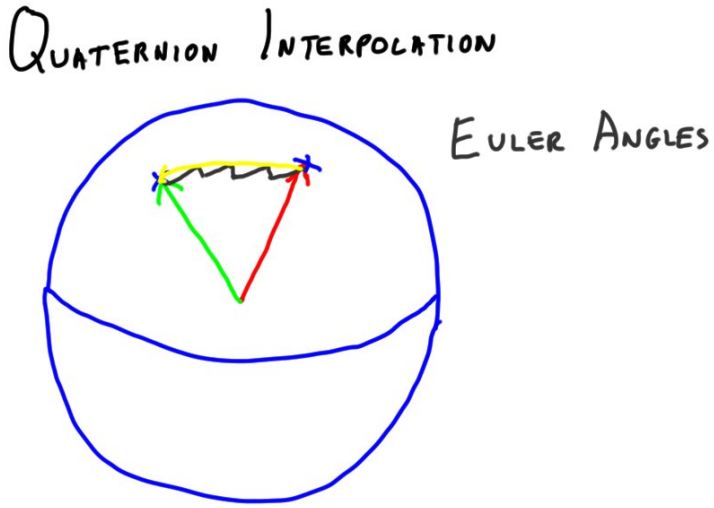
Look in the additional course materials for my snippet of code and try it out yourself, then try other functions - it's pretty addictive!

[Additional Course Materials:
Here's the code snippet to copy and paste to see my more artistic solution:

```
texture[effectController.mtlName].offset.set( 0.2*Math.sin(2*time), time );
texture[effectController.mtlName].repeat.set( effectController.repeat, effectController.repeat/3 );
```
]

Lesson: Quaternion Interpolation

[draw sphere and two vectors, and interpolating between]



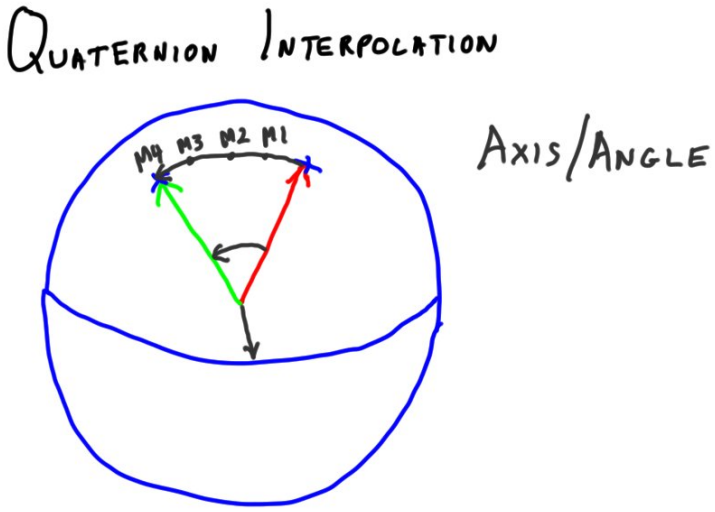
Quaternions represent the orientation of an object, how it is rotated. They do the same thing as axis/angle rotations: they rotate a model around a given axis by an angle. They get a lot of use in animation because you can easily interpolate between one orientation and another.

Think of two vectors poking through a sphere at the origin. If you want to change your orientation and go from one vector to another,there are a number of ways to do it. Say you want to do this over 4 frames.

[add Euler angles]

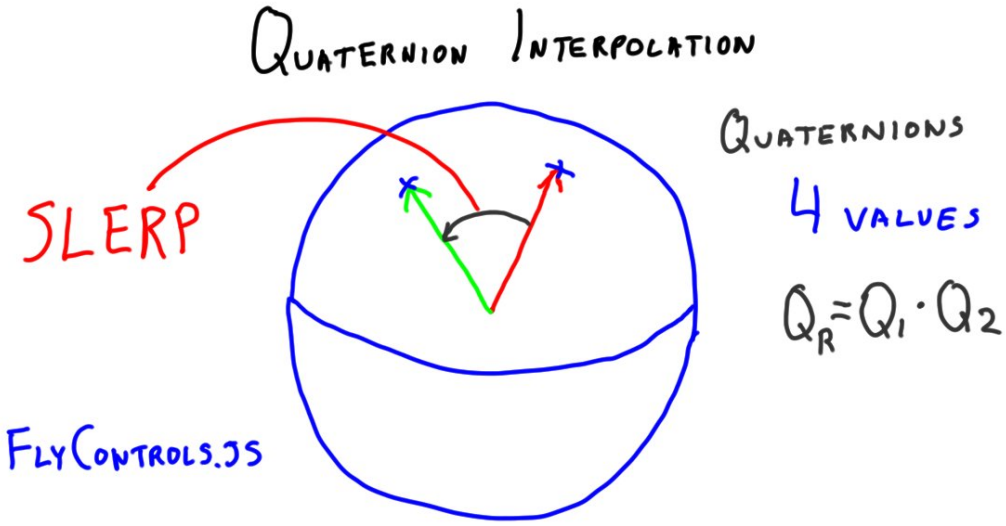
You can’t normally do this well with Euler angles. The change in direction is likely to need more than one Euler angle changed at a time. Changing two Euler angles at once can give you problems with gimbal lock, as one angle is applied before the other. Euler angles are terrible for interpolation around an arbitrary axis, especially when the change is large.

[add axis/angle]



Using an axis/angle scheme, you would go a quarter of the angle and form the matrix. Every frame would need to form a new matrix, which can get pricey: creating each takes a number of trigonometric function calls.

[add quaternion, slerp]



You can set quaternions to perform the same interpolation in a clean way, and in a form as

compact as axis/angle - just 4 values. No trigonometry calls are needed. Quaternions themselves are fairly non-intuitive as to what their 4 numbers mean in physical terms when compared to axis/angle. However, just like you rarely look at a single number in, say, a rotation matrix, the meaning of the quaternion's individual values don't really matter when you're using them.

When you interpolate around an axis, it's called a ***slerp,*** short for "spherical linear interpolation". All this means is that you are moving from one point on the sphere to another in a direct and linear fashion.

Three.js and many other packages support quaternions because of this simplicity. In three.js if you wanted to use axis/angle rotations, you'd also have to maintain your own translation and scale matrices for the object being rotated. Because quaternions are fully supported as a separate interpolation system, you can enable these in three.js while still using the position and scale parameters.

[quaternion multiplication]

Quaternions make it easy to specify and interpolate between various orientations. Similar to rotation matrices, but with less values involved, you can also multiply quaternions together. Doing so produces a new quaternion that is the result of this series of rotations.

[***FlyControls.js***]

Quaternions are often used for animation paths or flying controls. In fact, three.js uses them for just that class, FlyControls.js. That said, you do have to be careful in using quaternions with cameras, in that the "up" vector will not be maintained throughout the slerp and so you'll need to reorient.

Try the demo that follows. It moves the bird's hat back and forth on his head in a jaunty way and hilarity ensues. You can choose to interpolate using Euler angles or quaternions. With Euler angles on, try setting the angles to their maximums. If you look from above, you'll see that the path is not straight around the head but rather takes an S-shaped curve.

[Additional Course Materials:
Wikipedia discusses [slerping](http://en.wikipedia.org/wiki/Slerp) a bit if you want to know a little more.

[This presentation on
rotations](http://www.essentialmath.com/GDC2012/GDC2012_JMV_Rotations.pdf) gives a thorough run through of rotation interpolation and quaternions.
]

Demo: Jaunty Hat

[unit10-db_quaternion_demo.js
]

Lesson: Skinning

[Draw along these lines. From “Real-Time Rendering”
However, go through the solutions in video terms, and we just need the 2nd, 4th, then maybe 3rd, and 5th drawings, adding in the bones. Start rigid. Erase the rigid bits at ends and add triangles to stretch (just two - a schematic view). Then add the 3 triangles as shown and label.

Note the fold in the elbow - important.
]

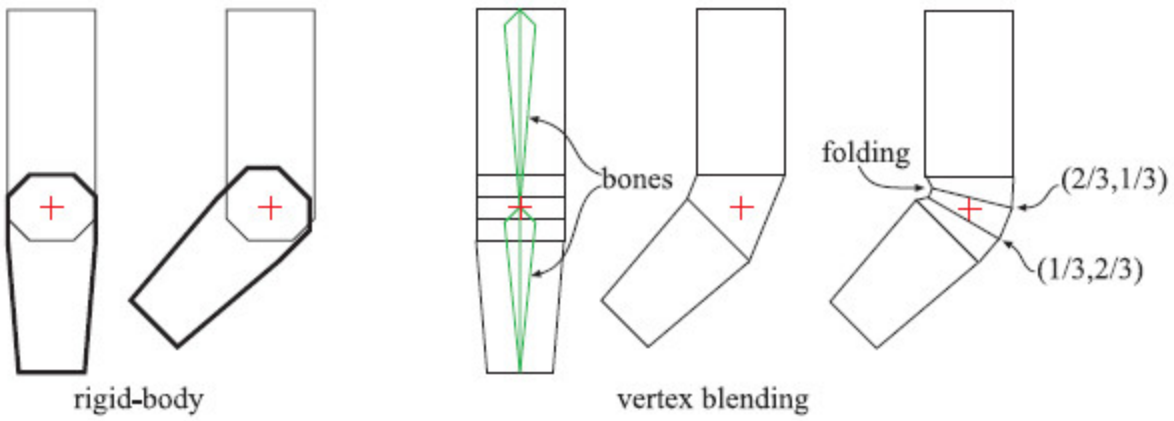
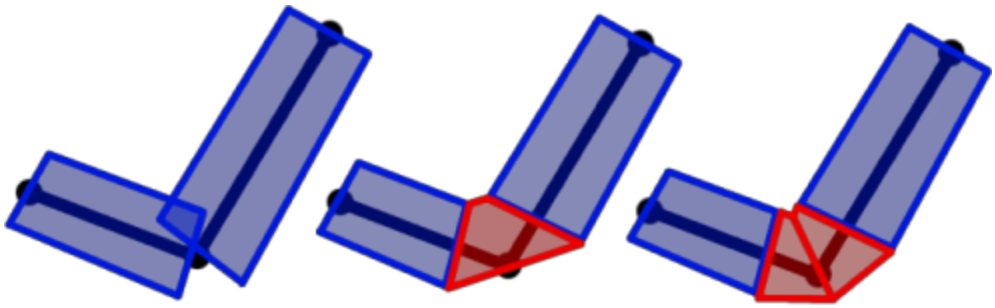
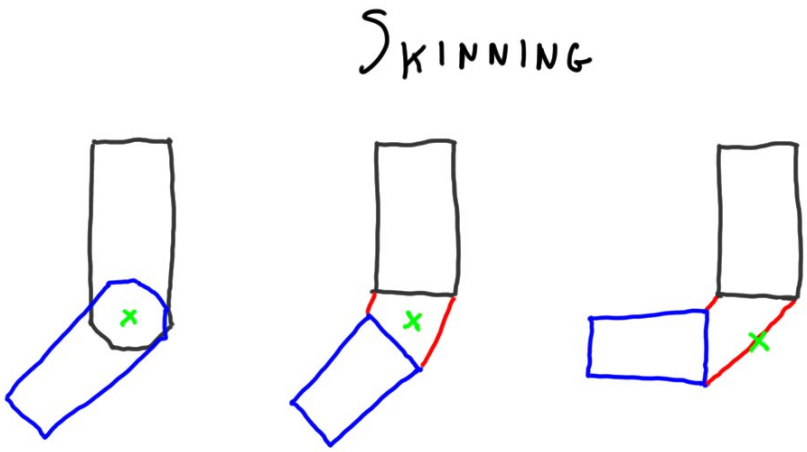


Figure 4.10. An arm consisting of a forearm and an upper arm is animated using rigid-body transforms of two separate objects to the left. The elbow does not appear realistic. To the right, vertex blending is used on one single object. The next-to-rightmost arm illustrates what happens when a simple skin directly joins the two parts to cover the elbow. The rightmost arm illustrates what happens when vertex blending is used, and some vertices are blended with different weights: $(2/3, 1/3)$ means that the vertex weighs the transform from the upper arm by $2/3$ and from the forearm by $1/3$. This figure also shows a drawback of vertex blending in the rightmost illustration. Here, folding in the inner part of the elbow is visible. Better results can be achieved with more bones, and with more carefully selected weights.



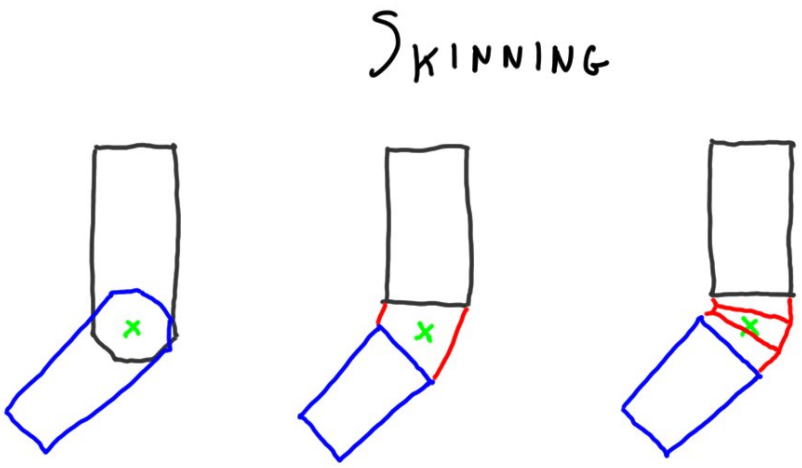
[More reference art from http://content.gpwiki.org/index.php/OpenGL:Tutorials:Basic_Bones_System#How_does_a_skin_work.3F - I like the trim back of the rigid models to place the skin, good colors.]



Say you've designed your robot, now you want to put a skin over his frame to make him look more lifelike and [mech voice] infiltrate the humans, I mean, become more accepted by humanity. A robot model typically has a few rigid elements that do not change over time, such as a forearm and upper arm. At the joint the objects are clearly separate; that's where you'd like to put skin.

One simple solution is to just put some triangles connecting the two rigid pieces, sort of like a cylinder. As the joint bends, these triangles deform and stretch, keeping the two arm pieces attached.

[maybe drawn more bent version temporarily, to show problem]

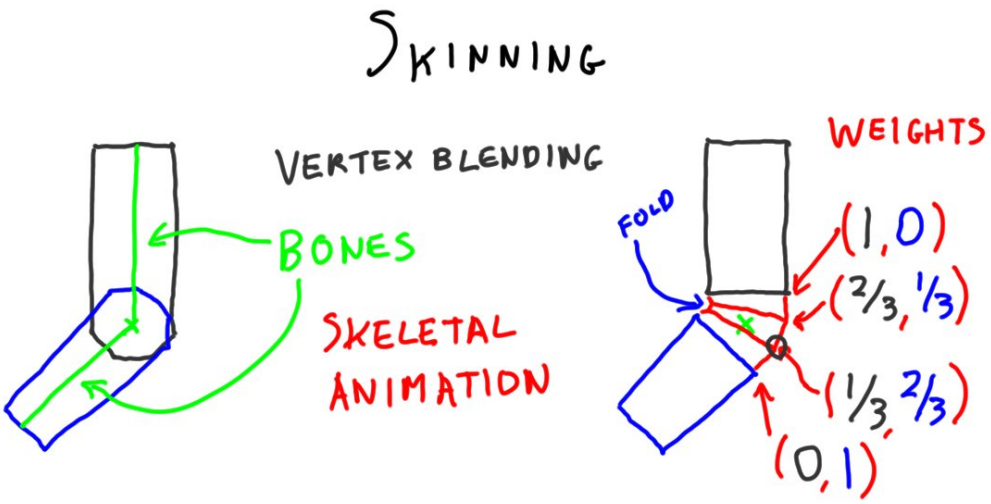


This doesn't look very good, as the more the arm bends, the more the joint flattens out. What we'd prefer is something more like a flexible tube that bends but mostly keeps its shape. Say we now add such a joint, with three cylinders one after the other. The challenge now is how to move these cylinder vertices as the joint bends.

The basic solution is surprisingly simple. Each vertex is assigned a weight. This weight says how much each rigid part's position affect the vertex. The top and bottom of the tube still stays connected to its rigid body part, so that body part's influence has a weight of 1. For vertices closer to the forearm, the weight is say 2/3rds forearm, 1/3rd upper arm.

This modeling process of adding polygons and assigning weights is called ***skinning***, or sometimes ***vertex blending***. The rigid parts are defined by ***bones***. These bones are like a skeleton, so this whole area is sometimes called ***skeletal animation***. As a bone moves, the skin is influenced by the bone's transform times its weight. Specifically, let's look at this one vertex on the skin that is nearer the forearm.

[Zoom in. Show a point, show its two transformed locations., then interpolate ⅓, ⅔]



The vertex location is transformed twice, once with respect to the forearm and once with respect to the upper arm. These two transformed locations are multiplied by their corresponding weights, giving the interpolated point between them.

The basic algorithm works fairly well, but can have problems. For example, in the inner part of the elbow the points may bend inwards in an unconvincing way. In practice a vertex can be influenced by more than two bones; the bones and weights are something the person modeling decides, and the weights always add up to one.

[http://alteredqualia.com/three/examples/webgl_animation_skinning_tf2.html]

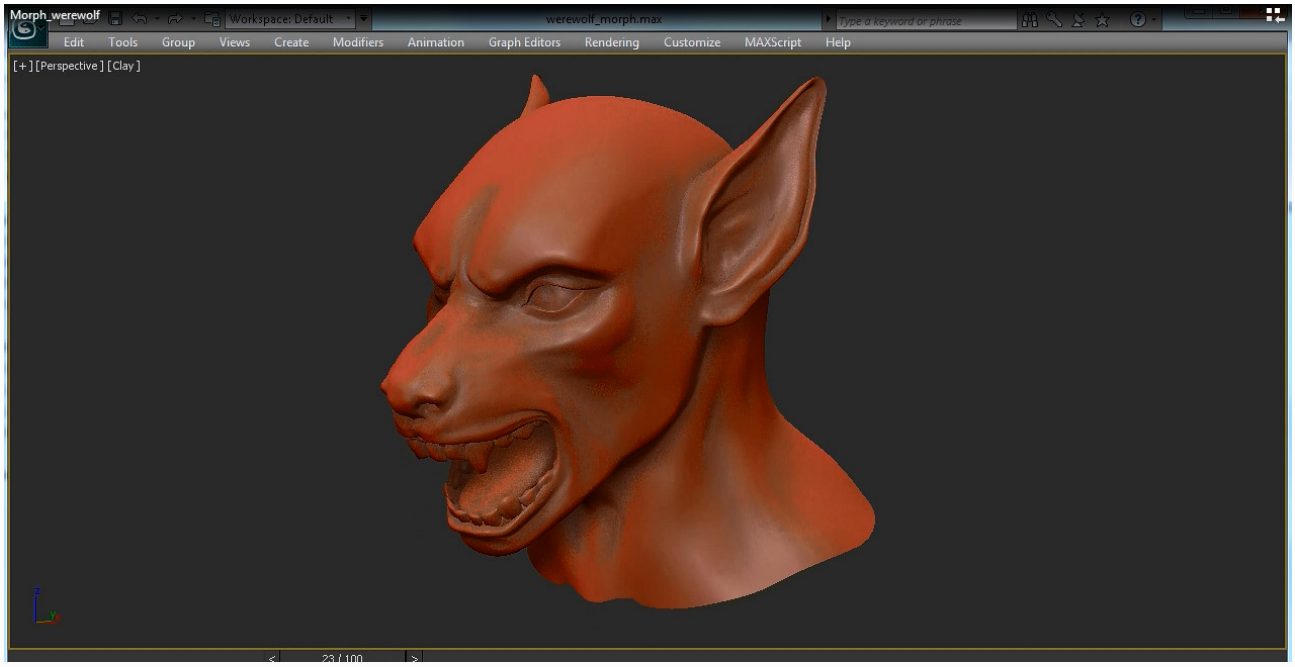
These characters were exported from Valve's Team Fortress 2 game and animated with three.js's skinning class, called SkinnedMesh.

Skinning is well-suited to the GPU, as the vertex shader can transform points multiple times and add together the weighted locations. There are more elaborate algorithms that give better results, but for interactive rendering in particular skinning is a mainstay.

[Additional Course Materials:
The Team Fortress 2 demo is
[here](http://alteredqualia.com/three/examples/webgl_animation_skinning_tf2.html). Another skinning demo is
[here](http://mrdoob.github.com/three.js/examples/webgl_animation_skinning.html).
]

Lesson: Morphing

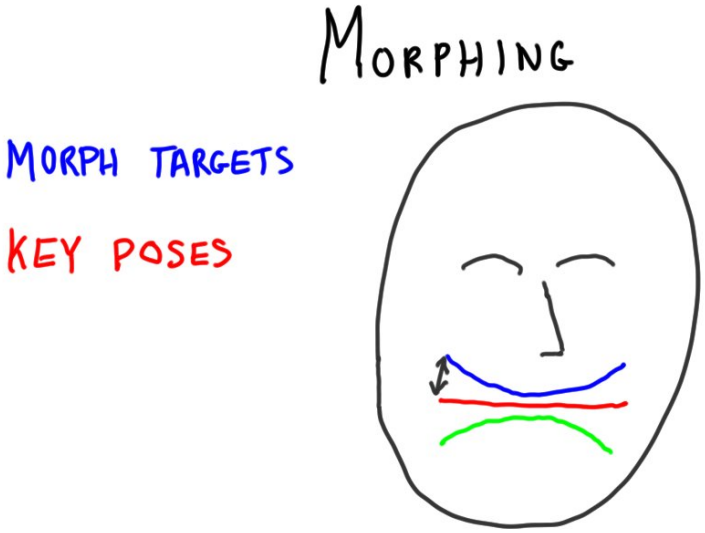
[video Unit9_SkinningAndMorphing, Morph_werewolf.mp4]



Morphing is the process of changing from one model to another. It's a fairly tricky problem to solve if the two objects are quite different. For example, if I want to morph a cube into a unicorn model, I'll have to create a bunch of vertices on the cube and decide how these move in a convincing way.

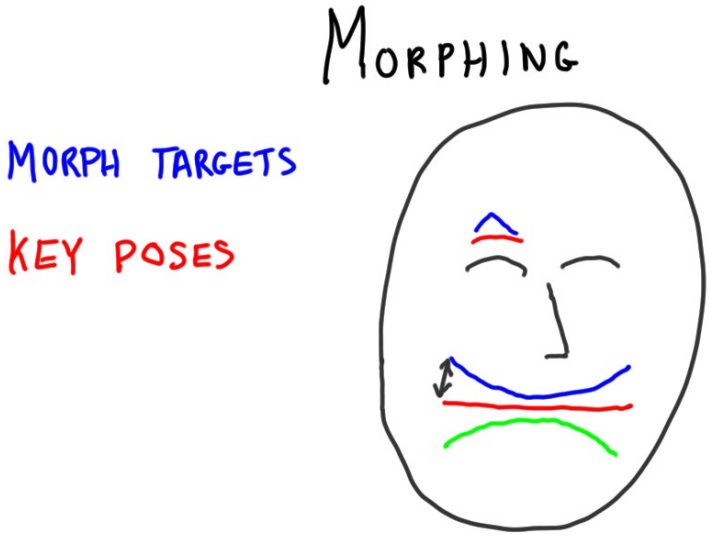
Much simpler is to interpolate between two meshes with the same number of vertices. In this case linear interpolation between two meshes is trivial. A more general approach uses what are called ***morph targets*** or ***key poses***.

[smile example, put it low so we can add eyebrow later]



A simple example is a smile. Say we have three poses: a neutral pose, a smile, and a frown. To go from one to the other we can interpolate. Essentially, the key poses are similar to keyframes to interpolate between. By changing the amount of influence a particular target has affects the result. For example, I could specify a half smile by saying to have each vertex be half the neutral pose, half the smile pose.

[add an eyebrow]



A further advantage of morph targets is that they can be combined. For example, I could also have key poses for an eyebrow raise and use it as I wish with a smile or frown. A pose can be associated with parts of a mesh and so influence only them.

[Put Ginger demo here http://stickmanventures.com/labs/demo/webgl-threejs-morph-target/# Turn mouse tracking off. Good strong ones are jaw range, eyes, expression, and sex.
else use http://mrdoob.github.com/three.js/examples/webgl_morphtargets_md2.html to switch between different animations]

For example, here I'm using the wonderful Ginger demo by Stickman Ventures. There are sliders set on many different facial features. By changing a setting, I'm changing the influence of a morph target.

[Alternate text here:]

Even skinned models can be exported and treated as morph targets. Morph targets are well supported by three.js and so are a popular way to display animated models. The main drawback on the modeling end is that it's labor-intensive to create morph targets. The whole process of setting up a character's animation controls is called rigging, and is one that takes skill and time to do properly. Such processes are almost always done with interactive modeling and animation programs.

Another drawback is that morphing needs much more data than skinning while running an animation, both to download and to keep in the GPU's memory.

All that said, morphing has the advantage of being generic; *any* model changes can be saved into morph targets and there's no real ambiguity. Skinning can and is done in a myriad different ways, with research continuing to find new techniques. Lots of tears have been shed by many developers trying to convert skinning animations from one format to another.

[Additional Course Materials:
[The Ginger demo](http://blog.stickmanventures.com/2011/09/07/simple-facial-rigging-utilizing-morph-targets-powered-by-three-js/) shown in this lesson gives a great demo of the power of morph targets. Further technical information is [here](http://blog.stickmanventures.com/page/2/).

Lee Stemkoski has a [morph animation code tutorial](http://stemkoski.github.com/Three.js/Model-Animation-Control.html) - use the arrow keys. Three.js has a few relevant demos: [morphing between animations](http://mrdoob.github.com/three.js/examples/webgl_morphtargets_md2.html) and [skinning and morphing](http://mrdoob.github.com/three.js/examples/webgl_animation_skinning_morph.html).

A bit more about morph targets can be found on [Wikipedia](http://en.wikipedia.org/wiki/Morph_target_animation).

See the [Loader class](http://mrdoob.github.com/three.js/docs/) and the various model loaders available for three.js. [Webgl-loader](http://code.google.com/p/webgl-loader/) convert models for fast loading into WebGL web pages. There's more information on the [three.js FAQ page](https://github.com/mrdoob/three.js/wiki).

Various ways to load geometry into WebGL are discussed in Chapter 2 of the "[WebGL Beginner's Guide](http://www.amazon.com/WebGL-Beginners-Guide-Diego-Cantor/dp/184969172X?tag=realtimerenderin)" and Chapter 7 in "[WebGL: Up and Running](http://www.amazon.com/WebGL-Up-Running-Tony-Parisi/dp/144932357X?tag=realtimerenderin)". I recommend the Kindle version of each; see [my review](http://www.realtimerendering.com/blog/books-the-good-the-bad-and-some-third-category/).

[Register](http://students.autodesk.com/?nd=register&tagent=EDU-FY12_Launch_EC_Banner-JG-5-19-2011) as a student or teacher for [free use of Autodesk software](http://students.autodesk.com/?nd=download_center). When you register, note that Udacity is considered to be in Sunnyvale, CA for purposes of registration.
]

Video Clip: Previsualization Introduction

[headshot recorded 4/2]

One area where interactive 3D graphics is used extensively is in what is called “previsualization”. The idea is that a director can lay out and direct parts of a movie using a virtual set instead of a real one, vastly cutting costs and speeding delivery. What follows are some clips I selected from a longer interview with The Third Floor, a previsualization company.

[video: Unit9_Previsualization, ThirdFloor_Short_1280x720_MP4_5000kbps.mp4 - use 0:00 to 0:38, the first clip, and 1:19 to 2:08, the virtual camera. Yes, there will be a glitch in the background music track - do what you can. Cut the rest, it’s too commercial.]

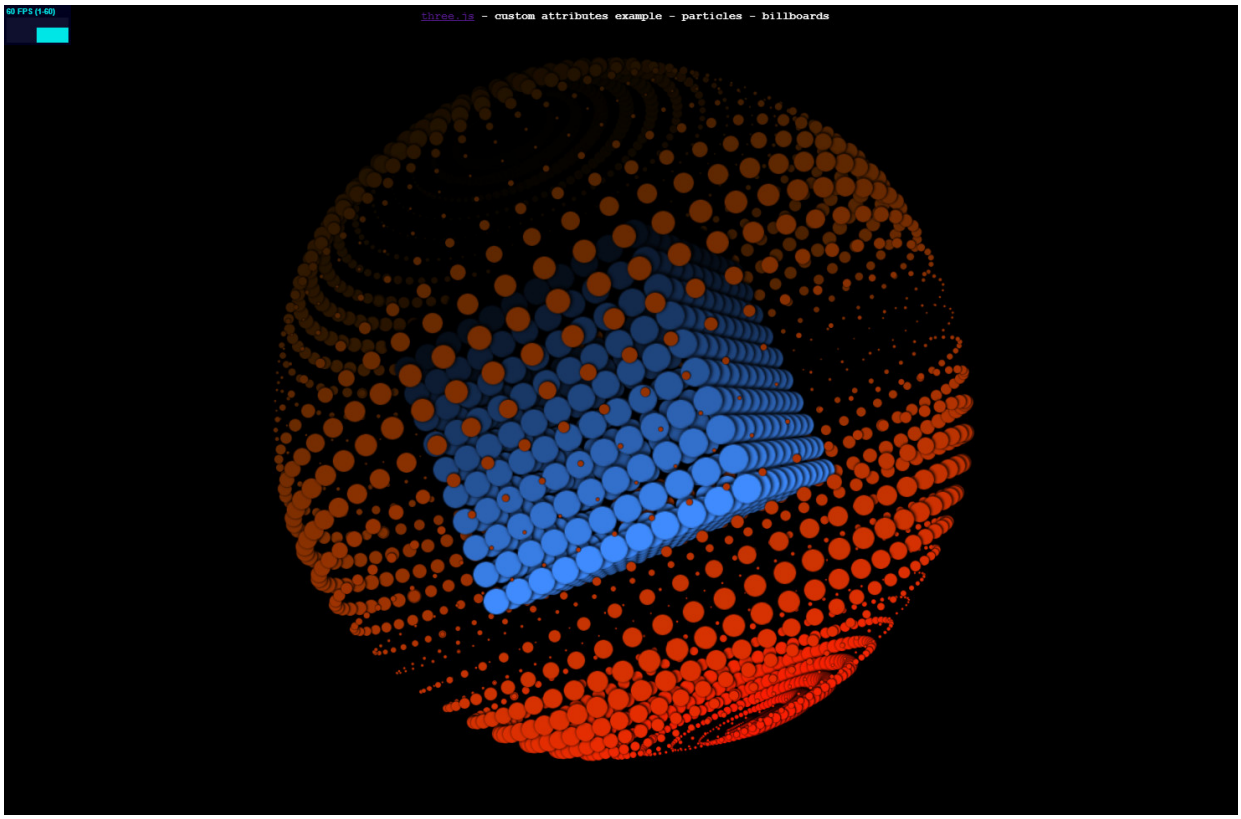
Video Clip: Previsualization Runthrough

[video: Unit9_Previsualization, ThirdFloor_Demo_1280x720_MP4_5000kbps.mp4 - good as a whole separate clip. To be honest, it’s already on YouTube at http://www.youtube.com/watch?v=WgJexAC0-0k, if you want to just use that.]

[Additional Course Notes:
The full interview clip is [here](http://www.youtube.com/watch?v=6vBx4Dm47iM). There is an extensive interview with Chris Edwards about previsualization
[here](http://www.youtube.com/watch?v=5qkFibEY3ZM),
[here](http://www.youtube.com/watch?v=SmL3aOvXCG0), and
[here](http://www.youtube.com/watch?v=WyyGJYHgygw).
]
[end recording part 3, 3/31]

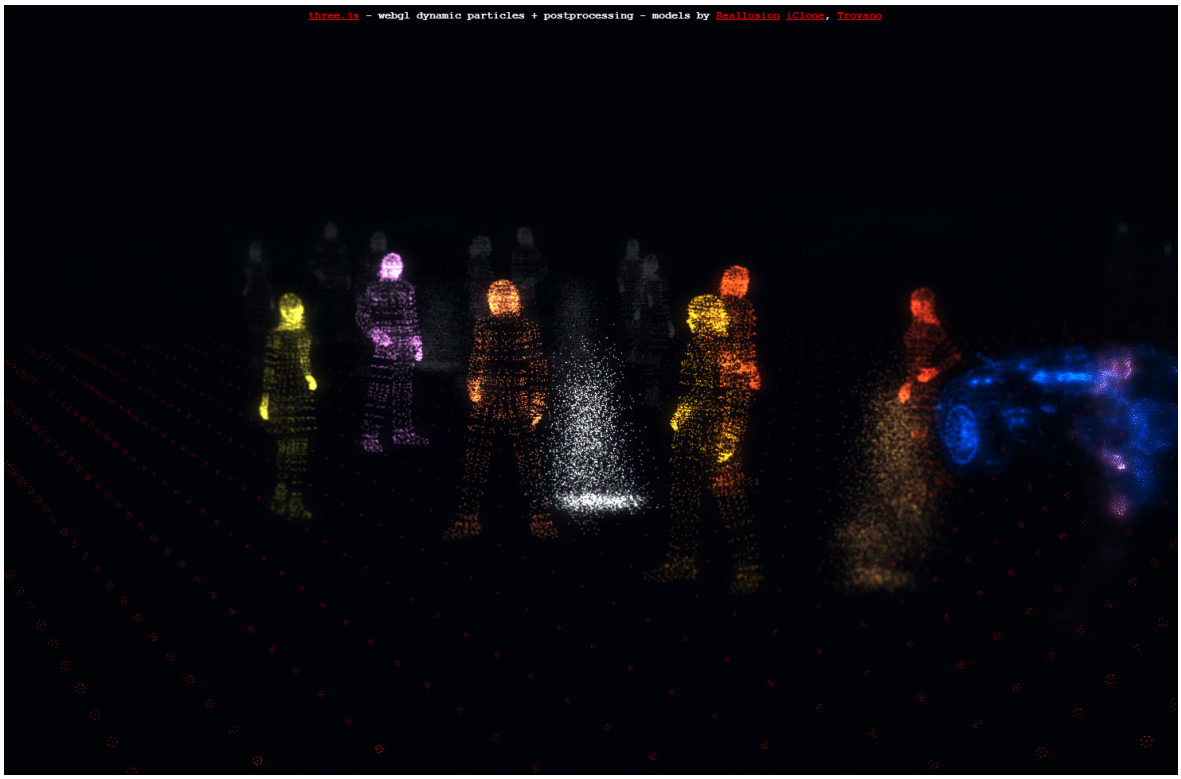
Lesson: Particle Systems

[http://mrdoob.github.com/three.js/examples/webgl_custom_attributes_particles2.html - organize those particles and life gets interesting]



We've talked about how to create particles and some of the effects achievable. You can create all sorts of animations with particle systems. Here is an abstract presentation by Altered Qualia which includes varying the sizes of the particles over time.

[http://mrdoob.github.com/three.js/examples/webgl_particles_dynamic.html - you could form other things]

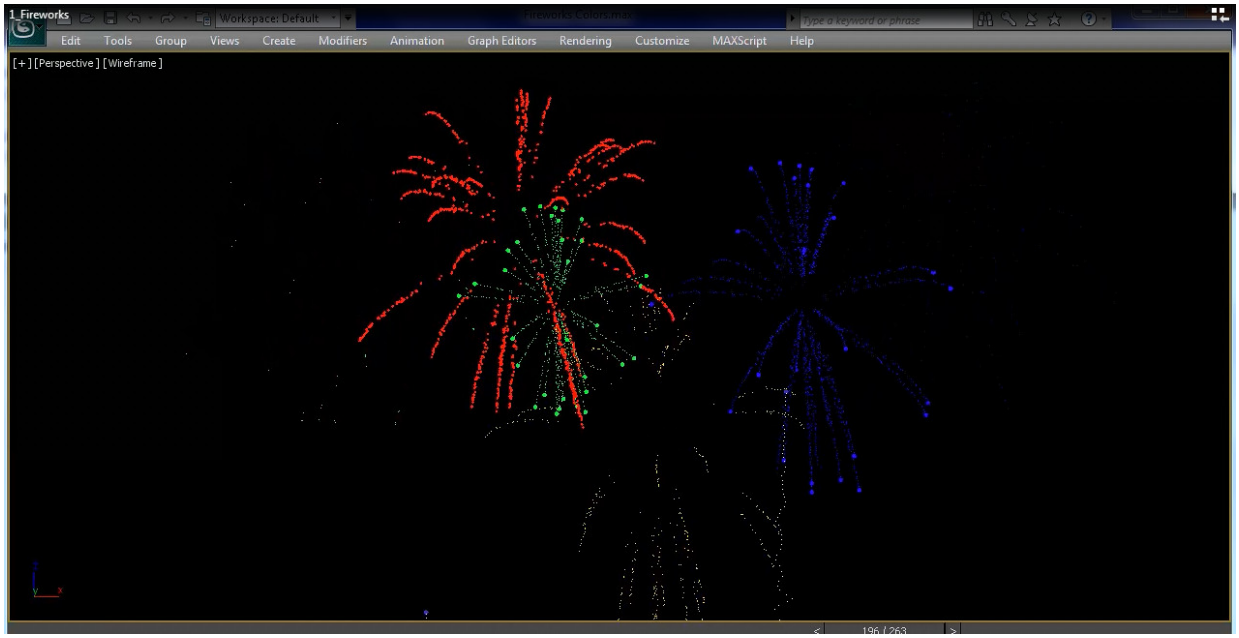


Objects can also be formed out of particles. In this demo the models are represented by particles instead of wireframe triangles, allowing interesting transformations to occur.

[http://mrdoob.github.com/three.js/examples/webgl_particles_shapes.html - firebolt]

Sets of particles can be used for all sorts of animations, such as this one of a firework emitting sparks as it goes. By overlapping particles you can get fuzzy effects.

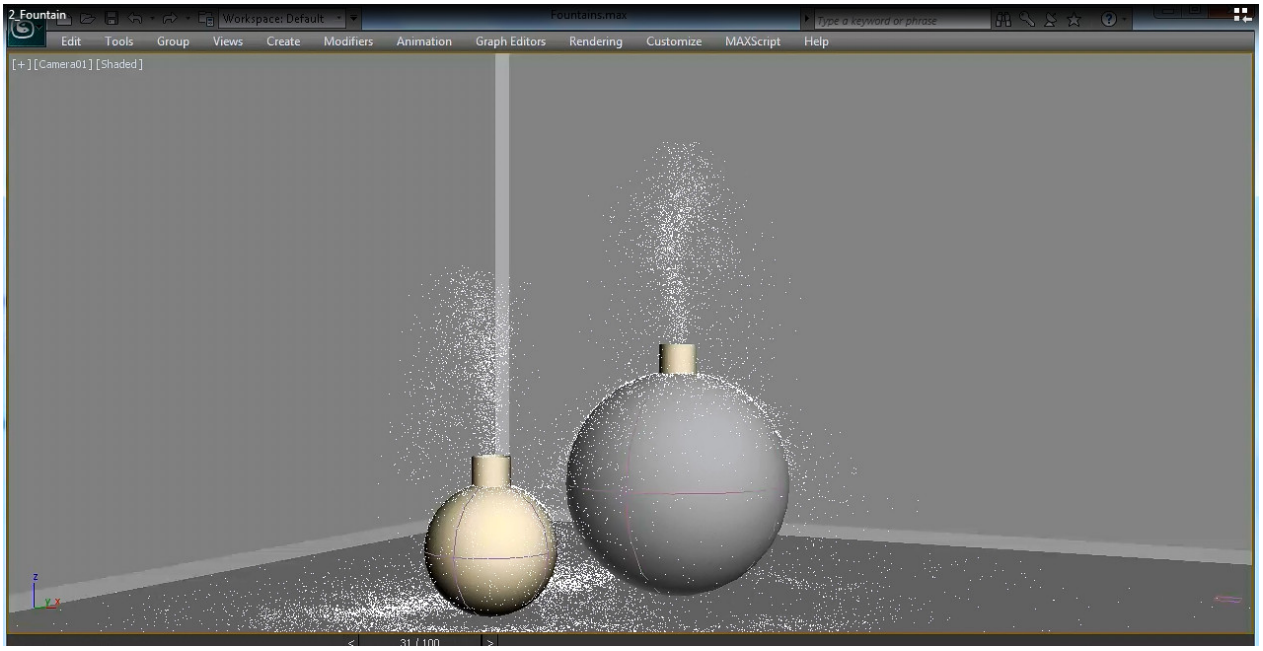
[video in Unit9_ParticleSystems, 1_Fireworks.mp4]



Of course, you can also make fireworks themselves. Here's how animators preview some

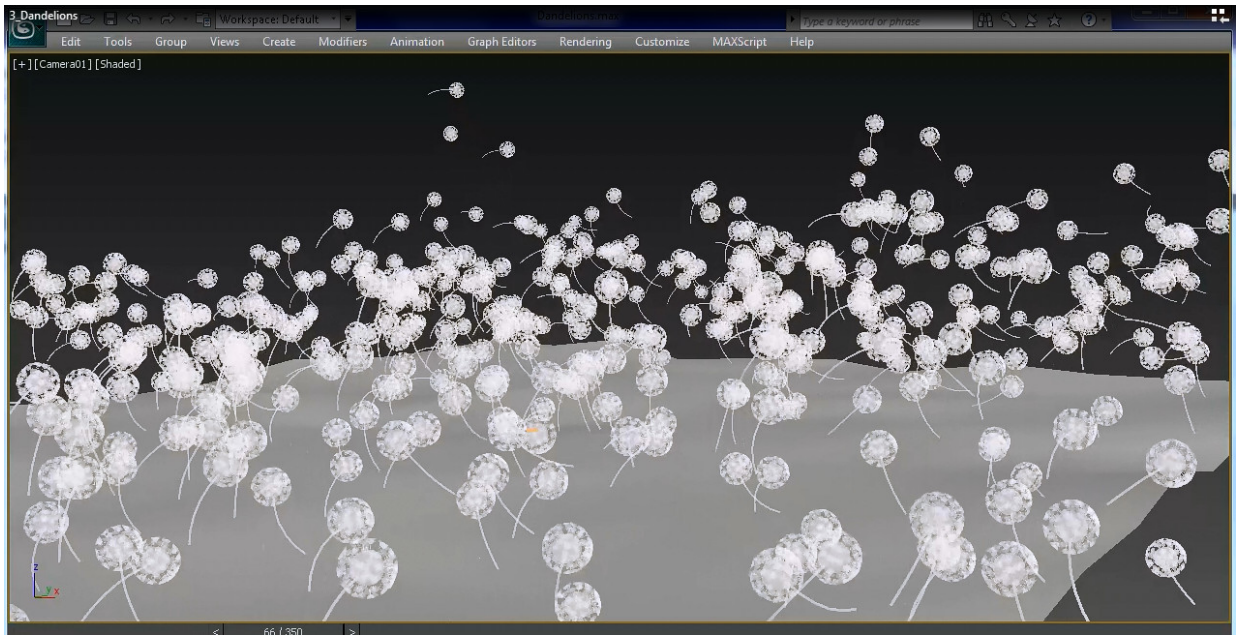
effects. Particle systems are often set up with guidelines, either fixed or with some randomness. Rules include elements such as when a particle is created, what path it travels, and whether it disappears after a certain duration. The path could be hand-animated or controlled by the laws of physics, or even a bit of both.

[video in Unit9_ParticleSystems, 2_Fountain.mp4]



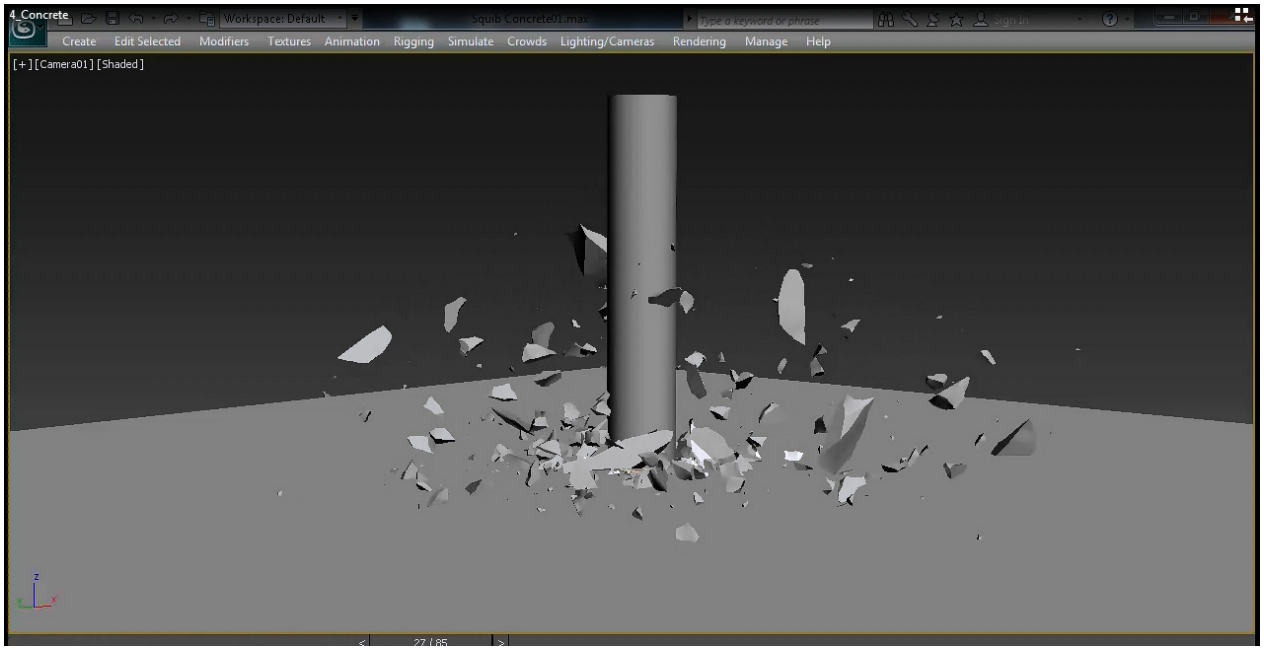
Another common use is to have large collections of particles simulate smoke, water, or other phenomena. Here particles are shown with a fountain effect.

[video in Unit9_ParticleSystems, 3_Dandelions.mp4]



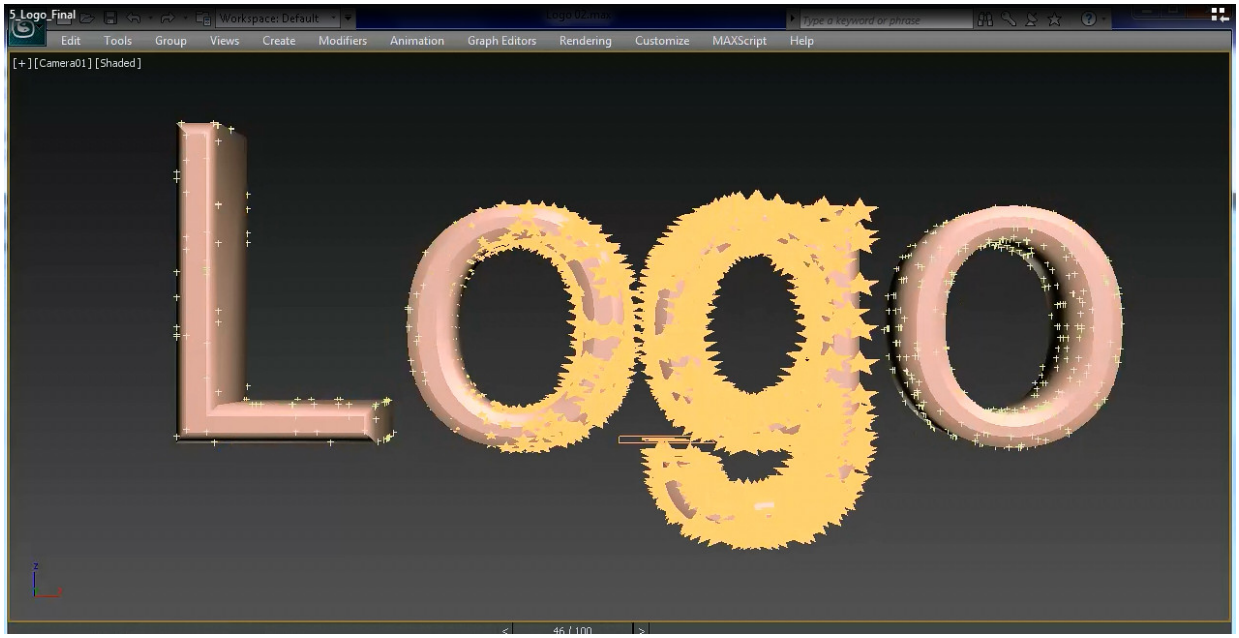
The particles don't have to be small, they can be objects in their own right, such as these dandelion seeds being blown away by the wind.

[video in Unit9_ParticleSystems, 4_Concrete.mp4]



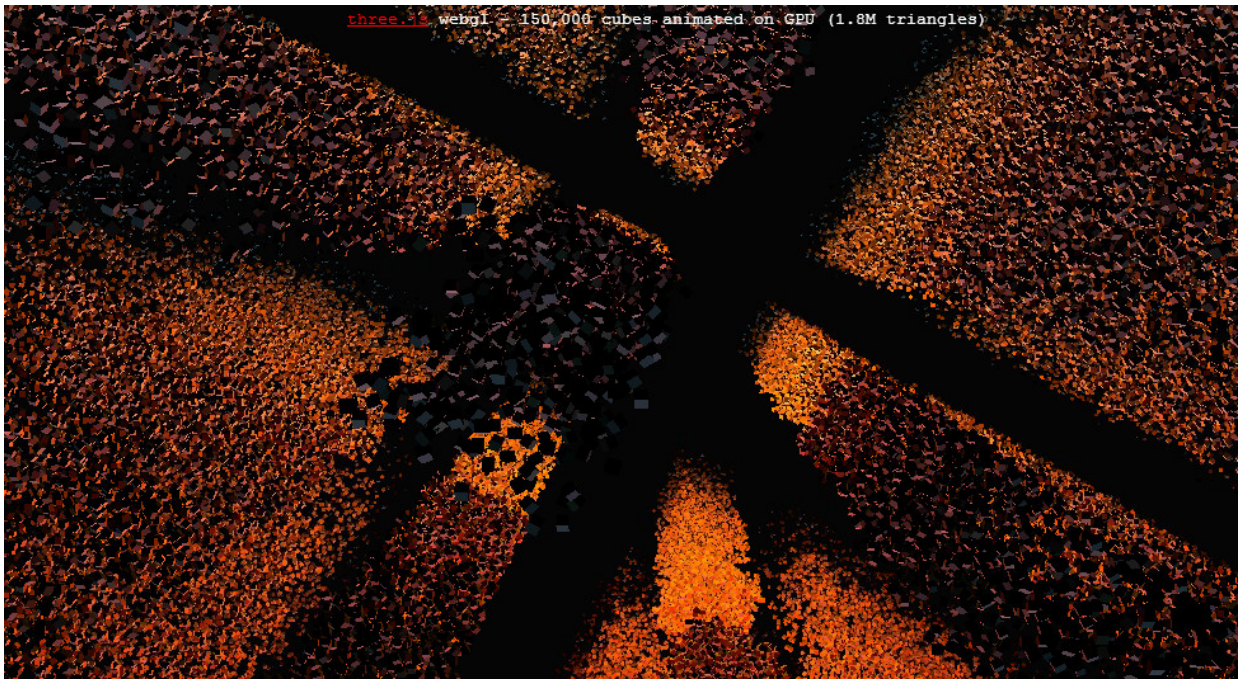
We can also create particles that are random 3D fragments which act as if they are generated from a collision.

[video in Unit9_ParticleSystems, 5_Logo_Final.mp4]



Even effects such as sparkling logos can be done by animating particles. Here's the preview, followed by the final off-line rendering.

[http://alteredqualia.com/three/examples/webgl_cubes.html - I have a video clip, since Mac is likely to be slow, It's in Unit9_ParticleSystems, but I should try as the clip itself seems slow.]



Or you can simply use procedural methods to create amazing scenes. It still astounds me how many particles you can animate and render at once. In this demo by Altered Qualia there are 150,000 rotating 3D cubes being thrown at the screen, and I'm getting a frame rate of 33 frames per second on my laptop.

The whole particle system evolution itself - creation, movement, and extinction - can now even be calculated entirely on newer GPUs. This allows more elaborate animations while also freeing up the CPU for other work.

[Additional Course Materials:
There is a tutorial [here](http://creativejs.com/tutorials/three-js-part-1-make-a-star-field/) about creating and animating particles.

For a code sample showing how to create and animate a simple particle system, see [Lee Stemkoski's tutorial](http://stemkoski.github.com/Three.js/Particles.html).

Demos used in this lesson:
[Varying sizes](http://mrdoob.github.com/three.js/examples/webgl_custom_attributes_particles2.html)
[Particle people](http://mrdoob.github.com/three.js/examples/webgl_particles_dynamic.html)
[Firebolt heart](http://mrdoob.github.com/three.js/examples/webgl_particles_shapes.html)
[150K particles](http://alteredqualia.com/three/examples/webgl_cubes.html)

Jerome Etienne has [a quick page of code](http://jeromeetienne.github.io/tquery/www/live/editor/#U/../../../plugins/particles/examples/smokepuff.html) showing an animated puff of smoke.

This [seminal paper on particles](https://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf) is quite readable and goes over the Genesis sequence from Star Trek 2.

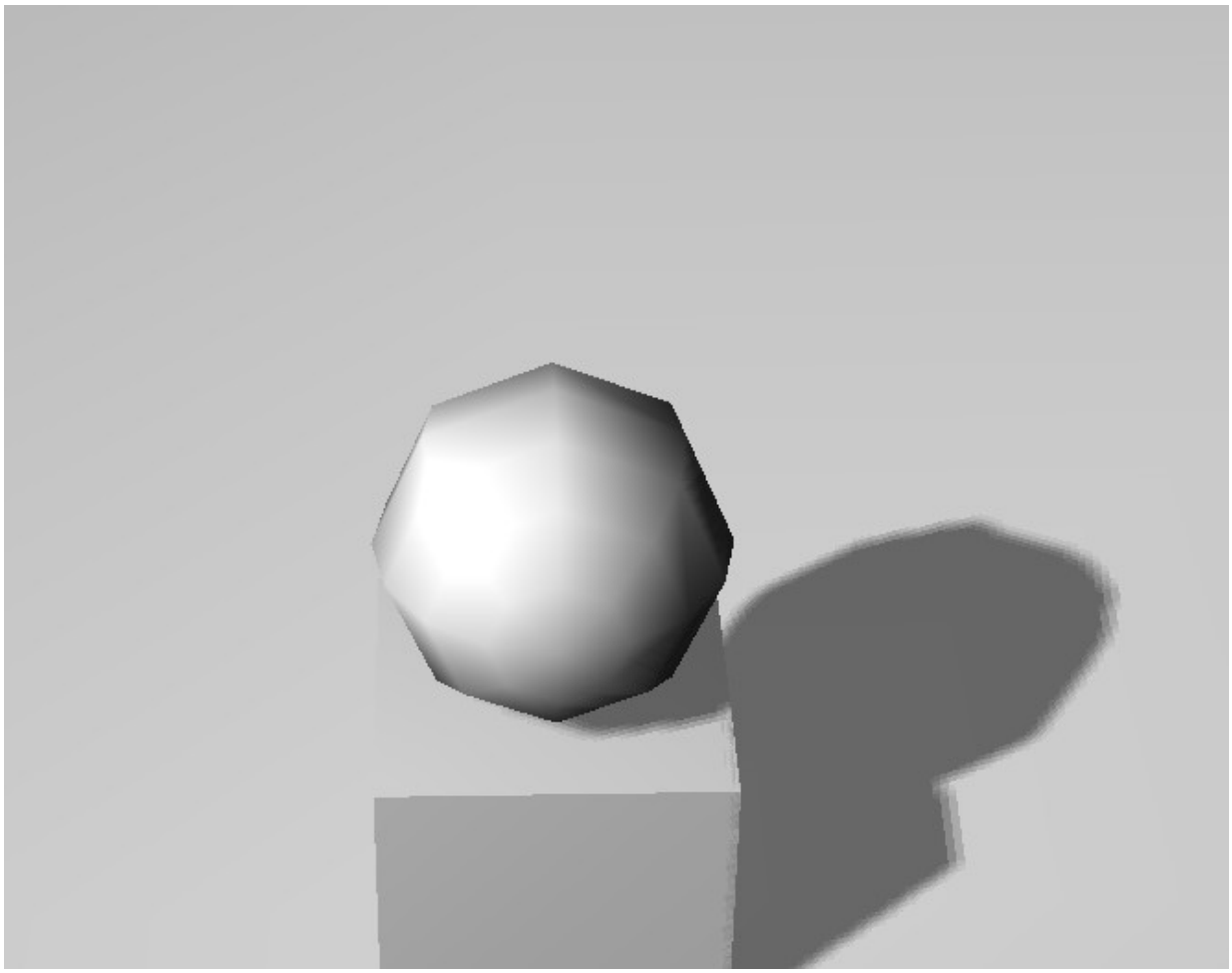
See the video [linked from this page](http://forum.unity3d.com/threads/166715-Museum-of-the-Microstar-A-DX11-contest-entry-by-RUST-LTD-and-co) for some modern, involved particle effects and much more.
]

[end recording 3/31 here]

Lesson: Collision Detection and Response

[recorded 4/1]

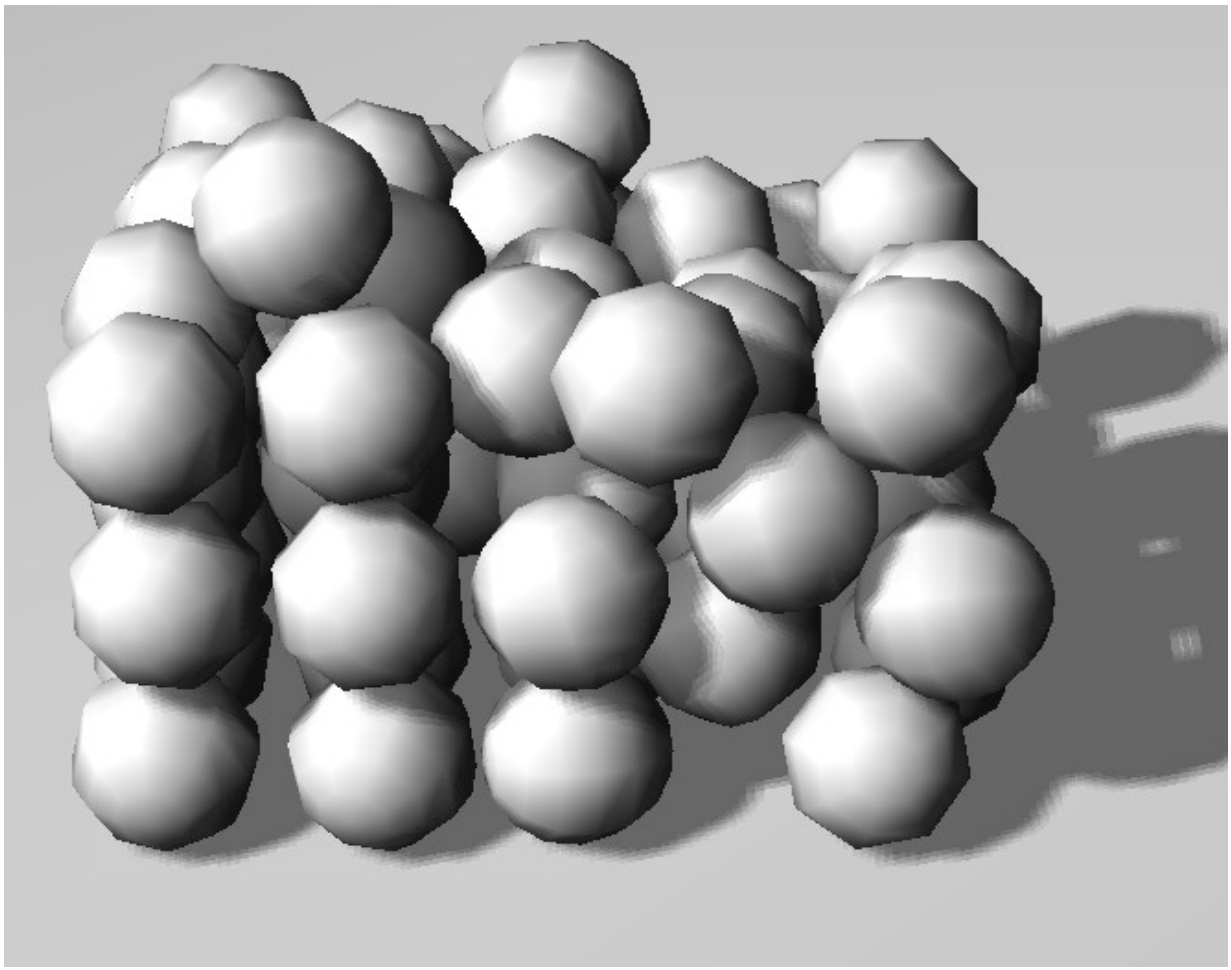
[http://schteppe.github.com/cannon.js/demos/motionstates.html]



One important element in making a virtual world seem real is collision detection and response. Collision detection is just that: you detect whether two objects have moved such that they overlap.

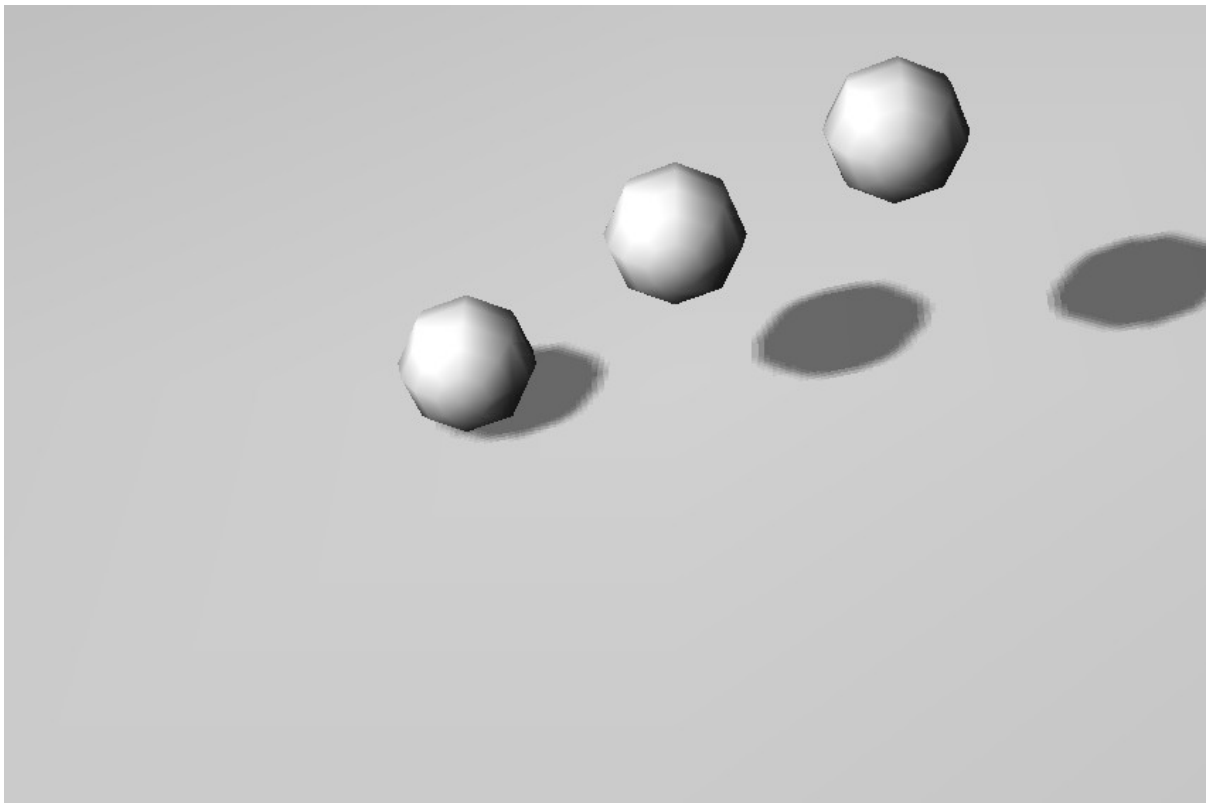
Even modest forms of collision detection can make a user feel they're in a virtual world instead of a ghost that can pass through anything. For example, simply shooting a ray from the user's location and finding the distance to the closest intersection will reveal whether the user's position should be adjusted.

[http://schteppe.github.com/cannon.js/demos/container.html]



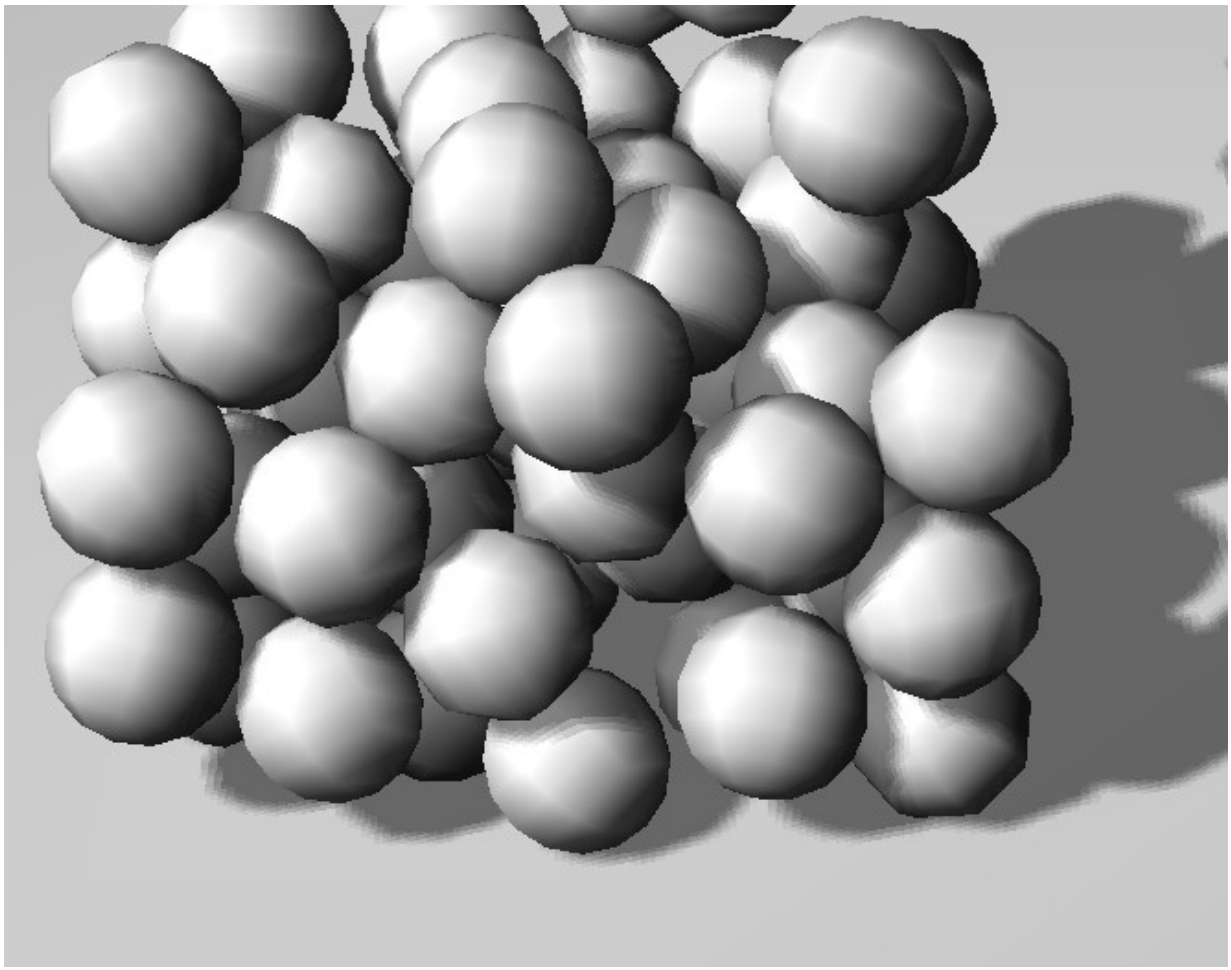
Where the problem begins to get complex is when you have many objects in a scene. With a brute force approach you compare everything to everything else for collision. Much of efficient collision detection coding is in creating data structures that quickly cull out possible collisions that can't occur in a frame.

[http://schteppe.github.com/cannon.js/demos/bounce.html]



The other half of the problem is collision response, what to do when two objects are found to have collided. This depends on a number of variables, including the velocity of the objects and, in this case, the elasticity of the materials.

[http://schteppe.github.com/cannon.js/demos/pile.html]



Collision detection is done primarily on the CPU in most programs at this point, though there have been some moves to off load the work to the GPU or even custom chips. This is a complex subject, entire books have been written about it, so this is all we'll cover here.

[Additional Course Materials:
Stefan Hedman has a fun little [collision detection and constraint solver library](http://schteppe.github.com/cannon.js/) that works with three.js, and that were used in this lesson. There's also another nice physics library for three.js, [Physijs](http://chandlerprall.github.io/Physijs/), with great demos.

Lee Stemkoski has code for detecting the intersection of a block with other blocks http://stemkoski.github.com/Three.js/Collision-Detection.html. Use the arrow keys and WASD to move the block around.

There's a short tutorial on [using ray casting for basic character collision detection](http://webmaestro.fr/blog/basic-collisions-detection-with-three-js-raycaster/).

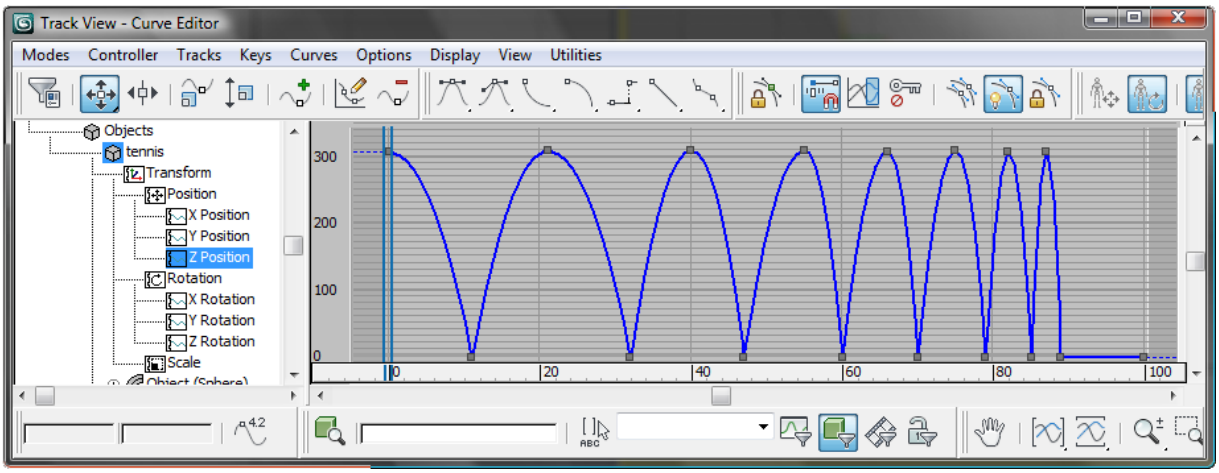
I maintain an [object/object intersection table](http://www.realtimerendering.com/intersections.html) with pointers to algorithms and books.
]

[removed possible exercise: Floor Collision]

[Force floor as limiter when viewing drinking bird. Could add a barrier around the bird, too.

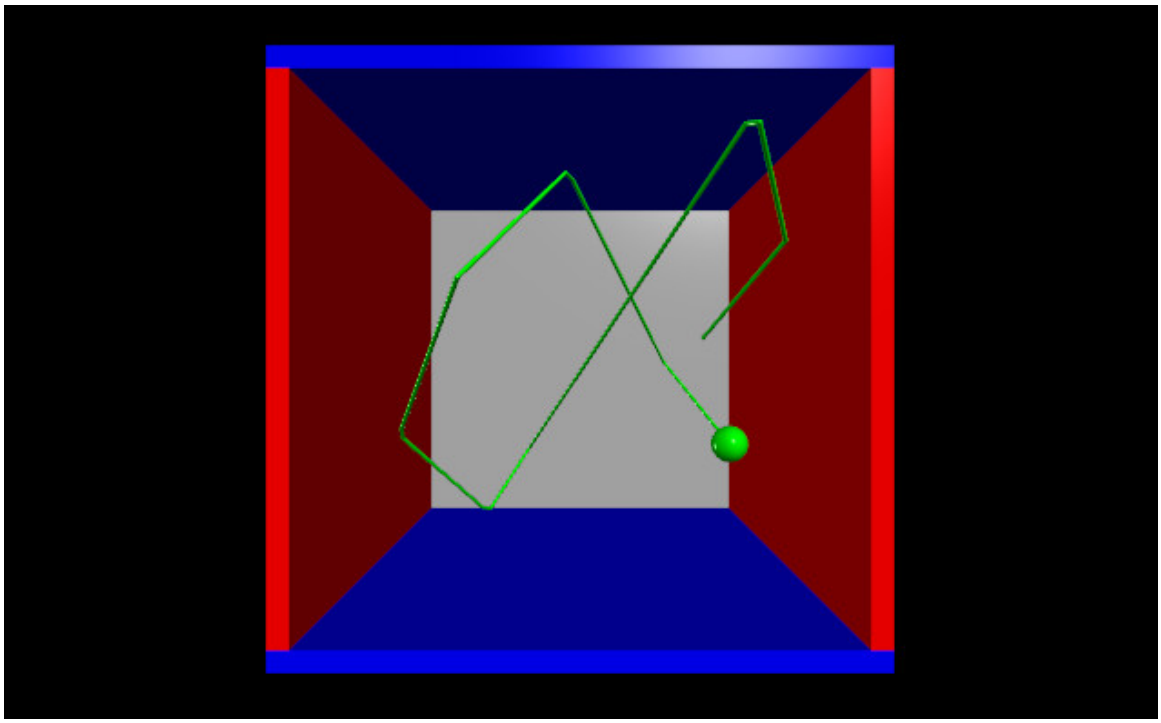
Interestingly, trying this out I was getting pretty unstable results at times. Depended on control.
]

Lesson: Simulation



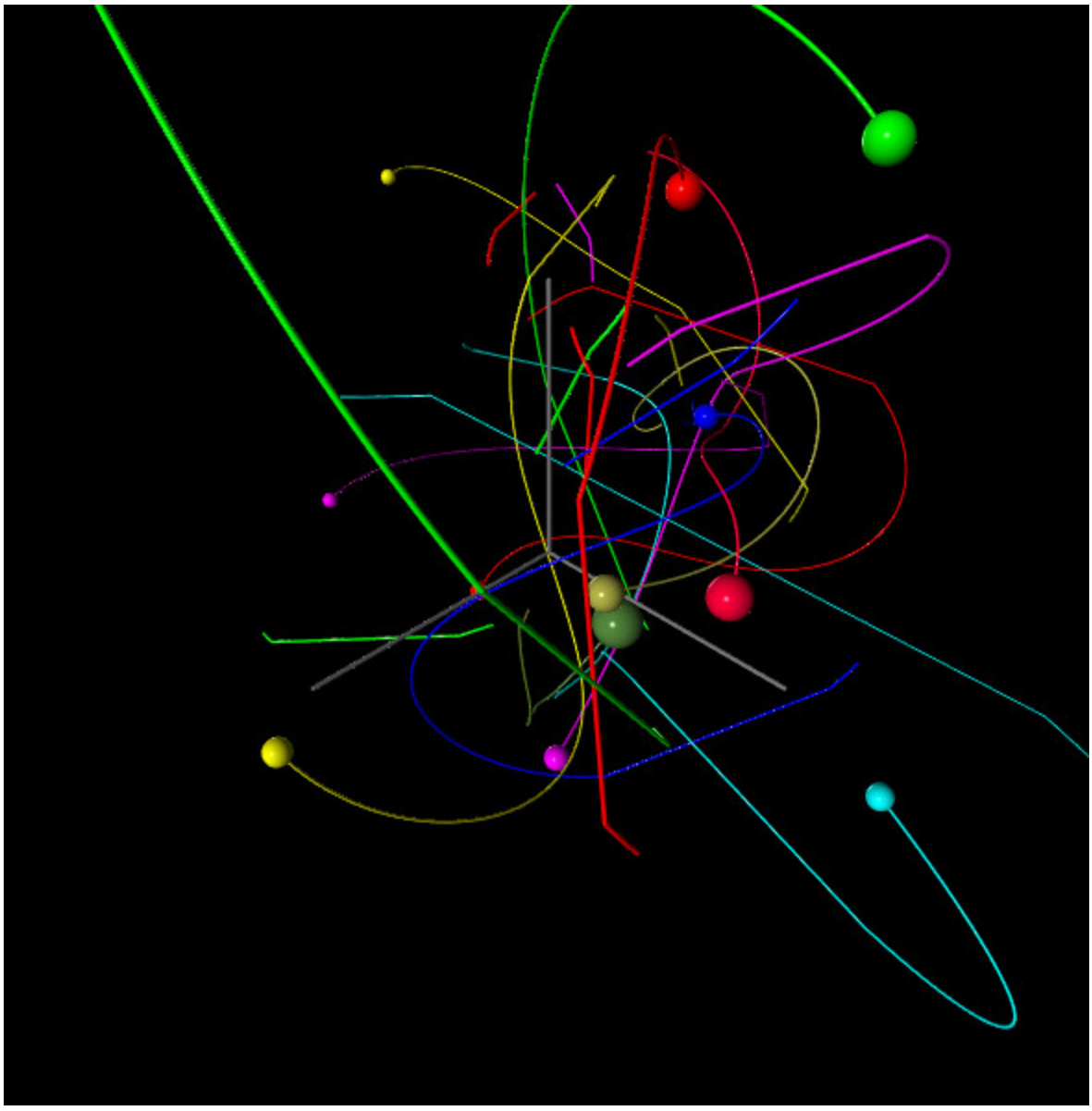
It's one thing to animate characters by hand. However, there's a huge variety of phenomena that are governed by the laws of physics. For simple tasks such as a bouncing ball, keyframing can suffice. Here's a keyframing timeline in 3DStudio Max, in the process of setting up an animation like this.

[http://www.glowscript.org/#/user/GlowScriptDemos/folder/Examples/program/Bounce - nice and simple, "try keyframing that!"]



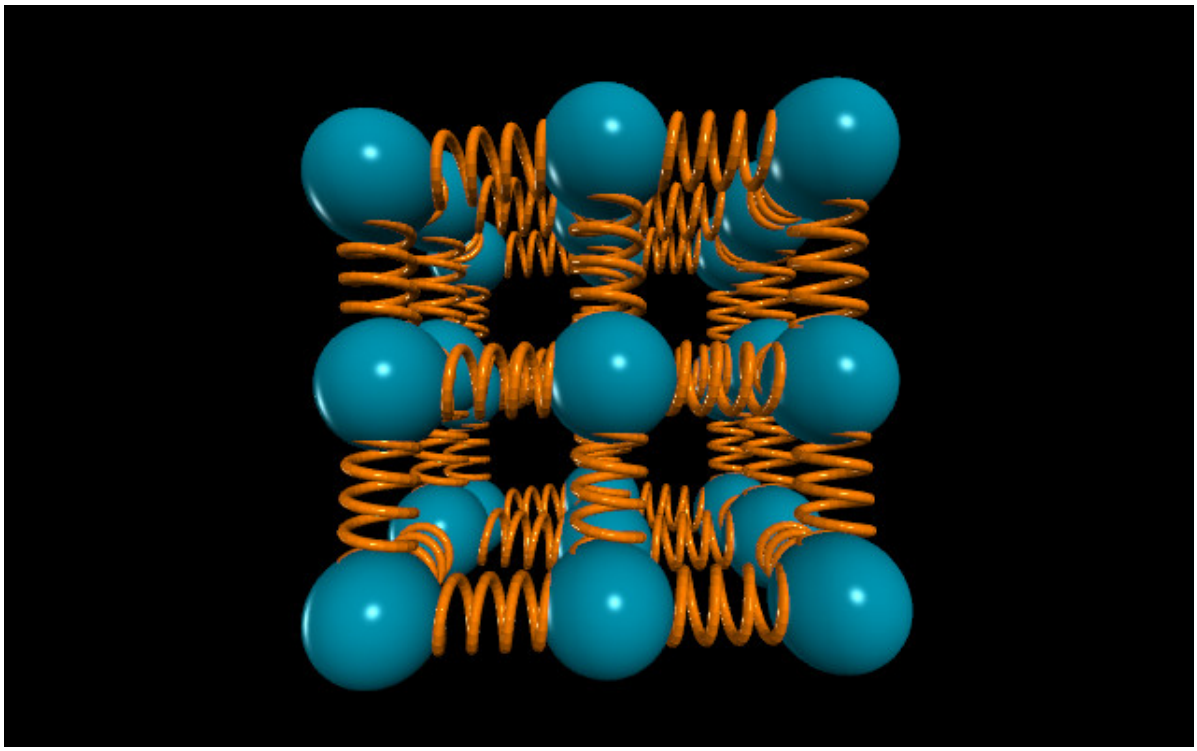
What if we wanted to have the bouncing ball keep bouncing? Keyframing by hand becomes a challenge, since there’s no simple repetition. For these phenomena we turn to simulation to generate the animations.

[http://www.glowscript.org/#/user/GlowScriptDemos/folder/Examples/program/Stars - good "here's a simulation" example.]



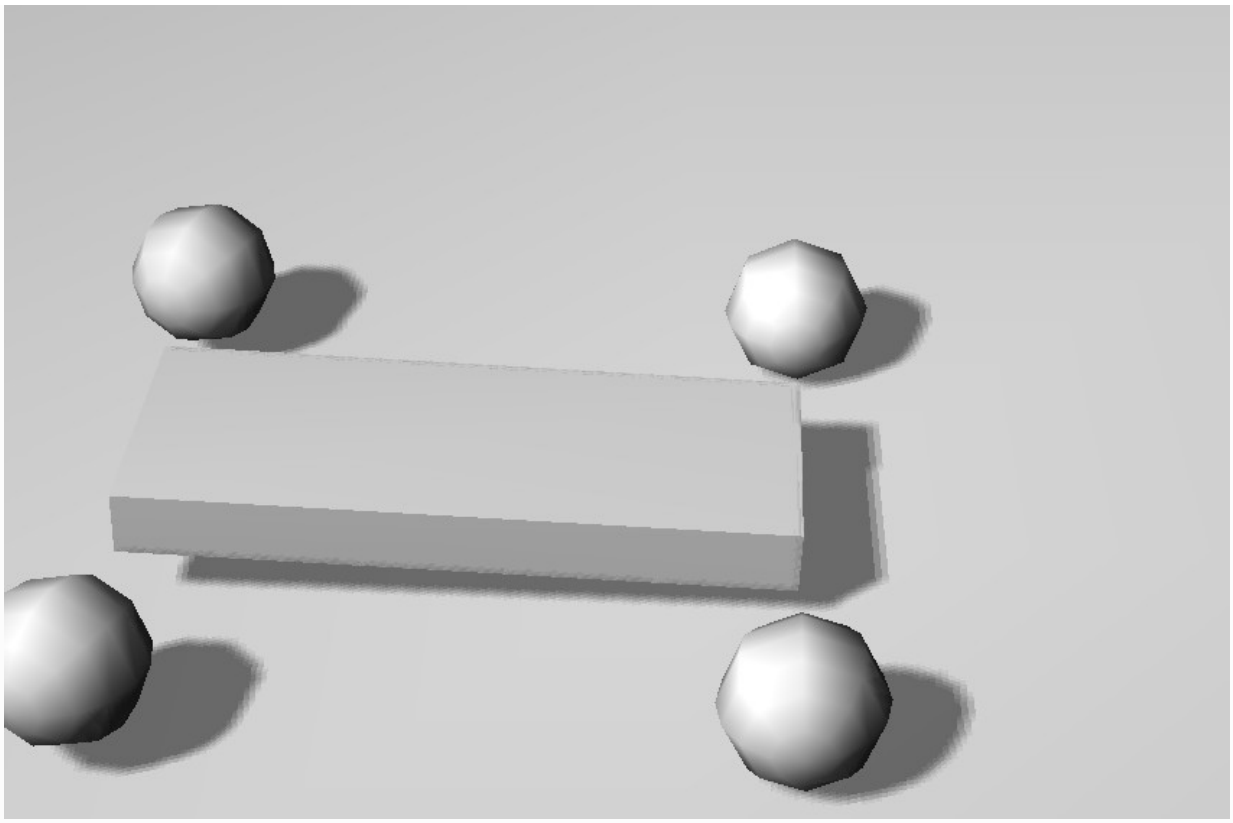
Here's another example, a set of bodies attracted to each other by gravitational forces. For the bouncing ball we could have perhaps written a little program that would create keyframes for us. Here the motions are complex and are calculated frame to frame on the fly.

[http://www.glowscript.org/#/user/GlowScriptDemos/folder/Examples/program/AtomicSolid - calculated on the fly]



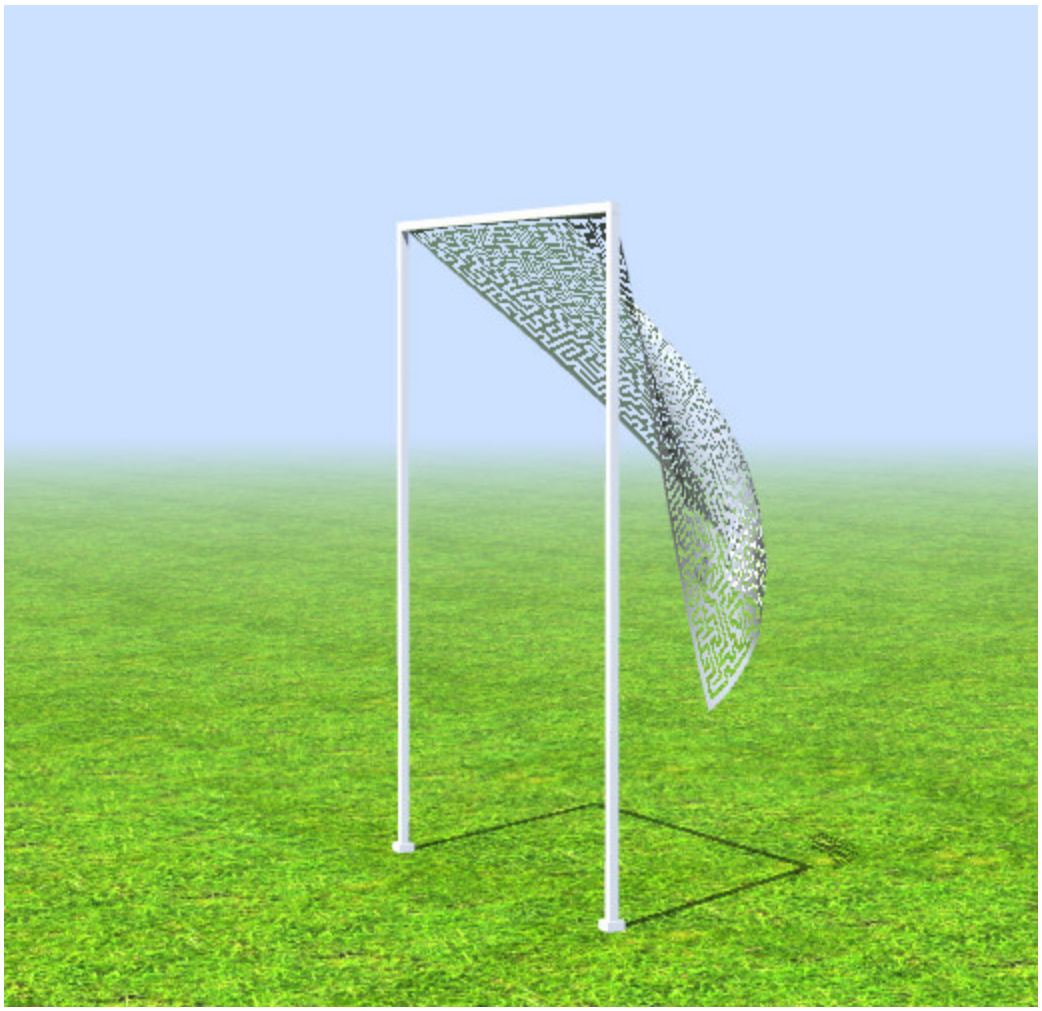
Many other phenomena can be simulated in real-time. Here's an example of a spring system. These three demos were all created in Glowscript, a system that is even higher level than three.js.

[http://schteppe.github.com/cannon.js/demos/rotational.html]



Simulating friction is an area that can be challenging, but critical if you want to get anywhere.

[http://mrdoob.github.com/three.js/examples/webgl_animation_cloth.html - wait for ground texture to load]



[Toggle wind on and off]

A blowing wind has drag and lift constraints, cloth has stretch constraints. This is a relatively simple system that can be solved in real time, as you can see.

Simulations can be affected by the user's actions. As a simple example, in this demo you can turn the wind on and off. These are not simply prestored, canned animations but are interactive.

[Additional Course Materials:
The physical simulations shown were done in [GlowScript](http://www.glowscript.org/).

The [wind and cloth
demo](http://mrdoob.github.com/three.js/examples/webgl_animation_cloth.html) is a part of three.js.
]

Lesson: Simulation Examples

[HUGE number of videos by Jos Stam. VIDEO TEAM: honestly, pick what you like and run say two or three of them on the screen at the same time. Montage time!]

Go to the physics and engineering sections of a technical school's library and you'll see a huge amount of information on a wide array of phenomena. Beyond drag, lift, and stretch there are many other properties we could simulate: friction, applied force, flow, elasticity, tension, normal force, turbulence, spring force, heat transfer, gravity, thrust, electromagnetism, and so on.

Computing these is done by what's called a "dynamics solver". What you're seeing on the screen now are a number of different simulations done for all sorts of phenomena. A few of these are computed at interactive speeds, some are created off line; in either case the rendering is separate from the physical computations.

Typically the way solvers work is that a tiny time step is chosen for the simulation, often smaller than the time represented by the frame. The effect of forces in the scene are computed for this sliver of time and objects are modified. This process is done again and again, time slice by time slice. Small steps help avoid various types of errors.

The area of physical simulation is wide ranging, and users can have different goals. An engineer or scientist may be making a visualization of a particular process and so the simulator is aiming to be as realistic and true to the physical world as possible. If interactive speeds are not needed, then time slices can be exceedingly small and mostly dependent on how much compute time the user wants to spend.

The other major area that uses simulation is for generating what I call "plausible" animations. These animations are not required to be physically correct; rather, their goal is to look reasonable to a viewer. Just as we don't pick up on reflection maps not having perfectly correct reflections, there are plenty of errors that we don't notice or can rationalize away. Though using physical principles for the creation of effects normally gives the most realistic results, for interactive rendering we often have to make significant simplifications.

There are also simulators for phenomena that are not based on physics. For example, flocks of birds in flight, traffic in a city, and crowds of people walking inside a building are all systems that can be simulated with some basic rules. The artist can also guide the simulation as desired, perhaps having characters run towards a particular location and jostle for position.

[Additional Course Materials:
The videos in this lesson were made by Jos Stam and others working on the [Nucleus dynamics

solver](http://www.autodeskresearch.com/pdf/nucleus.pdf). You can read about the main idea behind Nucleus
[here](http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/stam_on_mayas_ncloth).
]

Lesson: Guided Simulation

[video at Unit9_ButWaitTheresMore, cloth_wood.mov - let it run, repeat it to fill the text.]

Being able to guide a simulation is an important element. Here's a final render of a piece of cloth. It's a combination of the artist determining where one point on the cloth is moved, then physical simulation being computed. What you're seeing is a final, batch rendering, but this type of simulation can be created and viewed in real-time.

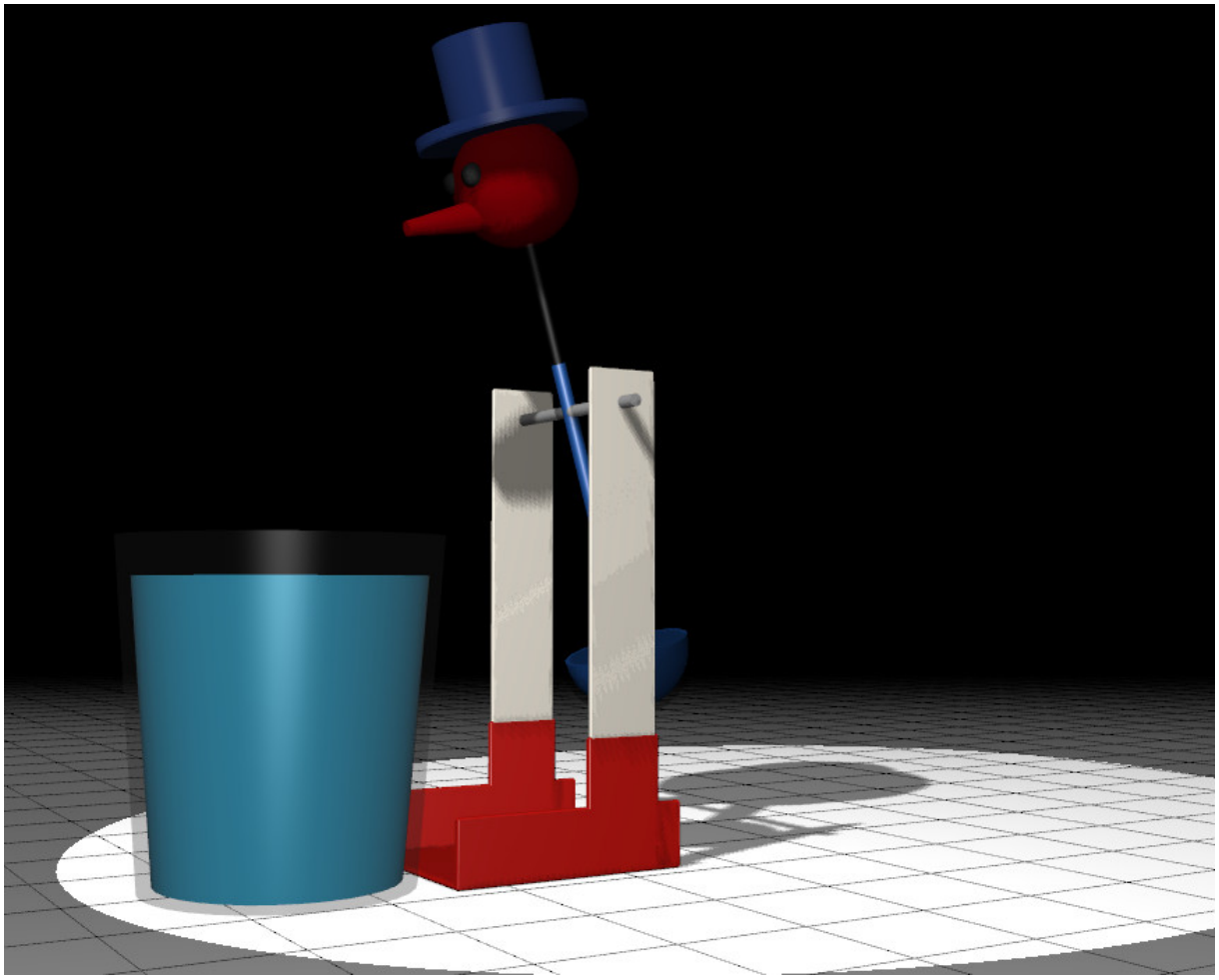
Simulations that are more complex can be run and the object locations stored for all frames. These stored sets of locations can be treated similarly to morph targets, with the animator being able to blend among two or more as desired. There are plenty of other combinations and possibilities of previewing and influencing the course of a simulation.

[Additional Course Materials:
The final render of the cloth simulation was by the 3DS MAX team under Neil Hazzard.

For cloth simulation, these [course notes](http://www.cs.cmu.edu/~baraff/sigcourse/) by Witkin and Baraff are worth your time.
]

Lesson: Drinking Bird Simulation

[unit10-db_tween_demo.js]



Let’s take one last look at our Drinking Bird. This version is just keyframes, and if you compare it to the real thing it’s not very true to life. One element we haven’t animated at all is the fluid level in the bird itself. The distribution of fluid matters to how the bird swings in the real world. Simulating this effect would be a bit of work, though once done properly would provide a true-to-life simulation of the physics of the situation. Perhaps you’ll be the one to come up with the equation for the Drinking Bird!

Question: What Kind of System?

What kind of system would you use for animating the following phenomena? Pick the most relevant algorithm. Some could use multiple techniques, so ***Use each algorithm only once.***

Skeletal Animation	***Morph Target***	***Motion Capture***	***Simulation***	
( )	***( )***	***( )***	***( )***	***A frowning troll***
( )	***( )***	***( )***	***( )***	***Avalanche!***
( )	***( )***	***( )***	***( )***	***A gymnastic flip***
( )	***( )***	***( )***	***( )***	***Showing the right-handed rule with my hand***

Answer

.X..
...X
..X.
X...

Let's start with the clearest match. The avalanche is certainly most easily done with simulation. Tracking all the small pieces of snow is extremely difficult with the other techniques.

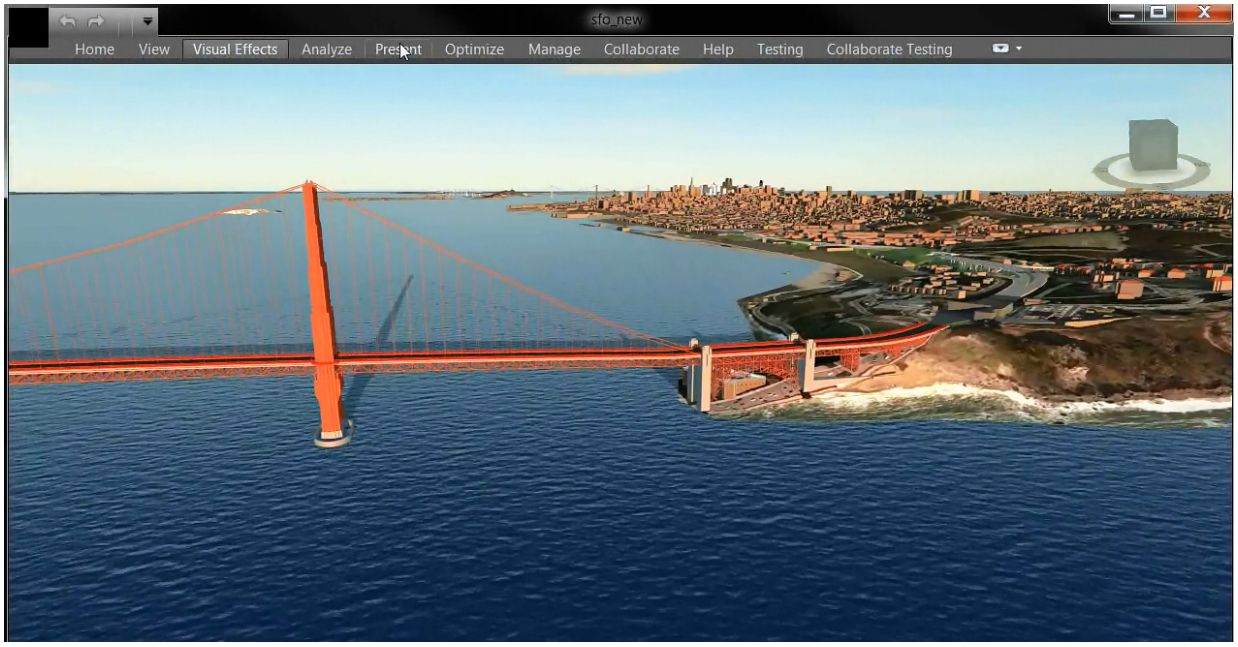
Of the remaining three, morph targets are best for making a troll frown, as just two keyposes would be needed.

That leaves skeletal animation and motion capture. Both tasks could be done with both. However, it's a fair bit of work to get the center of gravity correct for a gymnastic flip; motion capture would make this easy.

Motion capture might be able to capture a hand, but this can be tricky; skeletal animation can clearly handle this task.

Lesson: Level of Detail

[video: Unit9_LevelOfDetail, ScreenCapture_1-25-2013 1.14.54 PM.wmv - use 0:00 to 0:19 is best part, but can just let it run]



This lesson doesn't quite fit in this unit, but the topic is important enough that I'll cover it here.

As we move the camera or objects through a scene, models become larger and smaller. For very large scenes with many objects, rendering everything in the scene at full detail can make us lose any shred of interactivity.

We've seen how there are limits to how large a texture needs to be. In fact, it's usually best if the texel to pixel ratio stays about 1 to 1 - more than that and the texture can just become a source of noise.

Choosing the proper data representation is part of a much wider topic called "level of detail". In this area of computer graphics the focus is on how to change an object's representation as its size on the screen changes. Methods must balance tradeoffs between performance and quality.

------- page ------

[Level of Detail as title
Show texture mipchain and gesture.

]

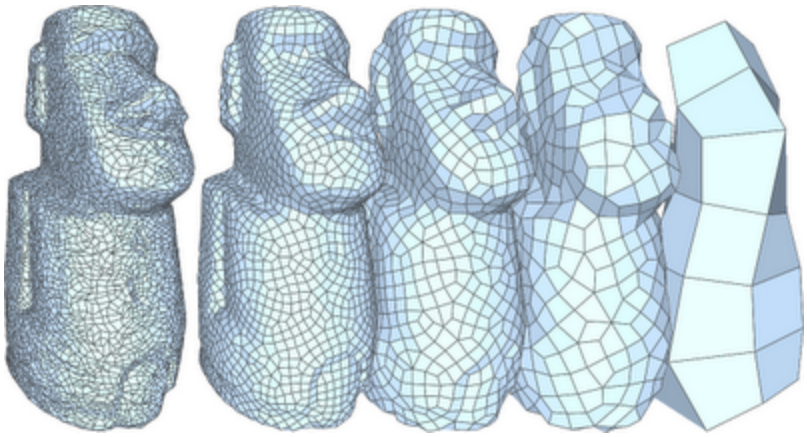
The two major areas where level of detail techniques come into play are with meshes and textures. Textures can be downsampled to create mipmaps, so that the proper texture level can be used for a scene. As a textured object gets farther away, a smaller texture on the mipmap is needed. There are also limits to how much memory a GPU has available for texture storage. For very large data sets, such as a city or world map, it's impossible to save all the high-resolution textures in memory at the same time.

Since we know that when an object is small on the screen it will need only a lower resolution texture, we can design our system to load only the low-resolution textures out of the mipchain. As we zoom in on a particular object, we can load the higher and higher resolution textures. At the same time, we can be deleting the higher resolution textures for objects that are now receding from view or that have been clipped off the screen entirely.

[use head of ninja, showing with displacement, smaller with normals, tiny with none]

As an object gets smaller on the screen, we can also consider simplifying the illumination equation, in other words using a shader with fewer instructions. For example, if we get far enough away from an object, its displacement map may add very little detail. Displacement maps help in particular by giving a more realistic silhouette to an object; at a distance these fine details are less important. As an object gets smaller, switching from a displacement map to a normal map, then to no bump map at all, saves considerable amounts of memory.

[decimation image]



Mesh geometry itself can be thought of in the same terms. As a mesh gets smaller on the screen, fewer vertices are needed to represent its details. For objects that are tessellated into triangles, this means a smaller tessellation factor can be given.

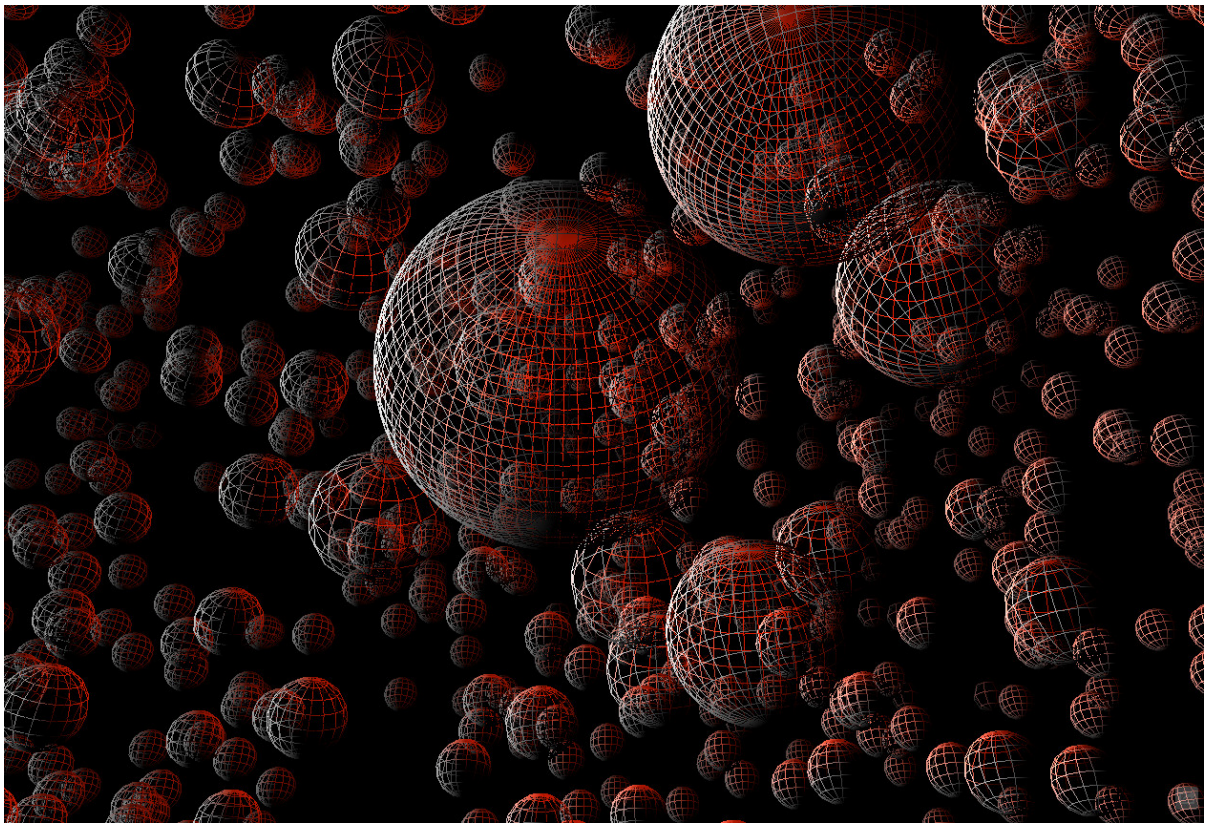
For more complex objects that do not have a simple tessellation factor, simplified meshes have

to either be created by hand, or by what is called a “decimation” algorithm. Such algorithms try to find the least important triangles and patch up their areas with fewer triangles. Here’s an example by Paolo Cignoni, using MeshLab, a powerful and free mesh manipulator.

These automatic techniques are effective, but can run into problems with objects such as faces, where the lips and eyes should be less subject to simplification because of their importance.

In mipmapping we interpolate between two textures in the mip pyramid. With level of detail techniques for geometry we can do the same with what are called geomorphs. This technique can provide a smoother transition between levels of detail, but at the cost of additional processing and memory, which is in opposition to much of the point of using level of detail techniques.

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



It’s much more common to simply replace a model with its simplified version when it becomes smaller on the screen. You’ll sometimes see a “pop” if the two versions of the model differ considerably.

Three.js supports this form of level of detail. In this demo you can see different versions of the sphere model being used, depending on distance from the viewer. These were explicitly chosen

by the programmer; if you look closely you'll see four different levels of detail.

[Additional Course Materials:
The demo used is [here](http://mrdoob.github.com/three.js/examples/webgl_lod.html).

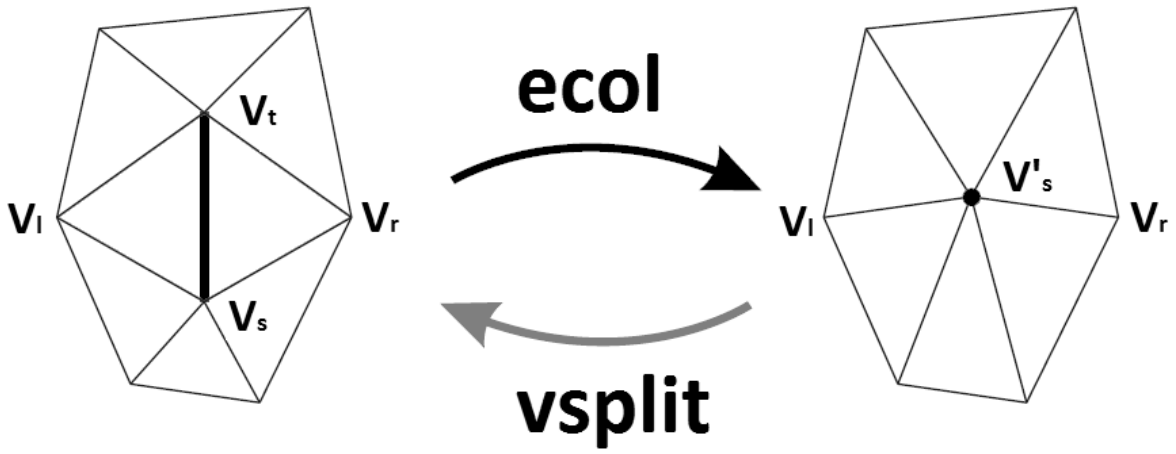
A performance tip: you can turn on an [FPS meter](http://beautifulpixels.blogspot.com/2013/03/chromes-fps-histogram.html) in Chrome to any app, from the debugger.

Texture levels and level of detail are discussed in detail in the books ["Level of Detail for 3D Graphics"](http://www.lodbook.com/) and ["3D Engine Design for Virtual Globes"](http://www.virtualglobebook.com/), among others. A free related resource is ["Virtual Texturing in Software and Hardware" SIGGRAPH course from 2012](http://www.jurajobert.com/data/Virtual_Texturing_in_Software_and_Hardware_course_notes.pdf).

[Quadric Error Metrics](http://illinois.edu/ds/www/garland/research/quadrics.html) is the most common algorithm today for mesh simplification. [MeshLab](http://www.meshlab.org/) is a free package that implements mesh simplification algorithms and much else.
]

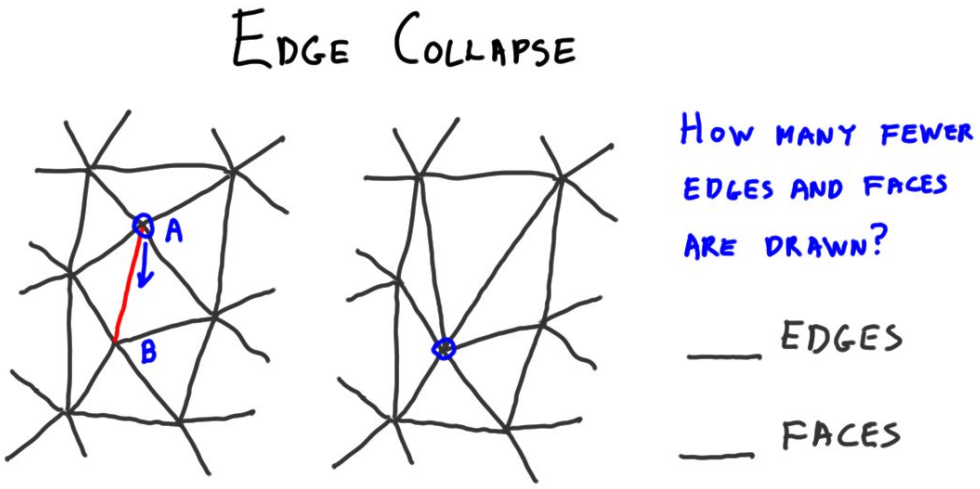
Question: Edge Collapse

[Show an edge collapse - not this; one vertex should stay in place. And make it look like it's part of a larger mesh!]



A common method of simplifying a mesh is the “edge-collapse”, where one vertex is moved along an edge to be at the same location as another vertex.

If I move vertex A to be at the same location at vertex B, I can then get rid of vertex A in memory and store just vertex B. This saves me from storing one vertex, but there are other benefits. If I’m drawing a wireframe, I’ll draw fewer edges. If I’m drawing triangles, I will need to save fewer faces.



The question to you is: ***how many fewer edges and faces are drawn?***

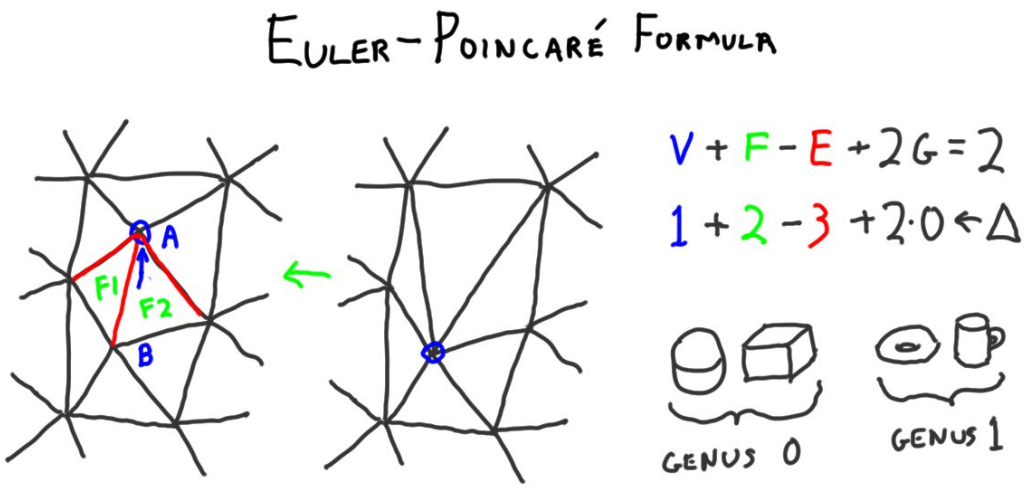
_____ edges and _____ faces

[Additional Course Materials:
Wikipedia's [edge collapse entry](http://en.wikipedia.org/wiki/Progressive_meshes#Edge_collapse) gives a bit more on this function.
]

Answer

The answer is three edges and two faces. The two triangles bordering the edge formed by A and B will disappear.

Lesson: Euler-Poincare formula



This problem brings up a useful little formula from topology, called the ***Euler-Poincare formula*** for connected planar graphs.

v + f − e + 2g = 2

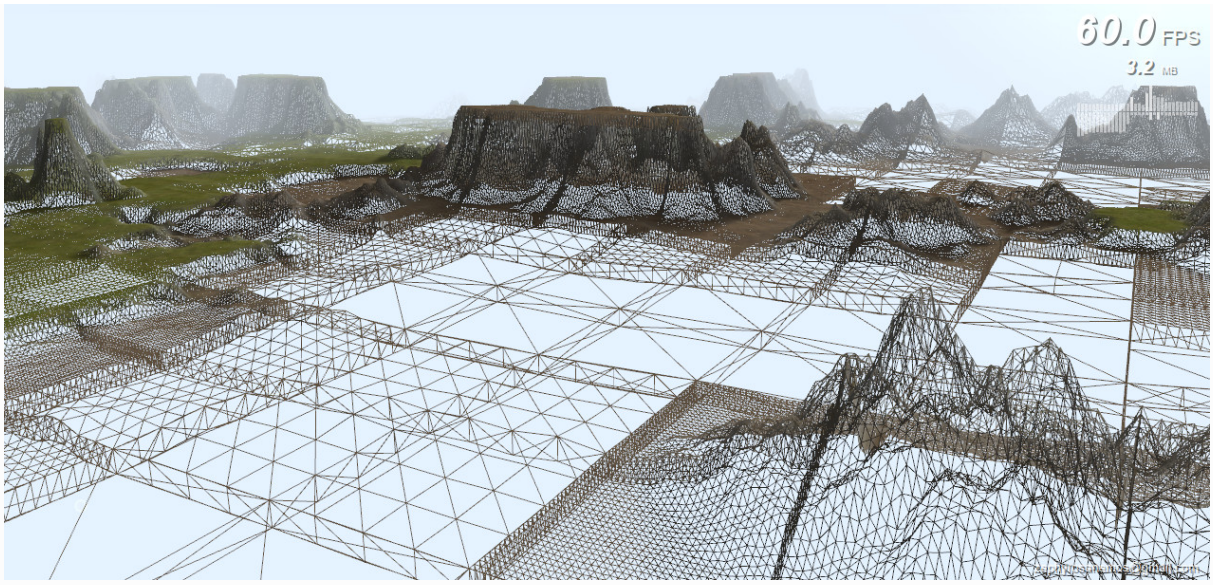
1 + 2
- 3

v is the number of vertices, f is the number of faces, e is the number of edges, and g is the genus. The genus is the number of holes in the object. As an example, a sphere and a cube have genus 0, a donut and a coffee cup with a handle have genus 1. There's an old joke about topologists not being able to tell their coffee cup from their donut.

For purposes of this formula a face includes the loop around the outer edge. We can use this formula as a check. We know we're subtracting one vertex, and it looks like two faces are going to disappear. The genus doesn't change, since we're not making a hole or removing one. So this equation says we've subtracted a vertex and two faces; to balance the equation three edges must also be removed.

Lesson: Terrain LOD System

[Free run of terrain demo: http://www.zephyrosanemos.com/]



[I have a video of this demo, Unit9_LevelOfDetail, Terrain_LOD.wmv]

Live demo, could do it on my laptop and with their mike: things to point out:

* Start with frustum culling: entire chunks thrown away

* Show how data is loaded on demand, do a quick move to see tile not loaded.
* The key thing is how much the terrain itself varies. This affects how the silhouette works. So flat parts of the terrain can stay low resolution even when close.

[Additional Course Materials:
[This is the terrain demo site](http://www.zephyrosanemos.com/), wonderfully instructive. The demo itself is [here](http://www.zephyrosanemos.com/windstorm/20130301/live-demo.html).

[This bit of video](http://www.youtube.com/watch?v=qx40CRwwkS8&t=15m43s) talks about the advantages of the efficient BufferGeometry class in three.js.

There is another form of culling called occlusion culling. Algorithms based on occlusion culling perform tests to rapidly determine if an object is fully hidden by other objects in the scene. Two articles which discuss GPU-based occlusion culling are available in [the free GPU Gems series of books](http://www.realtimerendering.com/#books). The articles are [here](http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html) and [here](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html). There are many other occlusion techniques, such as [this one for globes](http://blogs.agi.com/insight3d/index.php/2009/03/25/horizon-culling-2/).
]

[end, recorded 4/1]

Video: Interview, Biggest Challenge

[**video**, sd-02_question2.mp4 - biggest challenge for rendering]

Lesson: This is not the End

[headshot, recorded 4/2]

Ever since my first online Java program back in 1997 I was excited by the potential of the web to teach graphics. With WebGL and three.js I feel we've arrived. Interactive 3D graphics, with speed and quality equal to that seen in triple-A title games and high-end CAD packages, is finally possible. I'm also thrilled that we now can share programs and code in a straightforward way: no installs, no compilation, and anybody can look at the code being run.

[video: maybe have quote appear on the screen?]

Udacity's motto is "Learn. Think. Do." and I find that perfect for interactive rendering. You have all

the materials you require if you've taken this course - you don't need to buy a DNA sequencer or an array of radio telescopes.

Experimentation is easy and quick with three.js, and the only danger is the risk of staying up until 4 am, saying "what happens if I change this parameter?"

[video: maybe have quote appear on the screen?]
[hand gesture, form square, and then hold up just one hand with a "corner"]

I ran across a great quote from Confucius while working on this course: "Every truth has four corners: as a teacher I give you one corner, and it is for you to find the other three."

This fits my view of computer graphics: you can be given a foundation in the subject, but it's mostly up to you to pursue knowledge and learn by doing (Confucius probably meant something entirely different). Don't forget you have a giant reference library at your fingertips. Much of what I found for these lessons consisted of me typing phrases such as
[typing motion]
"three.js collision detection" into a search engine and seeing what popped up.

Developing any sort of interactive 3D graphics program was out of the reach of almost everyone 20 years ago, because back then graphics accelerators cost the price of a luxury car. Now they're the price of a large pizza. Even up until a few years ago, creating a 3D application involved compilers and downloads and installs and once you were done, it would usually work on only one operating system. If you ran someone else's glorious demo you were usually left scratching your head: how did they do it?

The fact is, you can now make a wonderful interactive 3D graphics program that anyone with a browser can access with just a click, and can read the code that created it with just a few more. How amazing is that?!

[with feeling]
There's so much you can do: figure out what makes others' code tick, make your own demos and applications, create your own libraries or contribute code to three.js itself (and, please, document it). [thumbs up and smile]

[hands out, then point to me]
Go off and do great things - and make sure to send me the link!

[Additional Course Materials:
Now what? Plenty. I track and host various resources at [our book's site](http://realtimerendering.com) - dig around. I've certainly [been blogging](http://www.realtimerendering.com/blog/) about WebGL and three.js.

It's worth visiting this [wonderful "Why WebGL?" page](http://acko.net/blog/on-webgl/) for its clever header and the demos. [This math visualization page](http://acko.net/blog/making-mathbox/) at the same site is lovely.

New demos? Visit webgl.com and learningwebgl.com. This second site also has [16 WebGL lessons](http://learningwebgl.com/blog/?page_id=1217).

Mr. Doob & AlteredQualia give [a worthwhile talk about three.js](http://www.youtube.com/watch?v=qx40CRwwkS8), past and future.

Here's [three.js's development over a year](http://www.youtube.com/watch?v=ZRLVQY45Vx4), in just two minutes. If you want to see your name shoot lasers at three.js, contribute!

Finally, I'm the guy in front of the camera and above the table, but there are many people who worked as hard or harder. First and foremost Gundega Dekena, as well as the video editing team and all Udacity in general, for being helpful and friendly throughout this arduous and exciting process. Mauricio Vives, Patrick Cozzi, and near the end Branislav Ulicny, aka Altered Qualia, for the truly massive amounts of help in technical reviewing all this material - it honestly wouldn't be half as good without them. Particular thanks to Ricardo Cabello, aka mr.doob, the creator and manager of three.js, for quick and helpful replies to my questions. Beyond those interviewed, I received huge amounts of help from Neil Hazzard, Qilin Ren, Yong Guo, Jos Stam, and dozens more inside Autodesk. Many thanks to all those who let me use their fantastic demos and images in these videos; every person I asked was more than happy to give me permission and answer questions, which is one thing that makes the graphics community great. Finally, thanks to Carl Bass for getting me involved in the first place, Brian Mathews for all the support, and my manager Jean-Luc Corenthin, the rest of my team, and especially my wife Cathy for their patience and understanding.
]