

# How to Create an Interactive 3D Character with Three.js

In this in-depth tutorial you will learn how to create an interactive 3D character that follows the mouse and performs random animations on click.

By Kyle Wetton in Tutorials on October 14, 2019



[demo](#)

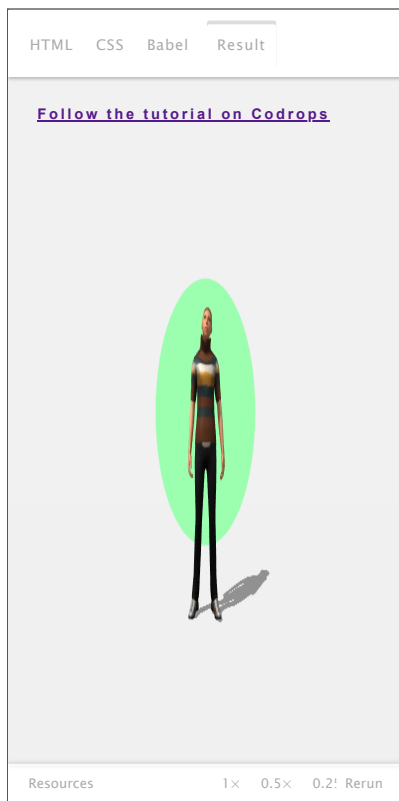
[download](#)

**From our sponsor:** Sell access to courses, classes, and community with Squarespace.

Ever had a personal website dedicated to your work and wondered if you should include a photo of yourself in there somewhere? I recently figured I'd go a couple steps further and added a fully interactive 3D version of myself that watched the user's cursor as they navigated around my screen. And ass if that wasn't enough, you could even click on me and I'd do stuff. This tutorial shows you how to do the same with a model we chose named Stacy.

Here's the demo (click on Stacy, and move your mouse around the Pen to watch her follow it).

We're going to use Three.js, and I'm going to assume you have a handle on JavaScript.



The [model](#) we use has ten animations loaded into it, at the bottom of this tutorial, I'll explain how its set up. This is done in Blender and the animations are from Adobe's free animation repo, [Mixamo](#).

## Part 1: HTML and CSS Project Starter

Let's get the small amount of HTML and CSS out of the way. This pen has everything you need. Follow along by forking this pen, or copy the HTML and CSS from here into a blank project elsewhere.



Our HTML consists of a loading animation (currently commented out until we need it), a wrapper div and our all-important canvas element. The canvas is what Three.js uses to render our scene, and the CSS sets this at 100% viewport size. We also load in two dependencies at the bottom of our

HTML file: Three.js, and GLTFLoader (GLTF is the format that our 3D model is imported as). Both of these dependencies are available as npm modules.

The CSS also consists of a small amount of centering styling and the rest is just the loading animation; really nothing more to it than that. You can now collapse your HTML and CSS panels, we will delve into that very little for the rest of the tutorial.

## Part 2: Building our Scene

In [my last tutorial](#), I found myself making you run up and down your file adding variables at the top that needed to be shared in a few different places. This time I'm going to give all of these to you upfront, and I'll let you know when we use them. I've included explanations of what each are if you're curious. So, our project starts like this. In your JavaScript add these variables. Note that because there is a bit at work here that would otherwise be in global scope, we're wrapping our entire project in a function:

```
(function() {  
  // Set our main variables  
  let scene,  
      renderer,  
      camera,  
      model,                // Our character  
      neck,                 // Reference to the neck bone in the skeleton  
      waist,                // Reference to the waist bone in the skeleton  
      possibleAnims,        // Animations found in our file  
      mixer,                 // THREE.js animations mixer  
      idle,                  // Idle, the default state our character returns to  
      clock = new THREE.Clock(), // Used for anims, which run to a clock instead of fr  
      currentlyAnimating = false, // Used to check whether characters neck is being us  
      raycaster = new THREE.Raycaster(), // Used to detect the click on our character  
      loaderAnim = document.getElementById('js-loader');  
  
})(); // Don't add anything below this line
```

We're going to set up Three.js. This consists of a scene, a renderer, a camera, lights, and an update function. The update function runs on every frame.

Let's do all this inside an `init()` function. Under our variables, and inside our function scope, we add our init function:

```
init();  
  
function init() {  
  
}
```

Inside our init function, let's reference our canvas element and set our background color, I've gone for a very light grey for this tutorial. Note that Three.js doesn't reference colors in a string like so "#f1f1f1", but rather a hexadecimal integer like 0xf1f1f1.

```
const canvas = document.querySelector('#c');
const backgroundColor = 0xf1f1f1;
```

Below that, let's create a new Scene. Here we set the background color, and we're also going to add some fog. This isn't that visible in this tutorial, but if your floor and background color are different, it can come in handy to blur those together.

```
// Init the scene
scene = new THREE.Scene();
scene.background = new THREE.Color(backgroundColor);
scene.fog = new THREE.Fog(backgroundColor, 60, 100);
```

Next up is the renderer, we create a new renderer and pass an object with the canvas reference and other options. The only option we're using here is that we're enabling antialiasing. We enable shadowMap so that our character can cast a shadow, and we set the pixel ratio to be that of the device, this is so that mobile devices render correctly. The canvas will display pixelated on high density screens otherwise. Finally, we add our renderer to our document body.

```
// Init the renderer
renderer = new THREE.WebGLRenderer({ canvas, antialias: true });
renderer.shadowMap.enabled = true;
renderer.setPixelRatio(window.devicePixelRatio);
document.body.appendChild(renderer.domElement);
```

That covers the first two things that Three.js needs. Next up is a camera. Let's create a new perspective camera. We're setting the field of view to 50, the size to that of the window, and the near and far clipping planes are the default. After that, we're positioning the camera to be 30 units back, and 3 units *down*. This will become more obvious later. All of this can be experimented with, but I recommend using these settings for now.

```
// Add a camera
camera = new THREE.PerspectiveCamera(
  50,
  window.innerWidth / window.innerHeight,
  0.1,
  1000
);
```

```
camera.position.z = 30
camera.position.x = 0;
camera.position.y = -3;
```

*Note that scene, renderer and camera are initially referenced at the top of our project.*

Without lights our camera has nothing to display. We're going to create two lights, a hemisphere light, and a directional light. We then add them to the scene using `scene.add(light)`.

Let's add our lights under the camera. I'll explain a bit more about what we're doing afterwards:

```
// Add Lights
let hemiLight = new THREE.HemisphereLight(0xffffffff, 0xffffffff, 0.61);
hemiLight.position.set(0, 50, 0);
// Add hemisphere light to scene
scene.add(hemiLight);

let d = 8.25;
let dirLight = new THREE.DirectionalLight(0xffffffff, 0.54);
dirLight.position.set(-8, 12, 8);
dirLight.castShadow = true;
dirLight.shadow.mapSize = new THREE.Vector2(1024, 1024);
dirLight.shadow.camera.near = 0.1;
dirLight.shadow.camera.far = 1500;
dirLight.shadow.camera.left = d * -1;
dirLight.shadow.camera.right = d;
dirLight.shadow.camera.top = d;
dirLight.shadow.camera.bottom = d * -1;
// Add directional light to scene
scene.add(dirLight);
```

The hemisphere light is just casting white light, and its intensity is at 0.61. We also set its position 50 units above our center point; feel free to experiment with this later.

Our directional light needs a position set; the one I've chosen feels right, so let's start with that. We enable the ability to cast a shadow, and set the shadow resolution. The rest of the shadows relate to the lights view of the world, this gets a bit vague to me, but its enough to know that the variable *d* can be adjusted until your shadows aren't clipping in strange places.

While we're here in our init function, lets add our floor:

```
// Floor
let floorGeometry = new THREE.PlaneGeometry(5000, 5000, 1, 1);
let floorMaterial = new THREE.MeshPhongMaterial({
  color: 0xeeeeee,
  shininess: 0,
});

let floor = new THREE.Mesh(floorGeometry, floorMaterial);
floor.rotation.x = -0.5 * Math.PI; // This is 90 degrees by the way
```

```
floor.receiveShadow = true;
floor.position.y = -11;
scene.add(floor);
```

What we're doing here is creating a new plane geometry, which is big: it's 5000 units (for no particular reason at all other than it really ensures our seamless background).

We then create a material for our scene. This is new. We only have a couple different materials in this tutorial, but it's enough to know for now that you combine geometry and materials into a mesh, and this mesh is a 3D object in our scene. The mesh we're making now is a really big, flat plane rotated to be flat on the ground (well, it is the ground). Its color is set to 0xeeeeee which is slightly darker than our background. Why? Because our lights shine on this floor, but our lights don't affect the background. This is a color I manually tweaked in to give us the seamless scene. Play around with it once we're done.

Our *floor* is a Mesh which combines the Geometry and Material. Read through what we just added, I think you'll find that everything is self explanatory. We're moving our floor down 11 units, this will make sense once we load in our character.

That's it for our `init()` function for now.

One crucial aspect that Three.js relies on is an update function, which runs every frame, and is similar to how game engines work if you've ever dabbled with Unity. This function needs to be placed *after* our `init()` function instead of inside it. Inside our update function the renderer renders the scene and camera, and the update is run again. Note that we immediately call the function after the function itself.

```
function update() {
  renderer.render(scene, camera);
  requestAnimationFrame(update);
}
update();
```

Our scene should now turn on. The canvas is rendering a light grey; what we're actually seeing here is both the background and the floor. You can test this out by changing the floors material color to 0xff0000. Remember to change it back though!

We're going to load the model in the next part. Before we do though, there is one more thing our scene needs. The canvas as an HTML element will resize just fine the way it is, the height and width is set to 100% in CSS. But, the scene needs to be aware of resizes too so that it can keep everything in proportion. Below where we call our update function (not inside it), add this function. Read it carefully if you'd like, but essentially what it's doing is constantly checking whether our renderer is the same size as our canvas, as soon as it's not, it returns *needResize* as a boolean.

```
function resizeRendererToDisplaySize(renderer) {
  const canvas = renderer.domElement;
  let width = window.innerWidth;
  let height = window.innerHeight;
  let canvasPixelWidth = canvas.width / window.devicePixelRatio;
  let canvasPixelHeight = canvas.height / window.devicePixelRatio;

  const needResize =
    canvasPixelWidth !== width || canvasPixelHeight !== height;
  if (needResize) {
    renderer.setSize(width, height, false);
  }
  return needResize;
}
```

We're going to use this inside our update function. Find these lines:

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

ABOVE these lines, we're going to check if we need a resize by calling our function, and updating the camera's aspect ratio to match the new size.

```
if (resizeRendererToDisplaySize(renderer)) {
  const canvas = renderer.domElement;
  camera.aspect = canvas.clientWidth / canvas.clientHeight;
  camera.updateProjectionMatrix();
}
```

Our full update function should now look like this:

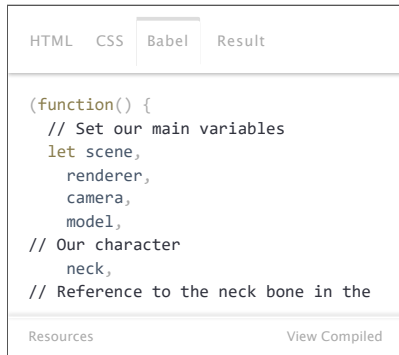
```
function update() {

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }
  renderer.render(scene, camera);
  requestAnimationFrame(update);
}

update();

function resizeRendererToDisplaySize(renderer) { ... }
```

Here's our project in its entirety so far. Next up we're going to load the model.



## Part 3: Adding the Model

Our scene is super sparse, but it's set up and we've got our resizing sorted, our lights and camera are working. Let's add the model.

Right at the top of our `init()` function, before we reference our canvas, let's reference the model file. This is in the GLTF format (.glb), Three.js support a range of 3D model formats, but this is the format it recommends. We're going to use our GLTFLoader dependency to load this model into our scene.

```
const MODEL_PATH = 'https://s3-us-west-2.amazonaws.com/s.cdn.io/1376484/stacy_lightweight.'
```



Still inside our `init()` function, below our camera setup, let's create a new loader:

```
var loader = new THREE.GLTFLoader();
```

This loader uses a method called `load`. It takes four arguments: the model path, a function to call once the model is loaded, a function to call during the loading, and a function to catch errors.

Lets add this now:

```
var loader = new THREE.GLTFLoader();  
  
loader.load(  
  MODEL_PATH,  
  function(gltf) {  
    // A lot is going to happen here  
  },  
  undefined, // We don't need this function  
  function(error) {  
    console.error(error);  
  }  
);
```



```
}  
);
```

Notice the comment “A lot is going to happen here”, this is the function that runs once our model is loaded. Everything going forward is added inside this function unless I mention otherwise.

The GLTF file itself (passed into the function as the variable *gltf*) has two parts to it, the scene inside the file (*gltf.scene*), and the animations (*gltf.animations*). Let's reference both of these at the top of this function, and then add the model to the scene:

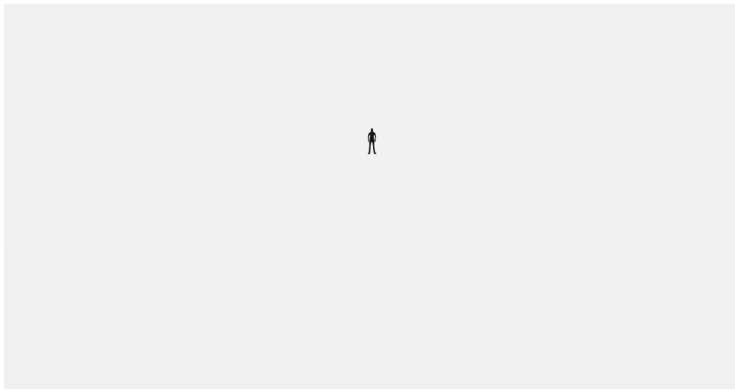
```
model = gltf.scene;  
let fileAnimations = gltf.animations;  
  
scene.add(model);
```

Our full `loader.load` function so far looks like this:

```
loader.load(  
  MODEL_PATH,  
  function(gltf) {  
    // A lot is going to happen here  
    model = gltf.scene;  
    let fileAnimations = gltf.animations;  
  
    scene.add(model);  
  
  },  
  undefined, // We don't need this function  
  function(error) {  
    console.error(error);  
  }  
);
```

*Note that model is already initialized at the top of our project.*

You should now see a small figure in our scene.



A couple of things here:

- Our model is really small; 3D models are like vectors, you can scale them without any loss of definition; Mixamo outputs the model really small, and for that reason we will need to scale it up.
- You can include textures inside a GLTF model, there are a number of reasons why I didn't, the first is that decoupling them allows for smaller file sizes when hosting the assets, the other is to do with color space and I cover that more in the section at the bottom of this tutorial which deals with how to set 3D models up.

We added our model prematurely, so *above* `scene.add(model)`, let's do a couple more things.

First of all, we're going to use the model's `traverse` method to find all the meshes, and enabled the ability to cast and receive shadows. This is done like this. Again, this should go above `scene.add(model)`:

```
model.traverse(o => {  
  if (o.isMesh) {  
    o.castShadow = true;  
    o.receiveShadow = true;  
  }  
});
```

Then, we're going to set the model's scale to a uniformed 7x its initial size. Add this below our `traverse` method:

```
// Set the models initial scale  
model.scale.set(7, 7, 7);
```

And finally, let's move the model down by 11 units so that it's standing on the floor.

```
model.position.y = -11;
```



Perfect, we've loaded in our model. Let's now load in the texture and apply it. This model came with the texture and the model has been mapped to this texture in Blender. This process is called UV mapping. Feel free to download the image itself to look at it, and learn more about UV mapping if you'd like to explore the idea of making your own character.

We referenced the loader earlier; let's create a new texture and material above this reference:

```
let stacy_txt = new THREE.TextureLoader().load('https://s3-us-west-2.amazonaws.com/s.cdn.:  
  
stacy_txt.flipY = false; // we flip the texture so that its the right way up  
  
const stacy_mtl = new THREE.MeshPhongMaterial({  
  map: stacy_txt,  
  color: 0xffffffff,  
  skinning: true  
});  
  
// We've Loaded this earlier  
var loader = new THREE.GLTFLoader()
```



Lets look at this for a second. Our texture can't just be a URL to an image, it needs to be loaded in as a new texture using TextureLoader. We set this to a variable called *stacy\_txt*.

We've used materials before. This was placed on our floor with the color 0xeeeeee, we're using a couple of new options here for our models material. Firstly, we're passing the *stacy\_txt* texture to the map property. Secondly we are turning skinning on, this is critical for animated models. We reference this material with *stacy\_mtl*.

Okay, so we've got our textured material, our files scene (*gltf.scene*) only has one object, so, in our traverse method, let's add one more line under the lines that enabled our object to cast and receive

shadows:

```
model.traverse(o => {  
  if (o.isMesh) {  
    o.castShadow = true;  
    o.receiveShadow = true;  
    o.material = stacy_mtl; // Add this line  
  }  
});
```



Just like that, our model has become the fully realized character, Stacy.

She's a little lifeless though. The next section will deal with animations, but now that you've handled geometry and materials, let's use what we've learned to make the scene a little more interesting. Scroll down to where you added your floor, I'll meet you there.

Below your floor, as the final lines of your init() function, let's add a circle accent. This is really a 3D sphere, quite big but far away, that uses a BasicMaterial. The materials we've used previously are called PhongMaterials which can be shiny, and also most importantly can receive and cast shadows. A BasicMaterial however, can not. So, add this sphere to your scene to create a flat circle that frames Stacy better.

```
let geometry = new THREE.SphereGeometry(8, 32, 32);  
let material = new THREE.MeshBasicMaterial({ color: 0x9bffaaf }); // 0xf2ce2e  
let sphere = new THREE.Mesh(geometry, material);  
sphere.position.z = -15;  
sphere.position.y = -2.5;  
sphere.position.x = -0.25;  
scene.add(sphere);
```

Change the color to whatever you want!

## Part 4: Animating Stacy

Before we get started, you may have noticed that Stacy takes a while to load. This can cause confusion because before she loads, all we see is a colored dot in the middle of the page. I mentioned that in our HTML we had a loader that was commented out. Head to the HTML and uncomment this markup.

```
<!-- The Loading element overlays everything else until the model is loaded, at which point  
<div class="loading" id="js-loader"><div class="loader"></div></div>
```



Then again in our loader function, once the model has been added into the scene with `scene.add(model)`, add this line below it. `loaderAnim` has already been referenced at the top of our project.

```
loaderAnim.remove();
```

All we're doing here is removing the loading animation overlay once Stacy has been added to the scene. Save and then refresh, you should see the loader until the page is ready to show Stacy. If the model is cached, the page might load too quickly to see it.

Anyway, onto animating!

We're still in our loader function, we're going to create a new `AnimationMixer`, an `AnimationMixer` is a player for animations on a particular object in the scene. Some of this might look foreign, and is potentially outside of the scope of this tutorial, but if you'd like to know more, check out the [Three.js docs page on the AnimationMixer](#). You won't need to know more than what we handle here to complete the tutorial.

Add this below the line that removes the loader, and pass in our model:

```
mixer = new THREE.AnimationMixer(model);
```

*Note that `mixer` is referenced at the top of our project.*

Below this line, we're going to create a new `AnimationClip`, we're looking inside our *fileAnimations* to find an animation called 'idle'. This name was set inside Blender.

```
let idleAnim = THREE.AnimationClip.findByName(fileAnimations, 'idle');
```

We then use a method in our mixer called `clipAction`, and pass in our `idleAnim`. We call this `clipAction` *idle*.

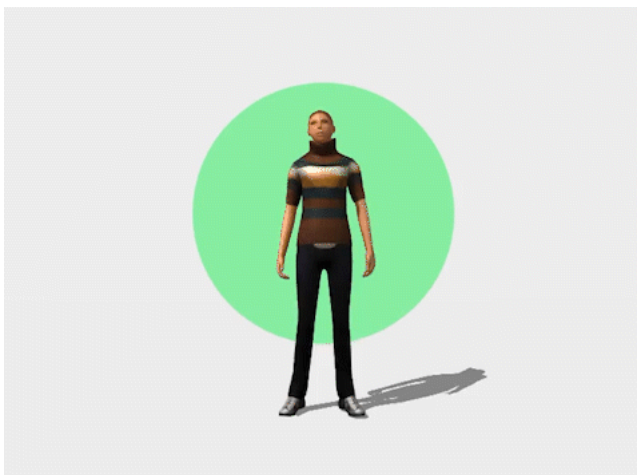
Finally, we tell *idle* to play:

```
idle = mixer.clipAction(idleAnim);
idle.play();
```

It's not going play yet though, we do need one more thing. The *mixer* needs to be updated in order for it to run continuously through an animation. In order to do this, we need to tell it to update inside our `update()` function. Add this right at the top, above our resizing check:

```
if (mixer) {
    mixer.update(clock.getDelta());
}
```

The update takes our clock (a `Clock` was referenced at the top of our project) and updates it to that clock. This is so that animations don't slow down if the frame rate slows down. If you run an animation to a frame rate, it's tied to the frames to determine how fast or slow it runs, that's not what you want.



Stacy should be happily swaying side by side! Great job! This is only one of 10 animations loaded inside our model file though, soon we will pick a random animation to play when you click on Stacy, but next up, let's make our model even more alive by having her head and body point toward our cursor.

## Part 5: Looking at our Cursor

If you don't know much about 3D (or even 2D animation in most cases), the way it works is that there is a skeleton (or an array of bones) that warp the mesh. These bones position, scale and rotation are animated across time to warp and move our mesh in interesting ways. We're going to hook into Stacy's skeleton (ek) and reference her neck bone and her bottom spine bone. We're then going to rotate these bones depending on where the cursor is relative to the middle of the screen. In order for us to do this though, we need to tell our current idle animation to ignore these two bones. Let's get started.

Remember that part in our model traverse method where we said `if (o.isMesh) { ... set shadows ...}`? In this traverse method (don't do this), you can also use `o.isBone`. I console logged all the bones and found the neck and spine bones, and their *names*. If you're making your own character, you'll want to do this to find the exact name string of your bone. Have a look here... (again don't add this to our project)

```
model.traverse(o => {  
  if (o.isBone) {  
    console.log(o.name);  
  }  
  if (o.isMesh) {  
    o.castShadow = true;  
    o.receiveShadow = true;  
    o.material = stacy_mtl;  
  }  
})
```

I got an output of a lot of bones, but the ones I was trying to find were these (this is pasted from my console):

```
...  
...  
mixamorigSpine  
...  
mixamorigNeck  
...  
...
```

So now we know our spine (from here on out referenced as the waist), and our neck names.

In our model traverse, let's add these bones to our neck and waist variables which have already been referenced at the top of our project.

```
model.traverse(o => {  
  if (o.isMesh) {  
    o.castShadow = true;  
    o.receiveShadow = true;  
    o.material = stacy_mtl;  
  }  
})
```

```

    }
    // Reference the neck and waist bones
    if (o.isBone && o.name === 'mixamorigNeck') {
        neck = o;
    }
    if (o.isBone && o.name === 'mixamorigSpine') {
        waist = o;
    }
  });
};

```

Now for a little bit more investigative work. We created an AnimationClip called *idleAnim* which we then sent to our mixer to play. We want to snip the neck and skeleton tracks out of this animation, or else our idle animation is going to overwrite any manipulation we try and create manually on our model.

So the first thing I did was console log *idleAnim*. It's an object, with a property called *tracks*. The value of *tracks* is an array of 156 values, every 3 values represent the animation of a single bone. The three being the position, quaternion (rotation) and the scale of a bone. So the first three values are the hips position, rotation and scale.

What I was looking for though was this (pasted from my console):

```

3: ad {name: "mixamorigSpine.position", ...
4: ke {name: "mixamorigSpine.quaternion", ...
5: ad {name: "mixamorigSpine.scale", ...

```

...and this:

```

12: ad {name: "mixamorigNeck.position", ...
13: ke {name: "mixamorigNeck.quaternion", ...
14: ad {name: "mixamorigNeck.scale", ...

```

So inside our animation, I want to splice the tracks array to remove 3,4,5 and 12,13,14.

However, once I splice 3,4,5 .... My neck becomes 9,10,11. Something to keep in mind.

Let's do this now. Below where we reference *idleAnim* inside our loader function, add these lines:

```

let idleAnim = THREE.AnimationClip.findByName(fileAnimations, 'idle');

// Add these:
idleAnim.tracks.splice(3, 3);
idleAnim.tracks.splice(9, 3);

```



We're going to do this to all animations later on. This means that regardless of what she's doing, you still have some control over her waist and neck, letting you modify animations in interesting ways in real time (yes, I did make my character play air guitar, and yes I did spend 3 hours making him head bang with my mouse while the animation ran).

Right at the bottom of our project, let's add an event listener, along with a function that returns our mouse position whenever it's moved.

```
document.addEventListener('mousemove', function(e) {
  var mousecoords = getMousePos(e);
});

function getMousePos(e) {
  return { x: e.clientX, y: e.clientY };
}
```

Below this, we're going to create a new function called `moveJoint`. I'll walk us through everything that these functions do.

```
function moveJoint(mouse, joint, degreeLimit) {
  let degrees = getMouseDegrees(mouse.x, mouse.y, degreeLimit);
  joint.rotation.y = THREE.Math.degToRad(degrees.x);
  joint.rotation.x = THREE.Math.degToRad(degrees.y);
}
```

The `moveJoint` function takes three arguments, the current mouse position, the joint we want to move, and the limit (in degrees) that the joint is allowed to rotate. This is called **degreeLimit**, remember this as I'll talk about it soon.

We have a variable called `degrees` referenced at the top, the `degrees` come from a function called `getMouseDegrees`, which returns an object of `{x, y}`. We then use these `degrees` to rotate the joint on the x axis and the y axis.

Before we add `getMouseDegrees`, I want to explain what it does.

`getMouseDegrees` does this: It checks the top half of the screen, the bottom half of the screen, the left half of the screen, and the right half of the screen. It determines where the mouse is on the screen in a percentage between the middle and each edge of the screen. For instance, if the mouse is half way between the middle of the screen and the right edge. The function determines that `right = 50%`, if the mouse is a quarter of the way UP from the center, the function determines that `up = 25%`. Once the function has these percentages, it returns the percentage of the **degreelimit**. So the function can determine your mouse is 75% right and 50% up, and return 75% of the degree limit on the x axis and 50% of the degree limit on the y axis. Same for left and right.

Here's a visual:



I wanted to explain that because the function looks pretty complicated, and I won't bore you with each line, but I have commented every step of the way for you to investigate it more if you want.

Add this function to the bottom of your project:

```
function getMouseDegrees(x, y, degreeLimit) {
  let dx = 0,
      dy = 0,
      xdiff,
      xPercentage,
      ydiff,
      yPercentage;

  let w = { x: window.innerWidth, y: window.innerHeight };

  // Left (Rotates neck left between 0 and -degreeLimit)

  // 1. If cursor is in the left half of screen
  if (x <= w.x / 2) {
    // 2. Get the difference between middle of screen and cursor position
    xdiff = w.x / 2 - x;
    // 3. Find the percentage of that difference (percentage toward edge of screen)
    xPercentage = (xdiff / (w.x / 2)) * 100;
    // 4. Convert that to a percentage of the maximum rotation we allow for the neck
    dx = ((degreeLimit * xPercentage) / 100) * -1; }
  // Right (Rotates neck right between 0 and degreeLimit)
  if (x >= w.x / 2) {
    xdiff = x - w.x / 2;
    xPercentage = (xdiff / (w.x / 2)) * 100;
    dx = (degreeLimit * xPercentage) / 100;
  }
  // Up (Rotates neck up between 0 and -degreeLimit)
  if (y <= w.y / 2) {
    ydiff = w.y / 2 - y;
    yPercentage = (ydiff / (w.y / 2)) * 100;
    // Note that I cut degreeLimit in half when she looks up
    dy = (((degreeLimit * 0.5) * yPercentage) / 100) * -1;
  }

  // Down (Rotates neck down between 0 and degreeLimit)
```

```

if (y >= w.y / 2) {
  ydiff = y - w.y / 2;
  yPercentage = (ydiff / (w.y / 2)) * 100;
  dy = (degreeLimit * yPercentage) / 100;
}
return { x: dx, y: dy };
}

```

Once we have that function, we can now use `moveJoint`. We're going to use it for the neck with a 50 degree limit, and for the waist with a 30 degree limit.

Update our `mousemove` event listener to include these `moveJoints`:

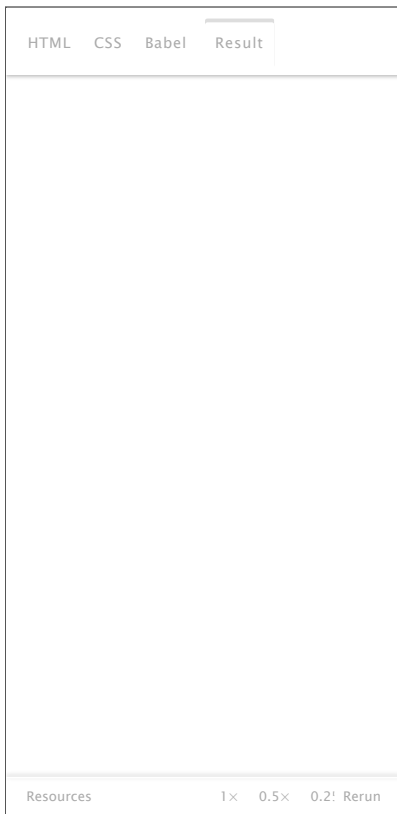
```

document.addEventListener('mousemove', function(e) {
  var mousecoords = getMousePos(e);
  if (neck && waist) {
    moveJoint(mousecoords, neck, 50);
    moveJoint(mousecoords, waist, 30);
  }
});

```

Just like that, move your mouse around the viewport and Stacy should watch your cursor wherever you go! Notice how idle animation is still running, but because we snipped the neck and spine bone (yuck), we're able to controls those independently.

This may not be the most scientifically accurate way of doing it, but it certainly looks convincing enough to create the effect we're after. Here's our progress so far, dig into this pen if you feel you've missed something or you're not getting the same effect.



## Part 6: Tapping into the rest of the animations

As I mentioned earlier, Stacy actually has 10 animations loaded into the file, and we've only used one of them. Let's head back to our loader function and find this line.

```
mixer = new THREE.AnimationMixer(model);
```

Below this line, we're going to get a list of AnimationClips that aren't idle (we don't want to randomly select idle as one of the options when we click on Stacy). We do that like so:

```
let clips = fileAnimations.filter(val => val.name !== 'idle');
```

Now below that, we're going to convert all of those clips into Three.js AnimationClips, the same way we did for idle. We're also going to splice the neck and spine bone out of the skeleton and add all of these AnimationClips into a variable called possibleAnims, which is already referenced at the top of our project.

```
possibleAnims = clips.map(val => {  
  let clip = THREE.AnimationClip.findByName(clips, val.name);  
  clip.tracks.splice(3, 3);
```

```

clip.tracks.splice(9, 3);
clip = mixer.clipAction(clip);
return clip;
}
);

```

We now have an array of clipActions we can play when we click Stacy. The trick here though is that we can't add a simple click event listener on Stacy, as she isn't part of our DOM. We are instead going to use raycasting, which essentially means shooting a laser beam in a direction and returning the objects that it hit. In this case we're shooting from our camera in the direction of our cursor.

Let's add this above our mousemove event listener:

```

// We will add raycasting here
document.addEventListener('mousemove', function(e) {...}

```

So paste this function in that spot, and I'll explain what it does:

```

window.addEventListener('click', e => raycast(e));
window.addEventListener('touchend', e => raycast(e, true));

function raycast(e, touch = false) {
  var mouse = {};
  if (touch) {
    mouse.x = 2 * (e.changedTouches[0].clientX / window.innerWidth) - 1;
    mouse.y = 1 - 2 * (e.changedTouches[0].clientY / window.innerHeight);
  } else {
    mouse.x = 2 * (e.clientX / window.innerWidth) - 1;
    mouse.y = 1 - 2 * (e.clientY / window.innerHeight);
  }
  // update the picking ray with the camera and mouse position
  raycaster.setFromCamera(mouse, camera);

  // calculate objects intersecting the picking ray
  var intersects = raycaster.intersectObjects(scene.children, true);

  if (intersects[0]) {
    var object = intersects[0].object;

    if (object.name === 'stacy') {

      if (!currentlyAnimating) {
        currentlyAnimating = true;
        playOnClick();
      }
    }
  }
}

```

We're adding two event listeners, one for desktop and one for touch screens. We pass the event to the `raycast()` function but for touch screens, we're setting the `touch` argument as `true`.

Inside the `raycast()` function, we have a variable called `mouse`. Here we set `mouse.x` and `mouse.y` to be `changedTouches[0]` position if `touch` is `true`, or just return the mouse position on desktop.

Next we call `setFromCamera` on `raycaster`, which has already been set up as a new `Raycaster` at the top of our project, ready to use. This line essentially raycasts from the camera to the mouse position. Remember we're doing this every time we click, so we're shooting lasers with a mouse at Stacy (brand new sentence?).

We then get an array of intersected objects; if there are any, we set the first object that was hit to be our object.

We check that the objects name is 'stacy', and we run a function called `playOnClick()` if the object is called 'stacy'. Note that we are also checking that a variable `currentlyAnimating` is `false` before we proceed. We toggle this variable on and off so that we can't run a new animation when one is currently running (other than idle). We will turn this back to `false` at the end of our animation. This variable is referenced at the top of our project.

Okay, so `playOnClick`. Below our rayasting function, add our `playOnClick` function.

```
// Get a random animation, and play it
function playOnClick() {
  let anim = Math.floor(Math.random() * possibleAnims.length) + 0;
  playModifierAnimation(idle, 0.25, possibleAnims[anim], 0.25);
}
```

This simply chooses a random number between 0 and the length of our `possibleAnims` array, then we call another function called `playModifierAnimation`. This function takes in `idle` (we're moving *from* idle), the speed to blend from idle to a new animation (`possibleAnims[anim]`), and the last argument is the speed to blend from our animation back to idle. Under our `playOnClick` function, lets add our `playModifierAnimation` and I'll explain what its doing.

```
function playModifierAnimation(from, fSpeed, to, tSpeed) {
  to.setLoop(THREE.LoopOnce);
  to.reset();
  to.play();
  from.crossFadeTo(to, fSpeed, true);
  setTimeout(function() {
    from.enabled = true;
    to.crossFadeTo(from, tSpeed, true);
    currentlyAnimating = false;
  }, to._clip.duration * 1000 - ((tSpeed + fSpeed) * 1000));
}
```

The first thing we do is reset the *to* animation, this is the animation that's about to play. We also set it to only play once, this is done because once the animation has completed its course (perhaps we played it earlier), it needs to be reset to play again. We then play it.

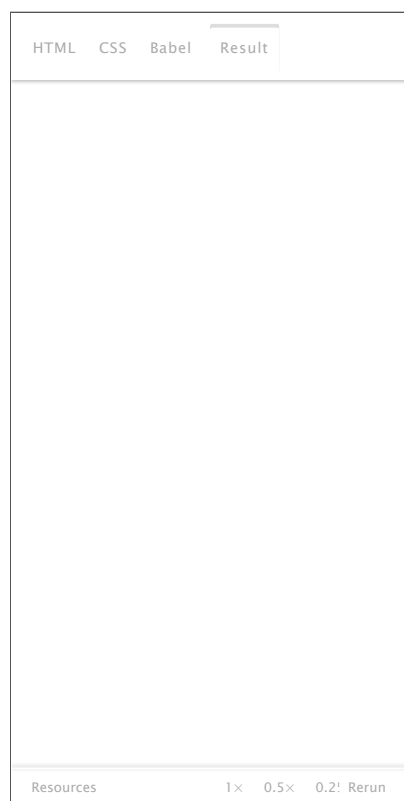
Each clipAction has a method called *crossFadeTo*, we use it to fade *from* (idle) *to* our new animation using our first speed (fSpeed, or *from speed*).

At this point our function has faded from idle to our new animation.

We then set a timeout function, we turn our from animation (idle) back to true, we cross fade back to idle, then we toggle currentlyAnimating back to false (allowing another click on Stacy). The time of the setTimeout is calculated by combining our animations length (\* 1000 as this is in seconds instead of milliseconds), and removing the speed it took to fade to and from that animation (also set in seconds, so \* 1000 again). This leaves us with a function that fades from idle, plays an animation and once it's completed, fades back to idle, allowing another click on Stacy.

Notice that our neck and spine bones aren't affected, giving us the ability to still control the way those rotate during the animation!

That concludes this tutorial, here's the completed project to reference if you got stuck.



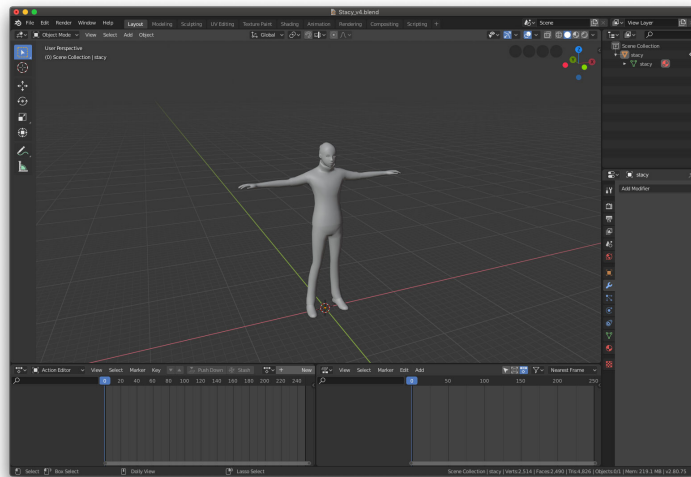
Before I leave you though, if you're interested in the workings of the model and animations itself, I'll cover some of the basics in the final part. I'll leave you to research some of the finer aspects, but this should give you plenty insight.

## Part 7: Creating the model file (optional)

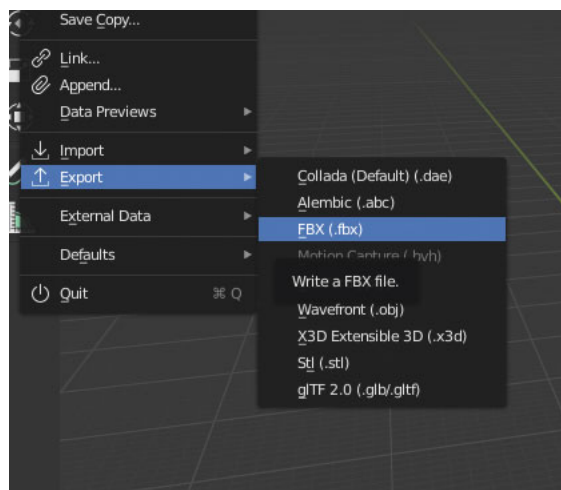
You'll require Blender for this part if you follow along. I recommend Blender 2.8, the latest stable build.

Before I get started, remember I mentioned that although you can include texture files inside your GLTF file (the format you export from Blender in), I had issues where Stacy's texture was really dark. It had to do with the fact that GLTF expects sRGB format, and although I tried to convert it in Photoshop, it still wasn't playing ball. You can't guarantee the type of file you're going to get as a texture, so the way I managed to fix this issue was instead export my file without textures, and let Three.js add it natively. I recommend doing it this way unless your project is super complicated.

Any way, here's what I started with in Blender, just a standard mesh of a character in a T pose. Your character most definitely should be in a T pose, because Mixamo is going to generate the skeleton for us, so it is expecting this.



You want to export your model in the FBX format.

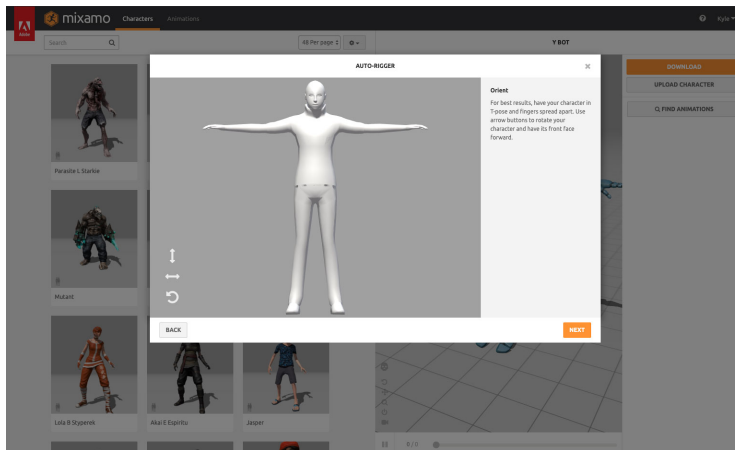


You aren't going to need the current Blender session any more, but more on that soon.

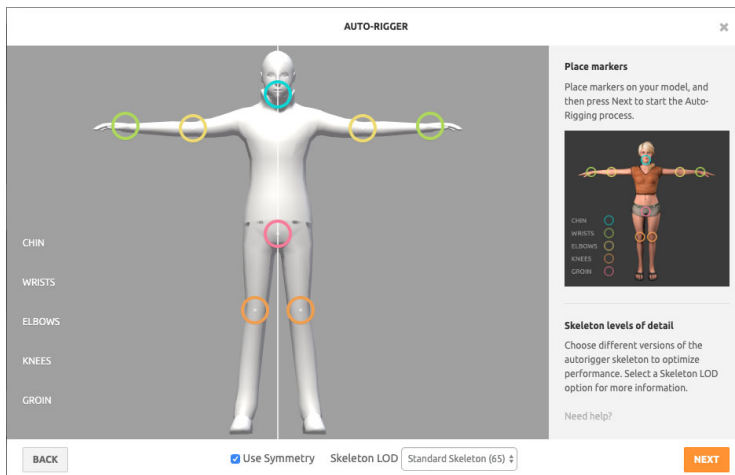


Head to [www.mixamo.com](https://www.mixamo.com), this site has a bunch of free animations that are used for all sorts of things, commonly browsed by Indie game developers, this Adobe service goes hand-in-hand with Adobe Fuse, which is essentially a character creator software. This is free to use, but you will need an Adobe account (by free I mean, you won't need a Creative Cloud subscription). So create one and sign in.

The first thing you want to do is upload your character. This is the FBX file that we exported from Blender. Mixamo will automatically bring up the Auto-Rigger feature once your upload is complete.



Follow the instructions to place the markers on the key areas of your model. Once the auto-rigging is complete, you'll see a panel with your character animating!

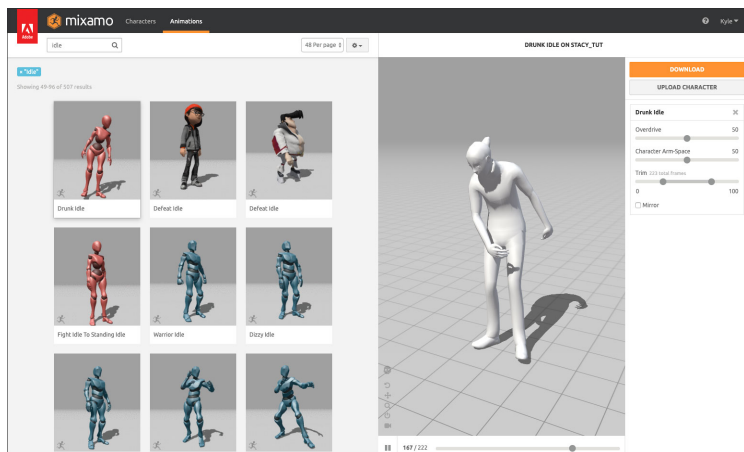


Mixamo has now created a skeleton for your model, this is the skeleton we hooked into in this tutorial.

Click next, and then select the animations tab in the top left. Let's find an idle animation to start with, use the search bar and type 'idle'. The one we used in this tutorial is called "Happy idle" if you're interested.

Clicking on any animation will preview it, explore this site to see some crazy other ones. But an important note: **this particular project works best with animations where the feet end up where they began, in a position similar to our idle animation, because we're cross fading these, it**

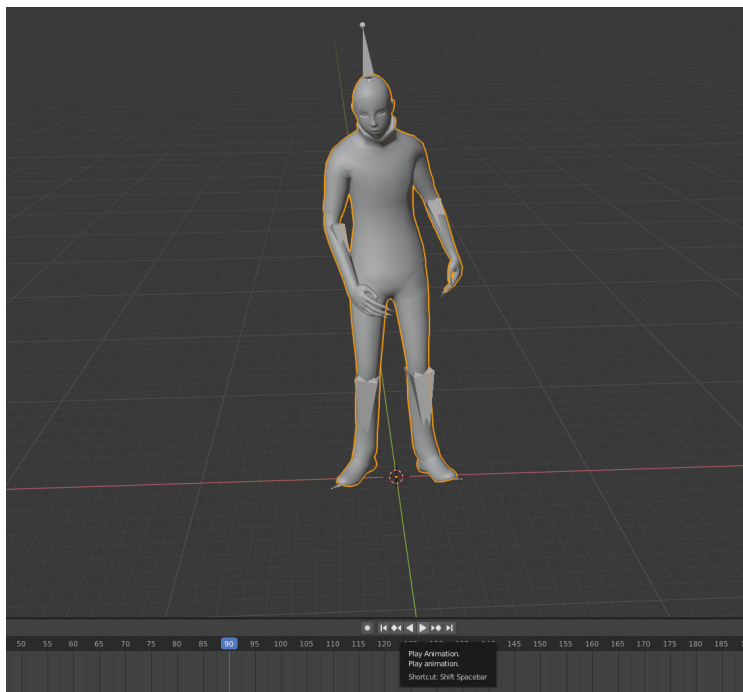
looks most natural when the ending pose is similar to the next animations starting pose, and visa versa.



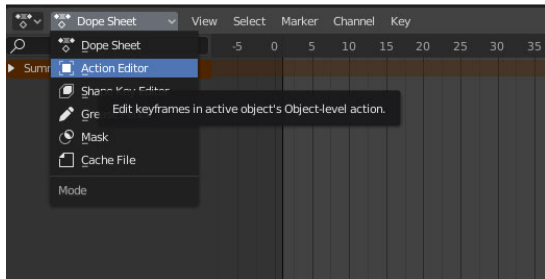
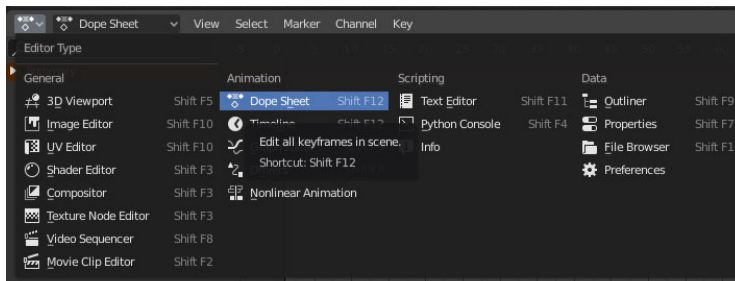
Once you're happy with your idle animation, click *Download Character*. Your format should be FBX and skin should be set to *With Skin*. Leave the rest as default. Download this file. Keep Mixamo open.

Back in Blender, import this file into a new, empty session (remove the light, camera and default cube that comes with a new Blender session).

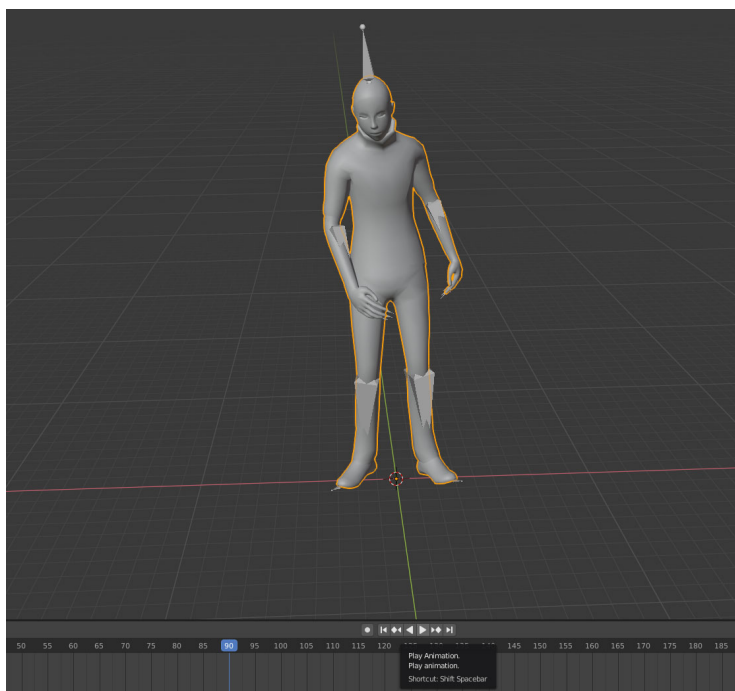
If you hit the play button (if you don't have a timeline in your session, you can toggle the Editor Type on one of your panels, at this point I recommend an intro into Blenders interface if you get stuck).



At this point you want to rename the animation, so change to the Editor Type called *Dope Sheet* and the select *Action Editor* as the sub section.

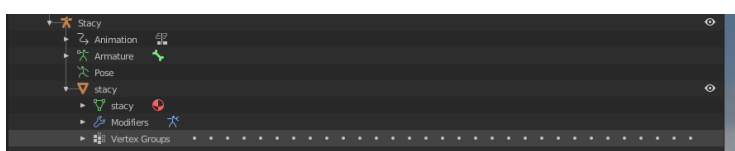


Click on the drop down next to + New and select the animation that Mixamo includes in this file. At this point you can rename it in the input field, lets call it 'idle'.



Now if we exported this file as a GLTF, there will be an animation called idle in gltf.animations. Remember we have both gltf.animations and gltf.scene in our file.

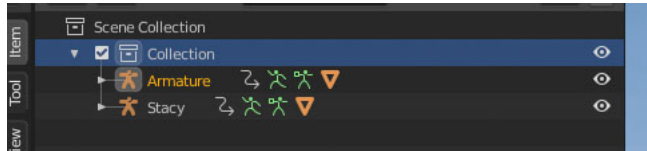
Before we export though, we need to rename our character objects appropriately. My setup looks like this.



Note that the bottom, child *stacy* is the object name referenced in our JavaScript.

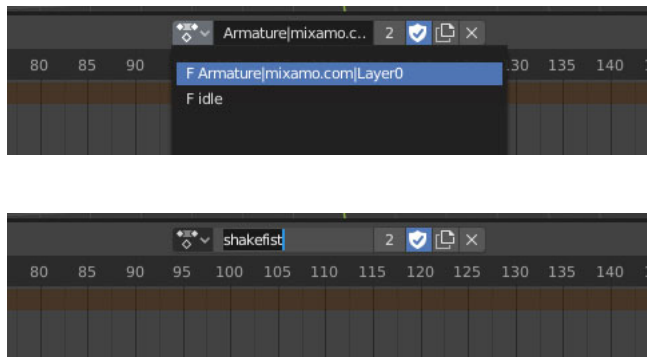
Let's not export yet, instead I'll quickly show you how to add a new animation. Head back to Mixamo, I've selected the Shake Fist animation. Download this file too, we still want to keep the skin, others probably would mention that you don't need to keep the skin this time, but I found that my skeleton did weird things when I didn't.

Let's import it into Blender.



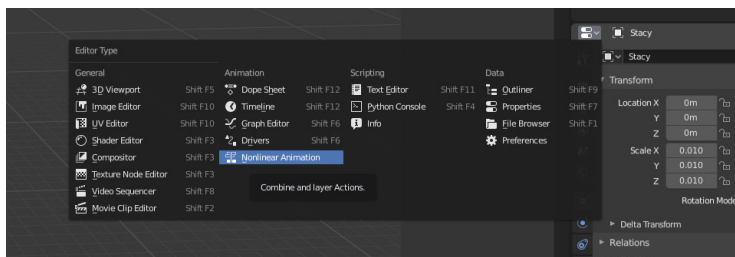
At this point we've got two Stacys, one called Armature, and the one we want to keep, Stacy. We're going to delete the Armature one, but first we want to move its current Shake Fist animation to Stacy. Let's head back to our Dope Sheet > Animation Editor.

You'll see we now have a new animation alongside idle, let's select that, then rename it *shakefist*.



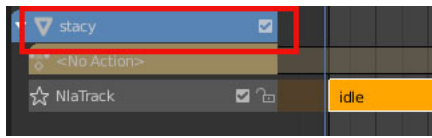
We want to bring up one last Editor Type, keep your Dope Sheet > Action Editor open, and in another unused panel (or split the screen to create a new one, again it helps if you get through an intro into Blenders UI).

We want the new Editor Type to be Nonlinear Animation (NLA).

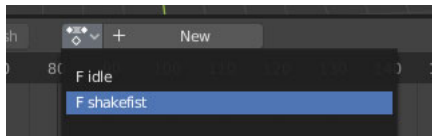


Click on *stacy*. Then click on the Push Down button next to the *idle* animation. We've now added idle as an animation, and created a new track to add our *shakefist* animation.

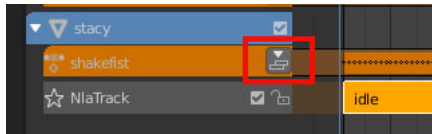
Confusingly, you want to click on *stacy's* name again before we we proceed.



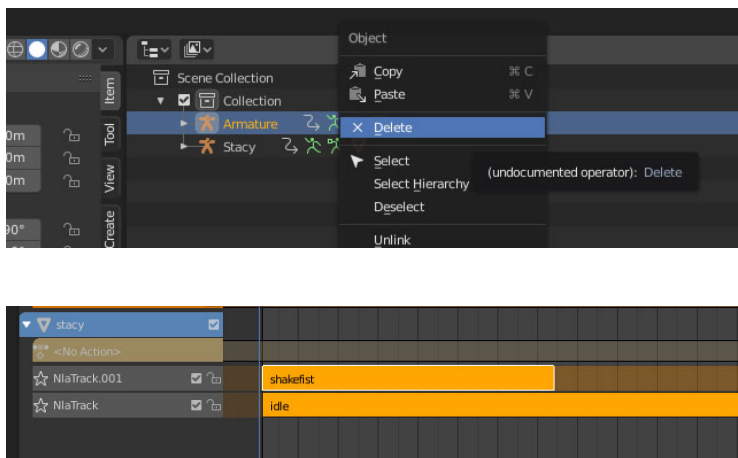
The way we do this is to head back to our Animation Editor and select *shakefist* from the drop down.



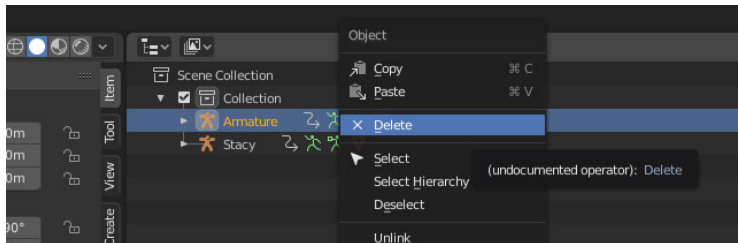
Finally, we can use the Push Down button next to *shakefist* in the NLA editor.



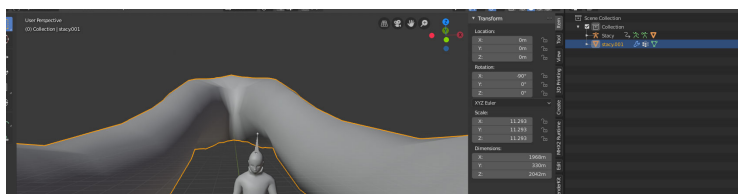
You should be left with this:



We've transferred the animation from Armature to Stacy, we can now delete Armature.

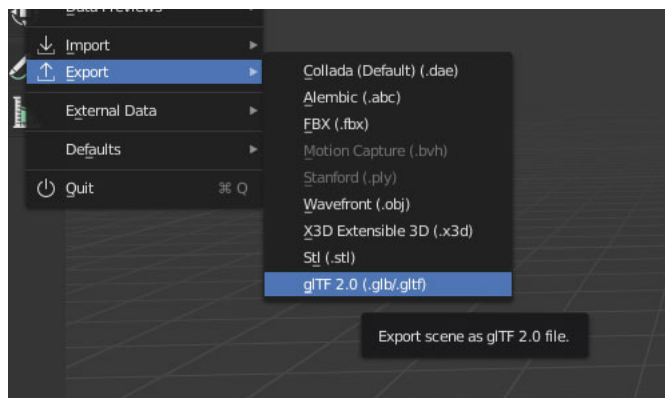


Annoyingly, Armature will drop its child mesh into the scene, delete this too



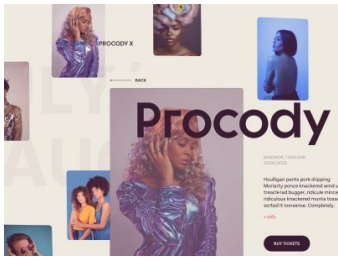
You can now repeat these steps to add new animations (I promise you it gets less confusing and faster the more you do it).

I'm going to export my file though:

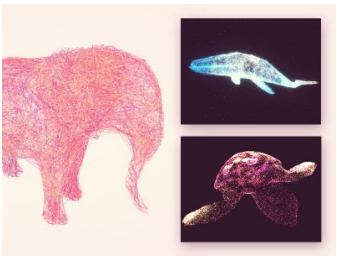


Here's a pen from this tutorial except it's using our new model! (Disclosure: Stacy's scale was way different this time, so that's been updated in this pen. I've had no success at all scaling models in Blender when Mixamo has already added the skeleton to it, it's much easier to do it in Three.js after it's loaded).

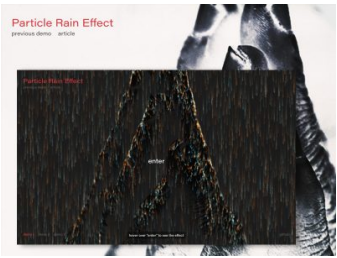
*The end!*



**Magnetic 3D Grid**  
**Interaction with Content**  
**Preview**



**Surface Sampling in**  
**Three.js**



**Tropical Particles Rain**  
**Animation with Three.js**





