Creating a 3D Product Display for Online Shopping Websites with WebGL and Three.js

Christian Norris

Abstract—Online shopping is one of the biggest sectors of retail, but the technology that it is built upon hasn't changed much in the last decade. Many companies have been looking towards 3D Product Displays to enhance the shopping experience for consumers and to get ahead of their competitors in such a cut-throat market. It has been proven through third party studies to improve the emotional response to products and to better brand perception, and if your business would improve from a more immersive shopping experience, this technology could benefit it immensely. Implementing 3D graphics for the browser can be a difficult process, but this paper will teach you how to create your first 3D scene, add custom models and animations, create immersive camera controls, and implement gorgeous lighting. Using the popular WebGL API and three.js library, bringing 3D product displays into your website has never been faster, easier, and more powerful.

Index Terms—Computer graphics, 3D graphics, WebGL, threejs, product display, online shopping, online retail, ecommerce

CONTENTS

Baskin School of Engineering, University of California, Santa Cruz

1 Introduction

Although the online shopping space is one of the most profitable models in business, it still has tons of potential to improve. Throughout the last two decades, companies have spent billions of dollars iterating upon and refining their user experience to maximize customer satisfaction and boost sales. This however has been mostly contained to the typical format we are used to seeing from online websites. Because of this, some people have started to question whether the accepted version of ecommerce we currently have is the most effective. With the recent leaps in modern technologies, many people have looked to 3D product displays as the possible future of the online shopping experience. In this paper, you will learn when and why to use this technology, and most importantly, how to implement it into your own shopping website.

1.1 Why 3D product displays for your website?

If you have an online shop for your retail business, 3D product displays are a great tool for enhancing your user experience. In a market over saturated with options for consumers to choose from, creating an experience that makes shoppers feel more involved in their purchases can make your website stand out from the crowd. A strong and memorable user experience will not only help sell the specific product, but can improve your brand perception, and make the consumer more likely to think of your company over a competitor next time they shop.

1.2 How well do they work?

To determine the efficacy of 3D product displays in online shopping, you can look to popular retailers who have implemented them into their websites over the last few years and determine if they view them as worthwhile investments. One good company to look at is Nike with their *Nike by You* addition to their website, which allows consumers to customize and view their favorite shoes in 3D. Even though this was launched in 2017, new shoes are constantly being added to the service to this day, proving that Nike find it to be a valuable addition to their online shop. Another company that has invested in 3D technology is Ikea with their *Ikea Planner* tools. This allows users to build the layout of their home/office and place 3D models of furniture, letting them personalize and view their dream room in 3D Considering the vast selection of furniture they have added and continue to expand upon, Ikea clearly believe that this technology is beneficial for them.

If you don't trust the companies who have tested 3D product displays on their own websites, you can also look to private studies on the effectiveness of this technology as well. Researchers from Jönköping University in Sweden have looked into this exact topic [1]. They directed users to shop on both Nike and Ikea's websites using their 3D product display technologies, and then surveyed the group on their experience. Over the hundreds of people in this study, the researchers concluded that 3D Product Displays can show products more comprehensively to users than 2D picture displays, allowing users to obtain more information only through visuals, which improves consumers' trust and satisfaction throughout the process. A similar study was conducted on the effects of augmented reality in online shopping by researchers from the University of Porto in Portugal [2]. After having 150 participants in a lab environment shop using both conventional interfaces and augmented reality, it was concluded that the 3D technology lead to a higher emotional response towards the product, as well as a greater affinity towards its brand.

This goes to show that companies who find value in the consumer experience in their online shops can achieve many benefits from 3D product displays, not only in selling their product, but also in improving consumer perception of their brand.

1.3 Are they right for you?

While 3D Product Displays have been proven to bring out positive responses from consumers, not every online store would benefit from these technologies. Because of this, some thought has to go into the decision of whether or not this is right for you. While this can be a nuanced topic, there are a few questions to ask yourself to help weigh the benefits with the potential challenges that it can create. The first question to ask yourself is this: how important is user experience to the product you are selling? An online store that sells jewelry will care much more about this than one that sells office supplies. Another thing to keep in mind is the range of your store's inventory. The number of products and their permanence in your store are two things to take heavily into consideration. If your online store contains thousands of products that are on a seasonal rotation, the time and resource investment for 3D product displays might not be worth it for you.

1.4 Why WebGL and three.js?

For websites built with HTML and Javascript, *WebGL* is the de facto technology for bringing 3D scenes to the web browser. It is a Javascript API built upon OpenGL, and is supported by all modern browsers without any additional downloads or cookies needed by users. *three.js* is a Javascript library built using WebGL, that makes it much easier for developers to add and customize 3D scenes into their websites. This will allow you to easily implement 3D product displays and get a functioning prototype running in a matter of hours.

2 Getting Started

After understanding the benefits of 3D product displays and if they are right for you, you can learn how to implement them on your own website. While there are a few different ways to import three.js into your Javascript project, they all accomplish the same goal of bringing the three.js library into your code.

2.1 Installation method

In this paper, the method that we will use will be simple downloading the entire three.js library and adding it to our project. This is the quickest and easiest way to get it set up, although it might not be best for larger, production scale projects. For those, installation through npm or a CDN (content delivery network) might be preferable, and it is entirely up to the project manager to decide what is best for the specific case.

2.2 Importing three.js

First thing to do is to decide where in your project directory the three.js file should go. For this paper, it will do within the directory `"js/three,js"`. Once you have created a blank .js file, open your web browser and navigate to `www.threejs.org/build/three.js`. Copy all text and paste it into the new file you have created, and just like that, three.js is added to your project!

3 CREATING A BASIC 3D SCENE

In WebGL, 3D environments exist within a *scene* and contain a few elements: the camera, models, and light sources. The scene is the animated each frame and displayed on screen with the renderer. In this section, you will learn how to implement basic versions of all of these, while in future sections some will be expanded upon.

3.1 Including three.js in your website

While the three.js library has been added to your project, it hasn't actually been included into your code yet. In order to do that, you will need to add the following script tag into your html:

```
<script src="js/three.js"></script>
```

This will bring the three.js file into your project. Note that you will have to change the source directory if your three.js file exists within a different directory name.

3.2 Creating your first scene

Now that three.js is included in your website, you can finally start creating your first 3D scene! If you will be typing this code into your project as we go, note that nothing will show up until we render everything in section 3.2.4.

3.2.1 Initializing an empty scene

Open the Javascript file you would like our code to go in and put in this code:

```
const scene = new THREE.Scene();
const camera = new
    THREE.PerspectiveCamera( 75,
    window.innerWidth /
    window.innerHeight, 0.1, 1000 );

const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth,
    window.innerHeight );
document.body.appendChild(
    renderer.domElement );
```

So, what is actually going on here? We have set up the scene, the camera, and the renderer. In WebGL, scenes are 3D spaces in which everything we create exists. The scene is built with an xyz coordinate system, where the xy plane is flat against your computer screen, and the z-axis points out of your screen with the positive direction being towards you.

For the camera, we are using the perspective camera, which are given the values for the field of view, aspect ratio, near clipping plane, and far clipping plane. While the FOV and aspect ratio should be self explanatory, the clipping plane is a lesser known attrubute. Essentially, all objects closer than the near clipping plane and further than the far one won;t get rendered. You won't have to worry about this now, but it might become important to you in the future if you are trying to increase performance in large scenes.

Finally the renderer—what is it? This is where three.js does all of the WebGL heavy lifting for us by bringing your scene to life. It is fairly complicated behind the curtains, and you can change the type of rendering being done, but that is beyond the scope of this project and won't be useful for simple 3D product displays. That being said, in our code all we have to do is create the renderer instance and then set its size. Last but not least, we add the renderer element to our HTML document, and you're done creating an empty scene.

3.2.2 Adding a basic model

Next thing to add to our scene is a basic model. In this example, it will be a cube. Here is the code for that:

```
const object = new THREE.BoxGeometry( 1,
    1, 1 );
const material = new
    THREE.MeshBasicMaterial( { color:
    0xff0000 } );
const cube = new THREE.Mesh( object,
    material );
scene.add( cube );

camera.position.z = 5;
```

While there are a few things happening here, it is important to first know that models in three.js have three main parts: the object, the material, and the mesh. If our model was a person, this would be analogous to the object being their body, the material being their skin, and the mesh being whatever holds them both together.

In the code here, we first create the object with length, width, and height of one. Next we create the material and set it to the color red. While three.js has many different types of materials, here we are just using `MeshBasicMaterial` to keep things simple. The third thing we need is the mesh, which binds the material to the object. After this, we have our final cube model! Only thing left to do is add it to the scene, which is done next. We should also probably move the camera back since both the cube and our camera are positioned at the origin (0, 0, 0) within the scene. And just like that, we have added a cube into our scene!

3.2.3 Implementing basic lighting

While you will learn more light types in section 8, in this example we will implement two basic lights: an ambiest light and a point light. Here is the code for those:

```
const ambientLight = new
    THREE.AmbientLight( 0x404040, 0.5 );

const pointLight = new THREE.PointLight(
    0xff0000, 1 );
```

```
pointLight.position.set( 50, 50, 50 );

scene.add( ambientLight );
scene.add( pointLight );
```

First we create an ambient light, which is a simple universal light applied to everything. We give it the color value of a soft white and the intensity value of 0.5.

Next is the point light. This is a light source that exists at a point within our scene that radiates light uniformly in all directions. Just like the ambient light, we set the color and intensity of the point light.

After we've created both lights, all we have left to do is to simply add them to the scene.

3.2.4 Animation and rendering

Now that we have added everything to the scene, its time to render it so everything will show up. Here is the simplest way to render your scene:

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

This is called a render loop, and it will continuously render new frames for our scene. While there are many different ways to go about rendering the scene, in this example we are using `requestAnimationFrame` because it is simple, and because it pauses when the user does not have its tab currently open. We will go over more rendering and animation techniques in section 9.

Additionally, we can also animate the cube in the animation loop. To add rotation, add these lines of code before the `renderer.render` function call:

```
cube.rotation.x += 0.01;
cube.rotation.y += 0.01;
```

Now everything is finished! To recap: You have created a 3D scene with a single model, camera, and two light sources. This is then animated by rendering the scene within a loop, and to confirm this, the sube is also set to rotate.

4 ADDING CUSTOM MODELS TO YOUR WEBSITE

While being able to easily create simple shapes with three.js is nice, it's not really useful for creating 3D product displays. You will probably want to create a 3D model in a dedicated modeling software, and luckily three.js gives us tools to do just that. One thing to note first however is that in order to import the file loaders that three.js uses, you will need to install the full three.js source code either through npm or with a CDN (content delivery network). This is different than what we covered back in section 2.2, and it is up to you to decide which method is best for you.

4.1 Compatible file types

The recommended file type to use with three.js is *glTF* (gl transmission format), and both .GLB and .GLTF versions are well supported. While many programs allow for glTF export, some of the most popular ones include: Blender, Substance Painter, Modo, Toolbag, and Cinema 4D.

If glTF format is not an option, some popular formats such as FBX, OBJ, or COLLADA are also available and regularly maintained. These formats however are not as fast to send or to load, making them inferior to glTF in our use case, and should only be considered if glTF is not possible.

4.2 Importing models into three.js

If you are using the recommended glTF file format, you first will need to import the glTF loader. Note that for other file formats, refer to section 10.2 for the three.js documentation where you can find the module name of their respective loader. Here is how the code looks for the glTF loader:

```
import { GLTFLoader } from
    'three/examples/jsm/
    loaders/GLTFLoader.js';
```

Note: if implementing in your own code, the file path should be together in one line with the import statement.

Now that you've imported the correct loader, you are ready to bring your model into the scene. While the specific scripts will differ for other loaders, here is what your code should look like for the glTF loader:

```
const loader = new GLTFLoader();

loader.load( 'path/to/your/model.glb',
    function ( gltf ) {
   scene.add( gltf.scene );
}, undefined, function ( error ) {
   console.error( error );
} );
```

There are a few things going on in this code. The `loader.load` function attempts to loaf your model. If it is successful, the first function runs and it gets added to the scene. If it fails, the second function runs and an error is thrown. In section 9.1, we will go further into detail about how you can display the imported model's animations as well.

5 DRAWING LINES

In some cases, you might want to draw lines in three.js, rather than a simple model or its wireframe. Doing so is much different than creating a model, and you will learn how to draw lines in this section. Additionally, sections 5.2 and 5.3 will cover how to draw the axes and a grid respectively.

5.1 Creating custom lines

Drawing lines in WebGL and three.js has two big steps: creating the points/vertices and connecting them together. Before we get into it, here is the driver code we will be using for this example. This is identical to the code from section 3.2.1, except for the added addition of moving the camera back along the z axis.

```
const scene = new THREE.Scene();
const camera = new
    THREE.PerspectiveCamera( 75,
    window.innerWidth /
    window.innerHeight, 0.1, 1000 );
camera.position.set( 0, 0, 50 );
camera.lookAt( 0, 0, 0 )

const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth,
    window.innerHeight );
document.body.appendChild(
    renderer.domElement );
```

5.1.1 Creating the points

After the driver code, you can start creating the list of points and then creating a geometry from that list:

```
const points = [];
points.push( new THREE.Vector3( -10, 0, 0
    ) );
points.push( new THREE.Vector3( 0, 10, 0 )
    );
points.push( new THREE.Vector3( 10, 0, 0 )
    );
```

The `points` array contains all of the points that act as the vertices to our line, and this array can be as long as you want. You could even create a wireframe model using this method, although it would be much more difficult than doing it the normal way.

5.1.2 Connecting them together

Now lets connect the points together:

```
const geomerty = new
    THREE.BufferGeometry.setFromPoints(
    points );
const material = new
    THREE.LineBasicMaterial( { color:
    0x0000ff } );
const line = new THREE.Line( geometry,
    material );

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

There are a few things going on here. The points are combined together to create a geometry in the buffer. We then select a material to set to the line, in this case the color blue. Finally, we combine those two to create our line, add it to the scene, and then render the scene. Note that you can keep adding points to the points array as long as you add them *before* setting the geometry buffer.

5.2 Drawing the axes

While you could draw the xyz axes with simple lines, three.js gives us a much easier way to do so using the axes helper. Here is how you would display the axes:

```
const axes = new THREE.AxesHelper( 5 );
scene.add( axes );
```

The constructor for `AxesHelper` is the size of the axis lines, where 1 is the default value.

5.3 Drawing a grid

If you would like to draw a grid to display the size of a product, three.js provides us with tools to easily set that up as well. Here is what the code would look like:

```
const grid = new THREE.GridHelper(
    10,
    10,
    0xff0000,
    0x555555
);
scene.add( grid );
```

`GridHelper` gives us four constructors. The first is grid size, which has a default of 10. Next is the number of divisions, also with a default of 10. After that is the grid's center line color, which has a default color of `0x444444`. Finally is the rest of the grid lines' color, which has a default color of `0x888888`.

6 OBJECT MOVEMENT

Most objects in three.js will be part of the Object3D class, which provides us with many tools for manipulating them.

6.1 Translation

If you are looking to move your model either initially or in real time, translation is the way you would go about it.

To move an object a set distance along an axis, you can use these class methods: `.translateX(distance)`, `.translateY(distance)`, and `.translateZ(distance)`. You can also get this same effect by doing `.position.x += distance` for the x, y, or z axes. Additionally, there is `.translateOnAxis(axis, distance)` if you would like to go in a specific direction, but you can accomplish this with a combination of the three previous functions, and creating unit vectors is beyond the scope of this paper.

To set an objects exact position rather than translate it, you can do this by setting the `.position` property like so: `.position.set(x, y, z)`. You can also set individual x, y, and z, coordinates: `.position.x = 5` and so on.

For more advanced object translation controls based on camera location, please refer to section 7.2.3 to learn about *transform controls*.

6.2 Rotation

Rotating a model is extremely useful, especially when showing off a product you sell.

To rotate an object a set amount along an axis, you can use these class methods: `.rotateX(rad)`, `.rotateY(rad)`, and `.rotateZ(rad)`. You can also get this same effect by doing `.rotation.x += rad` for the x, y, or z axes.

To set a specific position which you would like your object to point towards with the `.lookAt(x, y, z)` method, which can be useful for resetting a product to face towards the camera. Individual x, y, or z direction can be set directly as well like so: `.rotation.x = rad`

One thing that should be mentioned is that if you are looking to allow users to rotate your product so that you can show off all angles, then doing this through rotating the actual object may not be for you. In section 7.2, we will go over some more advanced camera controls that will allow a more intuitive approach for users by rotating and moving the camera around the product instead of the model itself.

6.3 Scaling

You can also scale models up and down, which is useful for sizing things to scale. If you are looking for a method to allow users to zoom in/out of your product, please refer to section 7.1 for help on how to camera zoom.

To scale an object a set amount, three.js provides you with the `.scale` property. By adding to the property directly, you can stretch/compress it along each axis relative to its current size: `.scale.x += 0.1`. To scale it a set amount, use the `.scale.setX(x)`, `.scale.setY(y)`, and `.scale.setZ(z)` methods, or change the property directly with `.scale.x = 2`. You can also use `.scale.set(x, y, z)` to set all three simultaneously, and `.scale.setScalar(scalar)` to set all three axes to the same scalar.

7 Camera movement

Being able to control the camera in a 3D product display is one of the biggest things that make the shopping experience interactive, and it is without a doubt very important. Because of this, some thought should be put into what kind of camera system would be best for you. In this section, we will go over various popular ways to both move and rotate the camera.

7.1 Zooming in or out with perspective camera

In WebGL and three.js, zooming is mainly done by changing the FOV. Decreasing it zooms the camera in, while increasing it zooms the camera out. While some of the camera controls we will go over in section 7.2 implement it for us, we have to do it ourselves for the normal perspective camera like we used in section 3.2.1.

For the perspective camera, there are two properties we can change in order to zoom the camera in/out: `.fov` and `.zoom`. The camera display is calculated as `fov / zoom`, and since the default zoom is set to 1, changing the FOV alone will allow you to zoom in or out as intended. However if you wanted to keep the original FOV value, you can change the zoom property instead. There is no difference between the two, it is just personal preference.

After changing the fov or zoom property, you will have to call `.updateProjectionMatrix()` for the changes to take effect.

7.2 Advanced camera controls

In this section, we will go over some of the community made camera controls that have been officially added into the three.js library. Due to the complexity of these controls, this paper will not go in depth on how to implement them, but rather we will discuss what each control's strengths are and what some possible use cases for them would be. Please refer to section 10.2 for the official documentation if you would like to read more, or if you would like to view online examples with source code easily available. This is highly recommended if you are interested in implementing advanced camera controls into your 3D product display.

Before we begin, note that in order to use all of these controls, you will need to install the full three.js source code either through npm or with a CDN (content delivery network). This is different than what we covered back in section 2.2, and it is up to you to decide which method is best for you.

7.2.1 Rotating and moving the camera around your object

For many 3D product displays, you will want to show off all angles of your product to users. While it might seem like you should rotate your product model itself to get this effect, rotating and moving the camera around the object provides much more intuitive controls for the user. To get this effect, there are two main camera controls which are recommended: *arcball controls* and *orbit controls*.

Both of these controls allow you to orbit around an object with the camera. Orbit controls have been around for longer, and thus have more examples and tutorials online to help you implement it in your 3D product display a bit easier.

On the flip side, arcball controls have full touch support, added controls through various mouse/touch gestures, and the ability to copy and paste the camera state. In addition to this, the controls are more easily customizable, meaning that you can change the gestures to fit your specific needs if need be. The downside of arcball controls is that it is a newer addition to three.js at the time of this paper being written, so there are fewer resources to troubleshoot your issues and fewer examples to take inspiration from.

Both orbit and arcball are great control schemes, and neither are a bad option. Orbit controls have the advantage of being the tried and true method with tons of online resources to help you get up and running. Arcball controls is the new kid on the block, with tons of features but a steeper learning curve.

7.2.2 First person camera controls

If you are presenting any sort of large and traversable 3D space, such as a showroom, then allowing for first person controls should be a serious consideration. While the controls are more involved for the user than the previous section, first person perspective might be the right fit for you and your 3D product display.

Three.js has two control classes to handle first person perspectives for your camera: first person controls and pointer lock controls. They both do the same thing, however the former does not lock your mouse while controlling the camera, and the latter does. While this might not seem like a huge deal, it actually has a massive impact on the user experience.

When you are implementing a first person control scheme, immersion and the user's experience are probably the first thoughts in your mind. Nothing will break that experience like having wonky camera controls, and with an unlocked mouse, controlling the camera in any meaningful way is much more difficult. While you are encouraged to try out both control schema if you wish, it is recommended that if you decide with a first person perspective for your 3D product display, you chose the *pointer lock controls*.

7.2.3 Moving your object with respect to the camera

In some cases, you might want to provide the user with controls to move objects around in a 3D space. Allowing users to move furniture around in a showroom would be a good use case of such a feature. For this, three.js provides us with the *transform controls* class. It is used to transform objects in 3D space by adapting a similar interaction model of other 3D modeling software.

Transform controls are different from the previous two sections in the fact that these controls don't alter the camera itself, but rather use the camera to determine how to alter the position of the selected object. This can be incredibly useful in cases where you want users to be able to move one or more products around, maybe to view how they look in a certain space or next to each other.

8 BETTER LIGHTING

In WebGL and three.js, there are many different kinds of lighting. Some are softer and used for adding global light, some are much stronger and give very direct light. All of them should be used together and experimented with to make your scene much more real. After all light types are covered, you will learn how to implement shadows for your lighting in section 8.8.

8.1 Realistic lighting render settings

One thing to note is that many light behave slightly unrealistically for various reasons, and three.js gives us a render mode to make them more realistic at a cost of performance. Just after creating the `WebGLRenderer` in your project, you can add this line of code to change the lighting to be more realistic:

```
renderer.physicallyCorrectLights = true;
```

This will make lighting more realistic and change some of the settings on how lights behave and how they cast shadows. Remember that this step is not necessary, but should still be considered if the upgrade in lighting is worth the hit in performance.

8.2 Ambient light

Ambient lighting is a base global light that illuminates all objects in your scene equally. This should never be your only light source, but should be used in addition with one or more direct light types. Here's how you can implement an ambient light in your scene:

```
const light = new THREE.AmbientLight(
    color, intensity );
scene.add( light );
```

The constructors for ambient light are optional and have default values of `0xffffff` and 1 respectively.

8.3 Hemisphere light

Hemisphere light is also intended to be a base lighting for your scene, however it is not a flat global light. It is instead a light source positioned directly above the scene with two colors. One coming from above to simulate something like a sun or ceiling light source, and a secondary color of light coming from below, to simulate the light reflected off the ground. This provides a much more realistic look than a global ambient light. Here is how to implement a hemisphere light:

```
const light = new THREE.HemisphereLight(
    skyColor, groundColor, intensity );
scene.add( light );
```

The constructors for hemisphere light are optional and have default values of `0xffffff`, `0xffffff`, and 1 respectively.

8.4 Point light

Point lights are a single light that gets emitted from a single point in your scene, tapering off in all directions. They work much like light bulbs, can be positioned at a specific location, and allow you to set how far the light travels as well as how much it decays. Here is how a point light can be implemented:

```
const light = new THREE.PointLight( color,
    intensity, range, decay );
light.position.set ( 10, 50, -10 );
scene.add( light );
```

The constructors for point lights are optional and have default values of `0xffffff`, 1, 0 (no max distance), and 1 respectively. For the decay constructor, while 1 is the default value, 2 simulates physically correct lighting.

8.5 Spotlight

Spotlights are like point lights in that they get emitted from a single point, however spotlights only shine light in one direction in a cone shape. The further away from the light source you get, the light will cover more area, however it will also be less intense. Here is how to implement a point light:

```
const light = new THREE.SpotLight(color,
    intensity, distance, angle, penumbra,
    decay);
light.position.set ( -30, 30, 10 );
scene.add( light );
```

The constructors for spotlights are optional and have default values of `0xffffff`, 1, 0, `Math.PI/3`, 0, and 1 respectively. The default position for a spotlight is `(0, 1, 0)`. The angle signifies the maximum angle of light dispersion from the spotlight, and can be any value between 0 and `Math.PI/2`. The penumbra is the percent of the spotlight cone that gets attenuated, and can be any value between 0 and 1. Like the point light, a decay value of 2 will simulate physically correct lighting.

In order to change the direction the spotlight is facing, you have to update its `.target` property. The spotlight points from its position to `.target.position`, which has a default value of `(0, 0, 0)`. If you change the target position, it must be added to the scene with `scene.add( light.target )` so that its position will get updated each frame. You can also have the spotlight target another object in your scene like so:

```
const object = new THREE.Object3D();
scene.add( object );

light.target = object;
scene.add( light.target )
```

Now the spotlight will track that object!

8.6 Directional light

Directional light is a powerful light source that gets emitted in a specific direction. It behaves as though it is infinitely far away and the rays produced from it are all parallel, much like how daylight from a sun would be produced. Here is how you can implement a directional light in your scene:

```
const light = new THREE.DirectionalLight(
    color, intensity );
light.position.set ( 50, 500, -20 );
scene.add( light );
```

The constructors for directional light are optional and have default values of `0xffffff` and 1 respectively. The default position for a spotlight is `(0, 1, 0)`.

In order to change the direction the directional light is facing, you have to update its `.target` property. The directional light points from its position to `.target.position`, which has a default value of `(0, 0, 0)`. If you change the target position, it must be added to the scene with `scene.add( light.target )` so that its position will get updated each frame. You can also have the spotlight target another object in your scene like so:

```
const object = new THREE.Object3D();
scene.add( object );

light.target = object;
scene.add( light.target )
```

Now the directional light will track that object!

8.7 Rectangular area light

The rectangular area light emits light uniformly from a rectangular plane. Although it doesn't cast shadows like other direct forms of light, it can still be used to simulate light sources such as bright windows, strip lighting, or even just a general backlight when combined with other light sources that can cast shadows. Here is how to implement a rectangular area light in your scene:

```
var light = new THREE.RectAreaLight(
    color, intensity, width, height );
light.positon.set( 0, 5, -20 );
scene.add ( light );
```

The constructors for a rectangular area light are optional and have default values of `0xffffff`, 1, 10, and 10 respectively. The default position for a spotlight is `(0, 1, 0)`.

To change the angle in which a rectangular area light points towards, you can use the `lookAt()` function to look at a specific coordinate: `light.lookAt(1, 1, 1)`, or at an existing object: `light.lookAt(object)`. Just make sure that if you use the latter, that object already exists in the scene.

While the rectangular area light is very useful, it has some noticable drawbacks. The first being that it does not cast shadows, so if this is something you would like to do, you will have to find a way around it, such as using other light sources to cast shadows. It also only supports *MeshStandardMaterial* and *MeshPhysicalMaterial*, which can cause issues with some models. Finally, in order to use this light, You have to include *RectAreaLightUniformsLib* into your scene and call its `init()` function. This library does not exist in the three.js file used in section 2.2, so you will have to refer to section 2.1 for other ways of getting the entire three.js library with this included.

8.8 Shadows

In WebGL and three.js, shadows are cast when certain kinds of lights are blocked by an object. In order to allow shadows to be created by objects, you have to allow the object to cast and/or receive shadows by light sources. Some materials/meshes also don't support shadows, so you will have to select your mesh with this in mind.

8.8.1 Which lights cast shadows?

Not all lights cast shadows, but here are the ones that do:

- Point light
- Spotlight
- Directional light

All other light sources in three.js do not cast shadows.

8.8.2 How to implement shadows

Enabling shadows in WebGL is a fairly complicated process, which is why three.js simplifies it greatly for us. First thing we need to do is enable shadows in our renderer like so:

```
const renderer = new THREE.WebGLRenderer();
renderer.shadowMap.enabled = true;
renderer.shadowMap.type =
    THREE.PCFShadowMap;
```

Here we are enabling shadows in the renderer and then setting our preferred shadow type. If nothing is set, `PCFShadowMap` is the default, however we also have `BasicShadowMap` for fast but low quality lighting and `PCFSoftShadowMap` for better soft shadows, especially at lower resolutions.

Next we will create a point light and turn on shadows for the light:

```
const light = new THREE.PointLight(
    0xffffff, 1, 100 );
light.position.set( 0, 10, 4 );
light.castShadow = true;
scene.add( light );

light.shadow.mapSize.width = 512;
light.shadow.mapSize.height = 512;
light.shadow.camera.near = 0.5;
light.shadow.camera.far = 500;
light.shadow.bias = 0.0001;
```

Now our spotlight will cast shadows on objects that either cast receive them. The last five lines are all shadow properties being set, and all values are the default ones except for the shadow bias, which is set to a very small number in order to help with shadow artifacting.

After that, lets create an object to cast shadows but not receive them:

```
const sphereGeometry = new
    THREE.SphereGeometry( 5, 32, 32 );
const sphereMaterial = new
    THREE.MeshStandardMaterial( { color:
    0xff0000 } );
const sphere = new THREE.Mesh(
    sphereGeometry, sphereMaterial );
sphere.castShadow = true;
scene.add( sphere );
```

The default value of `.castShadow` is false. There is a similar property called `receiveShadow` which is also defaults to false. Speaking of which, lets now create a plane that receives shadows but does not cast them:

```
const planeGeometry = new
    THREE.PlaneGeometry( 20, 20, 32, 32 );
const planeMaterial = new
    THREE.MeshStandardMaterial( { color:
    0x00ff00 } )
const plane = new THREE.Mesh(
    planeGeometry, planeMaterial );
plane.receiveShadow = true;
scene.add( plane );
```

And now you are done! just finish rendering the scene like you normally would, and you should now have a scene with a spotlight that uses a sphere to cast shadows onto a plane.

9 ADVANCED ANIMATION

While it is possible to do basic animation by moving objects, lights, and the camera around in the animation loop from section 3.2.4, three.js provides us with tools to create and control much more advanced movements. This is extremely important in 3D product displays, because it allows you to show off your product to users in the perfect way to fit your vision. The three.js animation system allows you to animate many things such as:

- The bones of a skinned and rigged model
- Material properties such as colors and opacity
- Object transformation
- Target morphing

The animations can also have added effects such as fading in, fading out, crossfading, warping, adjusted time scales, and synchronization between animations on both the same and different objects. Note that this animation system was completely overhauled in 2015 to be at the level it is at now, so any information online from before then is most likely outdated.

The animation system in three.js has multiple parts that all work together to create a cohesive structure. The remaining subsections will talk about each of these parts, with section 9.1 going over how to import animations with your custom models.

9.1 Importing animations with your imported models

While it is possible to build advanced custom animations within three.js, it is highly recommended to create the animations in your preferred digital content creation software that can create 3D model animations. Please refer to section 4.1 for the recommended file types and a list of popular programs which can export in it. For the entirety of section 9, we will be using the `glTF` loader. Here is the starter code from section 4.2:

```
import { GLTFLoader } from
    'three/examples/jsm/
    loaders/GLTFLoader.js';

const loader = new GLTFLoader();

loader.load( 'path/to/your/model.glb',
    function ( gltf ) {
  scene.add( gltf.scene );
}, undefined, function ( error ) {
   console.error( error );
} );
```

While all this code does right now is load in a glTF model, we will expand upon this in the upcoming sections and allow you to extract and play the animations form it as well. In the `function( gltf )` block from the previous section, the `gltf` object has multiple properties that can be used. Two of them however are the most important in the three.js animation process: `gltf.scene` and `gltf.animations`. We will go over these two in more detail in sections 9.4 and 9.5 respectively.

9.2 Animation clips

The `AnimationClip` class represents an animation which is represented by a set of `KeyframeTrack` objects. While animation clips can be created directly in three.js by editing various properties of a model keyframe by keyframe, this is incredibly inefficient, and why it is recommended to animate in a dedicated program. If you are importing a model that already has animations, `AnimationClip` objects can be easily extracted, and you will learn how to do so in sections 9.4 and 9.5.

9.3 Keyframe tracks

The `KeyframeTrack` class represents a timed sequence of keyframes, which are used to animate an object. Each keyframe is on object composed of two lists: a *times* list and a *values* list. The times list contains the time values for each keyframe in sequential order, while the values list contains the corresponding changing values of the animated property, such as movement, color, or opacity. `KeyframeTrack` has multiple subclasses to handle different types of animated values, however you wont need to worry about this if you are importing animations.

If you are curious to see how animations can be created from scratch with `AnimationClips` and `KeyframeTracks`, you can view some examples of custom-made animations in the *AnimationClipCreator.js* file within the `/examples/jsm/animation` directory in the three.js source code.

9.4 Animation mixer

The `AnimationMixer` is a class that plays the data stored by an `AnimationClip` object. In addition to simply playing animations for a model, the animation mixer can control multiple animations simultaneously by mixing, blending, and merging them. When multiple models in your scene are animated independently, one `AnimationMixer` may be used for each object.

To initialize the animation mixer, you will need to pass it the models data. Continuing our example from section 9.1, we can use the imported model. This is stored in `gltf.scene`, which is of type `THREE.Group` and contains the actual model you are importing. After adding the model to our scene in the `function( gltf )` block, we can create an animation mixer from it with this command:

```
mixer = new THREE.AnimationMixer(
    gltf.scene );
```

And just like that, our animation mixer is generated! In the next section, we will discuss a class method from `AnimationMixer` that will allow you extract individual animations from the mixer.

9.5 Animation actions

The `AnimationActions` class schedules the performance of the animations stored in `AnimationClip` objects. It can be configured to determine when an animation should be played, paused, or stopped on one of the mixers. It can also direct a mixer to repeat the animation, scale it with time, synchronize with other animations, and perform fading in/out as well as crossfading.

To create an `AnimationAction` instance, you will need to pass it the animation data from your model. Continuing our previous example, this data is stored in the `gltf.animations` property. This property is a list containing all of the animations for the imported model, all of type `AnimationClips`. For example, if the model is a person, the `gltf.animations` list may contain one `AnimationClip` for a walkcycle, a second for running, a third for jumping, and so on.

The actual `AnimationAction` instance can be generated with an `AnimationMixer` class method called `.clipAction`. We can use the mixer instance we made from our model and pass an element from our list of animations into this function, and it will automatically generate an `AnimationAction` from the given `AnimationClip`. After creating the mixer in section 9.4, here is how you would do that:

```
walkAction = mixer.clipAction(
    gltf.animations[0] );
runAction = mixer.clipAction(
    gltf.animations[1] );
jumpAction = mixer.clipAction(
    gltf.animations[2] );

let actions = [ walkAction, runAction,
    jumpAction ];
```

Now we have a list of `AnimationAction` objects, which contain all of the animations for our model. With this, and we can easily perform all of the different animation effects that three.js gives us access to. While there are too many to list here, please refer to section 10.2 for the official documentation. It is highly recommended to read through the documentation for `AnimationAction` to get an understanding of all of the different ways you can animate your models.

9.6 Reviewing what we learned

And just like that, we have covered advanced animation in three.js! After completing this section, you know how to import animations with a model and store them in three.js's animation system. To view all of the animation methods that exists within three.js, please refer to the `AnimationAction` for details on everything possible. To understand how to create your own animations within three.js instead of importing them, please refer to the examples of custom-made animations in the *AnimationClipCreator.js* file within the `/examples/jsm/animation` directory in the three.js source code. Finally, if you would like to animate multiple objects together as if they were the same, you can learn about grouping animated objects in the following section, 9.7.

9.7 Animation object groups

The `AnimationObjectGroup` class allows you to group objects together so that they share the same animation state. You can do this by creating a group out of multiple objects like this:

```
const animationGroup = new
    THREE.AnimationObjectGroup( obj1,
    obj2, ... );
```

You can then either set this as the root object in the `AnimationMixer` constructor by passing it in as the first argument, or you can override the mixer's default root object by passing it in as the second argument of the `.clipAction` method. Either way, this will either allow you to control multiple objects with the same `AnimationAction` instance.

While this can be very convenient, there are a couple of things to note. The animated properties must be compatible among all objects in the group. Additionally, a single property can either be controlled through a target group or directly, but not both.

10 PARTING WORDS

After following through the entire paper, you should understand what 3D product displays are, when they are useful, why they are impactful. Additionally, you should have a solid foundation for implementing a 3D product display in your own online shopping website.

10.1 Remembering the *who*, the *what*, and the *why*

Although we've spent the majority of this paper learning *how* to implement a 3D product display, it is equally important to remember the *who*, the *what*, and the *why*.

Who is the product display for? It is important to keep your targeted consumer base in mind. Are they young or old? Will they care more about flashy colors or a minimalistic style? Don't lose sight of who you are trying to appeal to when showcasing your product.

What kind of product are you displaying? Do you sell furniture? If so, spend some resources designing a beautiful room to showcase it in. Are you selling jewelry? Then it might be work putting a little more effort into the lighting to make sure it really pops. Your product is the main star for the display, and everything you put into your 3D scene should contextualize and compliment it.

Why did you choose to display the product in 3D? What are the features that could be better enhanced through a 3D product display over a conventional interface? What kind of experience are you trying to create for your consumer with the display? Staying grounded in your original plan is crucial into making sure that your 3D product display gives your consumers the best experience possible.

10.2 Official documentation

After all of the information covered in this paper, or maybe even because of it, you might be left with questions about some of the material, methods, or features that three.js has to offer. If this is the case, or if you just want to learn more, please visit the official documentation at https://threejs.org/docs/. You can also visit https://threejs.org/examples/ for live demos and code examples of various features in three.js.

ACKNOWLEDGEMENTS

While writing this paper, I found great help from the wonderfully written documentation for three.js on the official website mentioned in section 10.2. It is full of clearly formatted information on almost everything contained in the library, and served as a huge inspiration for the format of many of the things taught throughout this paper.

Another thank you to Professor Alex Pang of the University of California, Santa Cruz for teaching me all about the WebGL API in a computer graphics class. This serves as a basis for my knowledge in how computer graphics works, and how it can be optimized.

A final thanks goes to Michael Herzog, otherwise known as Mugen87, who is a main contributor to the three.js library. At the time writing, he also has posted almost nine thousand times and supplied over a thousand solutions on the three.js forums. For the majority of questions I had about features in three.js, it seemed like he was always who had already answered them on the forums.

REFERENCES

[1] B. Ferwerda and M. Lindh, "3D Product Display in eCommerce Websites," May 2021.
[2] P. Q. Brito, J. Stoyanova, and A. Coelho, "Augmented reality versus conventional interface: Is there any difference in effectiveness?," Multimed Tools Appl, vol. 77, no. 6, pp. 7487–7516, Mar. 2018, doi: 10.1007/s11042-017-4658-1.