Add Animations for Button State Transitions

To make the buttons more responsive to user interactions, we want to add some hover states and transitions between the default idle and the hover state. To do this, we will use the Animated library and Easing functions, and then write to functions for each transition: `animateIn` and `animateOut`:

```
/* Button.js */
import React from 'react';
import {
  Animated,
  asset,
  Image,
  View,
  VrButton,
} from "react-vr";

const Easing = require('Easing');

class Button extends React.Component {

  constructor(props) {
    super();

    this.state = {
      animatedTranslation: new Animated.Value(0),
    };
  }

  animateIn = () => {
    Animated.timing(
      this.state.animatedTranslation,
      {
        toValue: 0.125,
        duration: 100,
        easing: Easing.in,
      }
    ).start();
  }

  animateOut = () => {
    Animated.timing(
      this.state.animatedTranslation,
      {
        toValue: 0,
        duration: 100,
        easing: Easing.in,
      }
    ).start();
  }

  onButtonClick = () => {
    this.props.onClick();
  }

  render() {
    return (
      <Animated.View
        style={{
          alignItems: 'center',
          flexDirection: 'row',
          margin: 0.0125,
          transform: [
            {translateZ: this.state.animatedTranslation},
          ],
          width: 0.7,
        }}
      >
        <VrButton
          onClick={this.onButtonClick}
          onEnter={this.animateIn}
          onExit={this.animateOut}
        >
          <Image
            style={{
              width: 0.7,
              height: 0.7,
            }}
            source={asset(this.props.src)}
          />
        </VrButton>
      </Animated.View>
    );
  }
};

export default Button;
```

After adding the dependencies, we define a new state to hold the translation value we want to animate:

```
/* Button.js */
constructor(props) {
  super();

  this.state = {
    animatedTranslation: new Animated.Value(0),
  };
}
```

Next, we define two animations, each in a separate function, that describe the animation playing when the cursor enters the button, and when the cursor exits the button:

```
/* Button.js */
animateIn = () => {
  Animated.timing(
    this.state.animatedTranslation,
    {
      toValue: 0.125,
      duration: 100,
      easing: Easing.in,
    }
  ).start();
}

animateOut = () => {
  Animated.timing(
    this.state.animatedTranslation,
    {
      toValue: 0,
      duration: 100,
      easing: Easing.in,
    }
  ).start();
}
```

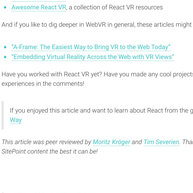
To use the `state.animatedTranslation` value in the JSX code, we have to make the `<View>` component animatable, by adding `<Animated.view>`:

```
/* Button.js */
<Animated.View
  style={{
    alignItems: 'center',
    flexDirection: 'row',
    margin: 0.0125,
    transform: [
      {translateZ: this.state.animatedTranslation},
    ],
    width: 0.7,
  }}
>
```

We will call the function when the event listeners `onButtonEnter` and `onButtonExit` are triggered:

```
/* Button.js */
<VrButton
  onClick={this.onButtonClick}
  onEnter={this.animateIn}
  onExit={this.animateOut}
>
```

A test of our code in the browser should show transitions between the position on the z-axis of each button:



via GIPHY

Building and Testing the Application

Open your app in a browser that supports WebVR and navigate to your development server, by using not `http://localhost:8081/vr/index.html`, but your IP address, for example, `http://192.168.1.100:8081/vr/index.html`. Then, tap on the `View in VR` button, which will open a full-screen view and start the stereoscopic rendering.

To upload your app to a server, you can run the npm script `npm run bundle`, which will create a new folder `build` within the `vr` directory with the compiled files. On your web server you should have the following directory structure:

```
Web Server
├─ static_assets/
|
├─ index.html
├─ index.bundle.js
└─ client.bundle.js
```

Further Resources

This is all we had to do create a small WebVR application with React VR. You can find the entire project code on GitHub.

React VR has a few more components we did not discuss in this tutorial:

- There is a `Text` component, to render text.
- Four different light components can be used to add light to a scene: `AmbientLight`, `DirectionalLight`, `PointLight`, and `SpotLight`.
- A `Sound` component adds spatial sound to a location in the 3D scene.
- To add videos, the `Video` component or the `VideoPano` component can be used. A special `VideoControl` component adds controls the video playback and its volume.
- With the `Model` component we can add 3D models in the `obj` format to the application.
- A `CylindricalPanel` component can be used to align child elements to the inner surface of a cylinder, for example, to align user interface elements.
- There are three components to create 3D primitives: a `sphere` component, a `plane` component and a `box` component.

Also, React VR is still under development, which is also the reason for it running only in the Carmel Developer Preview.

- React VR Docs
- React VR on GitHub
- Awesome React VR, a collection of React VR resources

And if you like to dig deeper in WebVR in general, these articles might be right for you:

- "A-Frame: The Easiest Way to Bring VR to the Web Today"
- "Embedding Virtual Reality Across the Web with VR Views"

Have you worked with React VR yet? Have you made any cool projects with it? I'd love to hear about your opinions and experiences in the comments!

> If you enjoyed this article and want to learn about React from the ground up, check out our course: React The ES6 Way

This article was peer reviewed by Moritz Kröger and Tim Severien. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

More: React VR, virtual reality

Meet the author

Michaela Lehr

Michaela is a front-end developer and UX designer from Berlin. She has co-founded the development studio GeilDanke. In her free time she enjoys making games, practicing yoga, surfing and knitting.