



Sign
in



Building 3D Web Components



Josch Rossa · [Follow](#)

7 min read · Mar 29, 2020



10



Web Component standard meets three.js and WebGL

The goal of this article is to share the experience of building a Web Component with 3D content

Circle Around Component



Web Component with 3D content

Demo: <https://josros.gitlab.io/circle-around-component/>

Repository: <https://gitlab.com/josros/circle-around>

The internet didn't exist at the time when I was born. When I was five years old, Sir Tim Berners-Lee invented HTML, the foundation of the world wide web. In the late 1990's, I did my first steps in the internet. Starting with email and chats I was quickly interested in how the internet works and therefore I taught myself HTML and CSS.

The internet was different back then. There was no material design or css frameworks such as bootstrap for example. Creating a website was a game with almost no rules. For my first layout, I used frames in combination with tables. It must have looked awful, but for me it was absolutely fascinating to see the visual result of just writing some lines of text.

My fascination has never passed and because of those early experiences I studied computer science and became a fullstack web-developer, who carefully observes the ongoing development of the world wide web.

3-Dimensional Web

The world wide web is heading into 3-dimensional space

Looking into my crystal ball, I see the internet becoming more 3-dimensional before it slowly drifts into virtual and augmented reality on its way to its ubiquity. Not for all applications of course, but it will make sense in many fields (e.g. education, sales, games, etc.).

Technologies such as [A-Frame](#), [three.js](#) or progress on the [WebXR](#) standard pointing into that direction. The [three.js website](#) links some demonstrations of 3-dimensional websites.

Web Components

Web Components standardize a component model for the Web

Some other web standards were recently worked out under the name “Web Components”, which standardize a component model for the Web. Such a standardized component has the advantage, that it can be integrated with anything on the Web. It integrates with a static webpage as well as with modern single page application frameworks (such as Vue.js, Angular.js or React.js).

The interface to the world of a single Web Component is the so called *custom-element*. It's a fully-featured DOM element. Compared to basic html elements such as the `<p>` element, you define the name and the behaviour of your *custom-element* yourself. You could for example create a `<preferred-wine>` element, that renders an image of your preferred wine bottle. Internally your preferred-wine Web Component could for example request an http resource every few seconds to update the image of your preferred wine. The example is probably stupid, but you can imagine, that behind a single *custom-element* a lot of magic may happen.

One last thing to say about *custom-elements* is, that the outside world — the DOM — can communicate with the inside world of the Web Component using attributes and event handler.

3-Dimensional Web Component

Let's combine recent technologies and trends and develop a Web Component with 3D content

Because I am curious about the evolvement of the Web, I wanted to combine recent technologies and trends developing a Web Component with 3D content.

Of course such a thing already exists (e.g. the <model-viewer> from google), but I wanted to build it on my own, to see how it could work under the hood. Let's see what I have done:

<circle-around> Web Component

The idea was to build a *<circle-around>* Web Component which renders a 3D model:

```
<circle-around gltfFile="./path/to/whatever/model.gltf"></circle-around>
```

Using the component you would simply assign a .gltf 3D model file path to the *gltfFile* Attribute of my *<circle-around>* custom-element and the component renders the given model on a canvas with a camera, circling around the model. The images below show the idea. A person model is rendered in a scene and the camera circles around.

Circle Around Component



`<circle-around>` Web Component on a demo page showing the 3D model of a person

Circle Around Component



<circle-around> Web Component on a demo page with the circle animation stopped

Web Component Starter

As I quickly wanted to start with the development of the Web Component, I used my [Web Component starter](#) hosted on gitlab.

The starter has the advantage that it already comes with typescript and test support you don't want to miss when developing a production ready Web Component (even though that was definitely not my goal here, I at least wanted some type support). The starter also includes [LitElement](#), which makes life a little easier building Web Components.

Basic Web Component structure

The basic structure of the `<circle-around>` Web Component with LitElement looks as follows:

```
1  import { LitElement, html, css, customElement, TemplateResult, property, CSSResult } from 'lit';
2
3  @customElement('circle-around')
4  export class CircleAround extends LitElement {
5
6      @property({ type: String })
7      private gltfFile!: string;
8
9
10     render(): TemplateResult {
11         return html`
12             <div class="main-container">
13             </div>
14         `;
15     }
16
17     static get styles(): CSSResult {
18         return css`
19             .main-container {
20                 display: flex;
21                 flex-direction: column;
22                 align-items: center;
23             }
24         `;
25     }
26 }
```

As you can see, with LitElement I define my custom-element with:

```
@customElement('circle-around')
```


LitElement takes care of the boiler plate under the hood:

```
class CircleAround extends HTMLElement {...}  
window.customElements.define('circle-around', CircleAround);
```

The .gltf file I want to load is defined as property of our component:

```
@property({ type: String })  
private gltfFile!: string;
```

The property is of type *String*, because from outside of the component I simply want to handover the path to the file. How the file is loaded is the problem of the Web Component internally.

render() is the function that defines the html structure of the component (what is later seen in the browser and hidden behind the Web Components *shadow-dom*). The template that is rendered can itself use custom-elements (as I'll show you later).

In the *styles()* function I can define css and use it in our template.

Loading a .gltf File model

Now that you have seen the basic structure of our LitElement based Web Component. How is the .gltf file loaded based on the path from outside the component?

Here is how it works:

```
1  import { GLTFLoader, GLTF } from 'three/examples/jsm/loaders/GLTFLoader';
2  import { DRACOLoader } from 'three/examples/jsm/loaders/DRACOLoader';
3
4  @customElement('circle-around')
5  export class CircleAround extends LitElement {
6
7      @property({ type: String })
8      private gltfFile!: string;
9
10     @property({ type: Object, attribute: false })
11     private object!: GLTF;
12
13     connectedCallback(): void {
14         super.connectedCallback();
15         if (this.gltfFile) {
16             this.loadGltf();
17         }
18     }
19
20     private loadGltf(): void {
21         const loader = new GLTFLoader();
22
23         // Optional:
24         const dracoLoader = new DRACOLoader();
25         dracoLoader.setDecoderPath('three/examples/jsm/libs/draco/');
26         loader.setDRACOLoader(dracoLoader);
27
28         loader.load(
29             this.gltfFile,
30             // called when the resource is loaded
31             (gltf) => {
32                 this.object = gltf;
33             },
34             (xhr) => {
35                 // Maybe do something with the progress
36             },
37             // called when loading fails
38             (err) => {
```

```
39         console.error('An error happened loading .gltf file. Debug to gain more info');
40     }
41     );
42 }
43 }
```

The *connectedCallback* is a Web Component lifecycle callback. It is called as soon as it is mounted into the DOM for the first time. At that time, it starts to load the .gltf model using the *GLTFLoader*. When the model is loaded, the model is assigned to the property *object*. The *object* is not exposed via the custom-element, nevertheless I want it to be a property to observe its changes (Observer pattern).

Render a scene

As soon as the *object* property gets a new .gltf model assigned (see Loading a .gltf file model). The Web Component update lifecycle is aware of that. In the lifecycle hook I can detect whether *object* has changed:

```
updated(properties: Map<string, string>): void {
  if(properties.has('object')) {
    this.renderSceneWith(this.object.scene);
  }
}
```

and in that case I render the scene:

```
1  import * as THREE from 'three';
2
3  //...
4
5  private renderer?: THREE.WebGLRenderer;
6  private scene?: THREE.Scene;
```

```

7  private camera?: THREE.PerspectiveCamera;
8
9
10 private renderSceneWith(loaderObject: THREE.Group): void {
11     const objectPosition = new THREE.Vector3(0.0, 0.0, 0.0);
12     this.camera = this.createCamera(objectPosition);
13     this.scene = new THREE.Scene();
14
15     // adjust object and add to scene
16     loaderObject = this.adjustObject(loaderObject, objectPosition);
17     this.scene.add(loaderObject);
18
19     this.addLights(this.scene);
20     this.addPlane(this.scene);
21
22     this.renderer = this.createRenderer();
23     this.renderer.render(this.scene, this.camera);
24 }

```

There is a lot going on “behind the scenes”. :-D

You can find the complete example under:

<https://gitlab.com/josros/circle-around>, but you probably get the basic concept by this excerpt. In the snippet above, I do something less relevant with the model recently loaded, before I add it to the scene. In the scene I need lights (otherwise the screen is dark as the night) and a plane (basically the ground our object “stands” on). To render the scene I need a camera that defines what part of the scene should be focussed. *createCamera* does nothing more than:

```

private createCamera(cameraTarget: THREE.Vector3):
THREE.PerspectiveCamera {
    const camera = new THREE.PerspectiveCamera(50, 1, 0.1, 2000);

```

```
camera.position.copy(new THREE.Vector3(0.0, 220.0, 600.0));
camera.lookAt(cameraTarget);
camera.updateProjectionMatrix();
return camera;
}
```

As the *object* (the .gltf model) is located at position $x=0,y=0,z=0$, the camera should be in front. That's why its z position is set to 600. Furthermore, it should look at the *object* from above. That's why its y position is 220. For the instantiation of the *PerspectiveCamera* have a look at: <https://threejs.org/docs/#api/en/cameras/PerspectiveCamera>

The update of the *ProjectionMatrix* is important, because I changed the camera's position.

To finally render the scene I need to create a *THREE.WebGLRenderer*:

```
private createRenderer(): THREE.WebGLRenderer {
  const renderer = new THREE.WebGLRenderer({ antialias: true });
  renderer.setSize(this.width, this.height);
  // we create the canvas based on the renderers dom element
  this.canvas = renderer.domElement;
  return renderer;
}
```

As the purpose of this project is a simple demonstration, I give the *renderer* a fixed size (the component is not responsive). The fixed size finally leads to an html *canvas* element that also has a fixed size.

The important part is *this.canvas = renderer.domElement*. Because *canvas*

is a *property*:

```
@property({ type: Object, attribute: false })
private canvas!: HTMLCanvasElement;
```

the update lifecycle kicks in and the *canvas* is rendered into my Web Component template:

```
render(): TemplateResult {
  return html`
    <div class="main-container">
      ${this.canvas}
    </div>
  `;
}
```

Of course that's only happening if *render()* is called on the *renderer* instance variable:

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

Circle the camera around the model

One last piece missing is that for a “*circle-around*” component, the camera should indeed circle around our model. To achieve that, I use three.js OrbitControls and apply them to the camera:

```
1 import { OrbitControls } from 'three/examples/jsm/controls/OrbitControls.js';
```

```
2
3 // ...
4
5 private renderSceneWith(loadedObject: THREE.Group): void {
6
7     //...
8
9     this.orbitControls = this.createOrbitControls(this.camera, this.renderer);
10    this.animateScene();
11 }
12
13 private createOrbitControls(camera: THREE.Camera, render: THREE.WebGLRenderer): OrbitControls {
14     const orbitControls = new OrbitControls(camera, render.domElement);
15     orbitControls.autoRotate = true;
16     orbitControls.autoRotateSpeed = 2;
17     return orbitControls;
18 }
```

circleAroundCircleCamera.ts hosted with ❤ by GitHub

[view raw](#)

After creating *OrbitControls* I set the *autoRotate* flag to *true*, which makes the camera rotating automatically in the animation function below:

```
1 private animateScene(): void {
2     requestAnimationFrame(() => this.animateScene());
3     if (this.orbitControls && this.doCircle) {
4         this.orbitControls.update();
5     }
6     if (this.camera && this.scene && this.renderer) {
7         this.renderer.render(this.scene, this.camera);
8     }
9 }
```

circleAroundAnimateScene.ts hosted with ❤ by GitHub

[view raw](#)

The animation is called recursively, which makes it render the scene all the time, moving the camera on the defined orbit around our model (which is exactly what I intended to do). The animation does not move

the camera if

```
this.doCircle = false
```

which will be explained in the next section.

Stop the circle animation

To complete the story, I wanted to demonstrate that another Web Components can be used inside our circle-around component. Have a look at:

```
1  import '@material/mwc-icon-button-toggle';
2  import '@material/mwc-linear-progress';
3
4  // ...
5
6  private circleModeChanged(event: CustomEvent): void {
7      if (event.detail.isOn) {
8          this.doCircle = true;
9      } else {
10         this.doCircle = false;
11     }
12 }
13
14 render(): TemplateResult {
15     return html`
16         <div class="main-container">
17             ${this.canvas} ${this.object ? this.renderBtn() : this.renderLoad()}
18         </div>
19     `;
20 }
21
22 renderLoad(): TemplateResult {
23     return html`
```



```

24         <mwc-linear-progress style="width: ${this.width}px" class="progress" index=
25     `;
26 }
27
28 renderBtn(): TemplateResult {
29     return html`
30         <div class="button-container">
31             <mwc-icon-button-toggle on @MDCIconButtonToggle:change="${this.circleModeC
32                 <svg slot="offIcon">
33                 <!-- ... -->
34             </svg>
35             <svg slot="onIcon">
36                 <!-- ... -->
37             </svg>
38         </mwc-icon-button-toggle>
39     </div>
40     `;
41 }

```

The *mwc-icon-button-toggle* is part of the material-components-web project, which provides basic material design ui components as Web Components. In my case, I use the button and register my `circleModeChanged` function to its change event. This makes the circle animation stop, as soon as the button changes its toggle state (e.g. if it gets clicked). The animation stops and can be started again.

This was the story of my first Web Component with 3D content. Should my crystal ball tell the truth, it wasn't the last one, but as a german comedian once said:

Predictions are never easy, especially if they address the future. (Karl Valentin)

Web Development

Web Components

Threejs



Written by Josch Rossa

27 Followers


Fullstack Web-Developer & nature lover

Follow



More from Josch Rossa




 Josch Rossa

Responsive Web Components

This article shares two thoughts about the challenge that Web Components must sta...

4 min read · Nov 15, 2020

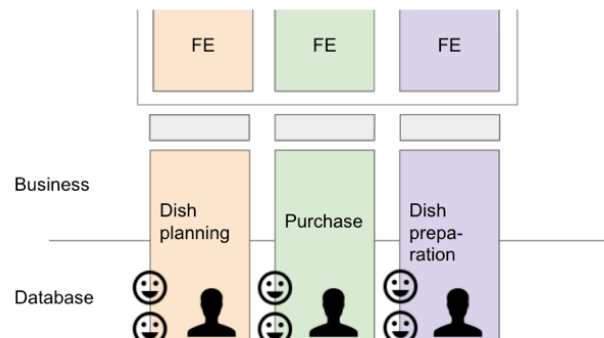



 Josch Rossa

Micro Frontend federation today and tomorrow

Opportunities of Micro Frontends

9 min read · Aug 2, 2020



 Josch Rossa

Scaling software development with Micro Frontends

A quick run through different software architectural approaches and how they...

6 min read · Sep 11, 2021



See all from Josch Rossa

Recommended from Medium



Franky Hung

Make Your Own Earth in Three.js

Hey folks, it's been a long while since my last article here...But I'm back in the game!...

20 min read · Oct 8, 2023

 73 



Salman Sherin

Unlocking 3D Worlds: A Guide to Photogrammetry with JavaScript

Introduction: In the realm of 3D modeling and spatial reconstruction,...

2 min read · Dec 6, 2023

 58 



Lists

Coding & Development

11 stories · 522 saves

General Coding Knowledge


20 stories · 1055 saves

Tech & Tools

16 stories · 189 saves

Stories to Help You Grow as a Software Developer


19 stories · 930 saves

 Sudheer Kumar Reddy Gowrigari

Using Templates and Slots in Web Components: Crafting Dynamic...

Web Components have heralded a new era in web development by offering a native...

2 min read · Oct 6, 2023

 Ben Schiller

Infinite Art, Finite Code

The Vision of reGoosinals


4 min read · Dec 29, 2023



101



1

 Tajammal Maqbool

Top 10 easy and very useful animations using CSS

Creating animations using CSS can add interactivity and visual appeal to your web...

4 min read · Oct 29, 2023



29

 Paul Brzeski

Building A Cool 3D website: Tips, Tricks and Lessons Learned

I first learned to make websites back in 1999 using Macromedia software like...

11 min read · Oct 27, 2023



5



See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)