

# UTILIZATION OF THREE.JS AND UNITY TO MAKE A WEBGL API TO PERFORM 3D IMMERSIVENESS IN WEB

Prankush Giri, Sayanabha Chandra, Subhrajyoti Chakraborty and Panchali Datta Choudhury

University of Engineering and Management, Kolkata

## Abstract

Hats off to Prometheus, who stole fire from heaven. From fire to WebGL, it's a long journey. But in today's world, WebGL is said to be the world's greatest invention. This paper is on the inquiry of the Web3D visualization methods in WEB. Three.js is a perfect 3D graphics engine out of many WebGL frameworks that has a lot of built-in lights and materials. Meanwhile, it provides many convenient functions for this study. In addition, this research can be applied to the customization of LED lights and lanterns, interior design, etc. This article introduces a kind of Web3D technology along with Unity 3D engine, that combines great advantages in GPU acceleration in fields of game designing and visual effects. The basic scene setup is a bit different from Three JS and Babylon JS because Unity has its own editor. All the basic things like creating a scene, adding mesh and lights can be done directly from the editor without coding a single line. This is a very basic Unity feature and anyone with a little experience with the program can easily do this by adding the necessary objects to the scene from the Unity menus. Unlike Three.JS and Babylon.JS, Unity does not require the creation of a rendering loop, which is done automatically by the game engine.

## Keywords:

three.js, Immersiveness, augmented reality, virtual reality, unity 3d engine.

## Introduction

Three.js is a 3D JavaScript library that allows developers to create 3D environments for the web. It works with WebGL, but we can also do it with SVG and CSS. WebGL is a JavaScript API that renders triangles on the canvas at remarkable speed. It is compatible with most modern browsers and is fast

because it uses the graphics processing unit (GPU) of the visitor.

For the sake of the tutorial, WebGL can draw more than triangles and can also be used to create 2D experiences, but we'll focus on 3D experiences using triangles for the sake of understanding.

A GPU can perform thousands of parallel calculations. Imagine we want to render a 3D model and that model consists of 1000 triangles - which, when I think about it, isn't that many. Each triangle contains three points. When we want to render our model, the GPU will have to calculate the position of these 3000 points.

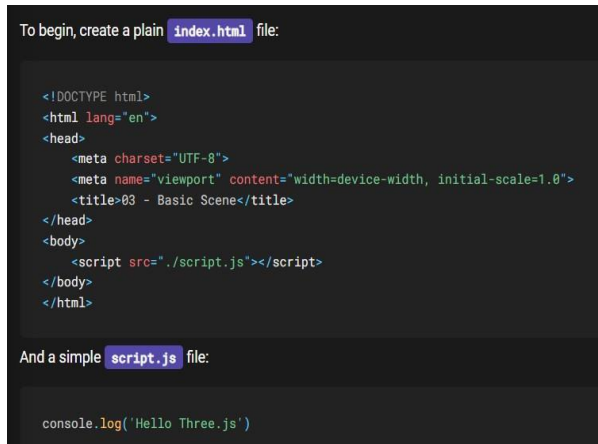
Once the model points are well positioned, the GPU must draw every visible pixel of these triangles. Again, the GPU can handle thousands and thousands of pixel calculations at once.

The instructions for placing points and drawing pixels are written in what we call shaders. Shaders are hard to master. We also need to provide some data to these shaders. For example: how to place points according to model transformations and camera properties. These are called matrices. We also need to provide data to help colour the pixels. If there is a light and the triangle is pointing towards that light, it should be brighter than if the triangle is not.

And that's why native WebGL is so hard. Drawing one triangle on the canvas would take at least 100 lines of code.

However, native WebGL benefits from existing at a low level, very close to the GPU. This allows for superior optimization and greater control.

## SETTING UP A SCENE



```
To begin, create a plain index.html file:
```

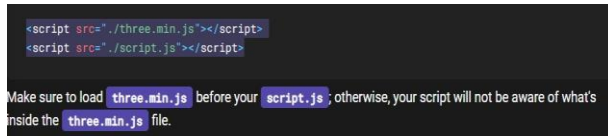
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>03 - Basic Scene</title>
</head>
<body>
  <script src="./script.js"></script>
</body>
</html>
```

And a simple `script.js` file:

```
console.log('Hello Three.js')
```

**Fig: 1. Setting up a scene**

Now we need to load the Three.js library.



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

Make sure to load `three.min.js` before your `script.js`, otherwise, your script will not be aware of what's inside the `three.min.js` file.

**Fig: 2. Loading the three.js library**

WE NEED A CAMERA RENDERER AND THE DOM ELEMENT CANVAS

## ANIMATIONS

We created a scene that we rendered once at the end of our code. That is already good progress, but most of the time, you'll want to animate your creations.

Animations, when using Three.js, work like stop motion. You move the objects, and you do a render. Then you move the objects a little more, and you do another render. Etc. The more you move the objects between renders, the faster they'll appear to move.

## PHYSICS

Physics can be one of the coolest features you can add to a WebGL experience. People enjoy playing with objects, see them collide, collapse, fall and bounce. There are many ways of adding physics to your project, and it depends on what you want to achieve. You can create your own physics with some mathematics and solutions like Raycaster, but if you wish to get realistic physics with tension, friction, bouncing, constraints, pivots, etc. and all that in 3D

space, you better use a library CANNON JS , OMIO JS ,AMMO JS(An additional library for three js).

## SHADERS

A shader is, in fact, one of the main components of WebGL. If we had started WebGL without Three.js, it would have been one of the first things we would have to learn, and this is why native WebGL is so hard.

A shader is a program written in GLSL that is sent to the GPU. They are used to position each vertex of a geometry and to colourize each visible pixel of that geometry. The term "pixel" isn't accurate because each point in the render doesn't necessarily match each pixel of the screen and this is why we prefer to use the term "fragment" so don't be surprised if you see both terms.

Then we send a lot of data to the shader such as the vertex coordinates, the mesh transformation, information about the camera and its field of view, parameters like the colour, the textures, the lights, the fog, etc. The GPU then processes all of this data following the shader instructions, and our geometry appears in the render.

There are two types of shaders, and we need both of them.

### Vertex shader

The vertex shader's purpose is to position the vertices of the geometry. The idea is to send the vertices positions, the mesh transformations (like its position, rotation, and scale), the camera information (like its position, rotation, and field of view).

### Fragment shader

The fragment shader purpose is to colour each visible fragment of the geometry.

The same fragment shader will be used for every visible fragment of the geometry. We can send data to it like a color by using uniforms —just like the vertex shader, or we can send data from the vertex shader to the fragment shader. We call this type of data —the one that comes from the vertex shader to the fragment shader— varying. We will get back to this later.

## IMPORTING OWN MODEL

When you do a render in a 3D software like Blender, it usually looks better than the model you import into Three.js, no matter how hard you try to get the exact same lighting and colors. This is because of the technique used while making the render.

We can start by loading the model. We are going to keep the cube in the scene to make sure that everything is working. Once we can see our portal scene, we will get rid of the cube.

Load the model after the loaders part and test the result in the console.

## SETUP

### Obtaining the Library

The Three.js project is hosted on GitHub at <https://github.com/mrdoob/three.js>. The latest release can be downloaded from <https://github.com/mrdoob/three.js/downloads>. Or if you are familiar with git, you can clone the repository:

git clone: <https://github.com/mrdoob/three.js.git>.

The library is under active development, and changes to the API are not uncommon. The latest complete API documentation can be found at the URL [mrdoob.github.com/three.js/docs/latest/](https://github.com/mrdoob/three.js/docs/latest/), which will redirect to the current version. There is a wiki page at <https://github.com/mrdoob/three.js/wiki/>, and there is no shortage of demos that use Three.js or articles about Three.js development on the Web. Some of the better articles are listed in Appendix D.

### Directory Structure

Once you download or clone the repository, you can place the files within your active development folder. The directory structure shows the following folder layout:

- /build compressed versions of the source files
- /docs API documentation
- /examples examples
- /gui a drag-and-drop GUI builder that exports Three.js source
- /src source code, including the central Three.js file
- /utils utility scripts such as exporters

Within the src directory, components are split up nicely into the following subfolders:

- /src
- /cameras camera objects
- /core core functionality such as colour, vertex, face, vector, matrix, math definitions, and so on
- /extra utilities, helper methods, built-in effects, functionality, and plugins
- /lights light objects
- /materials mesh and particle material objects such as Lambert and Phong
- /objects physical objects

## Conclusion

Hence from this paper we conclude that the three.js library is a really useful library which can be used for video game development using Unity or any other game engine. It is really easy to use and we have shown several practices of how-to setup the scenes in different game engines using WebGL component.

## Bibliography

1. Anttila, J., 2017. Creating room designer proof of concept with Three JS.
2. Semerikov, S., Mintii, M. and Mintii, I., 2021. Review of the course “Development of Virtual and Augmented Reality Software” for STEM teachers: implementation results and improvement potentials. CEUR Workshop Proceedings.
3. Frisk, D., 2016. WebGL: baserad ramverk prestandajämförelse: Mellan Three. Js och Babylon. Js.
4. Dirksen, J., 2015. Learning Three.js—the JavaScript 3D Library for WebGL. Packt Publishing Ltd.
5. Ververidis, D., Chantas, G., Migkotzidis, P., Anastasovitis, E., Papazoglou-Chalikias, A., Nikolaidis, E., Nikolopoulos, S., Kompatsiaris, I., Mavromanolakis, G., Thomsen, L.E. and Liapis, A., 2018, September. An authoring tool for educators to make virtual labs. In International Conference on Interactive Collaborative Learning (pp. 653-666). Springer, Cham.

6. Danchilla, B., 2012. Three.js framework. In *Beginning WebGL for HTML5* (pp. 173-203). Apress, Berkeley, CA.
7. Angel, E. and Haines, E., 2017. An interactive introduction to WebGL and three.js. In *ACM SIGGRAPH 2017 Courses* (pp. 1-95).
8. Rego, N. and Koes, D., 2015. 3Dmol.js: molecular visualization with WebGL. *Bioinformatics*, 31(8), pp.1322-1324.
9. Parisi, T., 2012. *WebGL: up and running*. "O'Reilly Media, Inc."
10. Dirksen, J., 2018. *Learn Three.js: Programming 3D animations and visualizations for the web with HTML5 and WebGL*. Packt Publishing Ltd.