Rendering interactive 3D scene as a web-page using Three.js, WebGL, and Blender

by

Sudeepto Bose
18241016

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
May 2022

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Sudeepto Bose

Sudeepto Bose
18241016

Approval

The project titled "Rendering interactive 3D scene as a web-page using Three.js, WebGL, and Blender" submitted by

1. Sudeepto Bose(18241016)

Of Spring, 2022 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on May 22, 2022.

Examining Committee:

Supervisor:
(Member)

Md. Golam Rabiul Alam, PhD
Associate Professor
Department of Computer Science and Engineering
BRAC University

Program Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Associate Professor
Department of Computer Science and Engineering
BRAC University

Head of Department:
(Chair)

Sadia Hamid Kazi
Associate Professor
Department of Computer Science and Engineering
BRAC University

Abstract

Building websites are the most sought skill in the tech world today. It has a low barrier of entry and is relatively easier to learn. Therefore, there is an abundance of web developers in our current market. Unfortunately, in Bangladesh, there are very few frontend developers and research show that most website developers become backend developers. The demand for frontend developers is ever increasing as users require a more immersive and new experience, therefore I decided to pursue it, but not in the traditional 2D way where it has only stylized HTML elements. I wanted to make immersive 3D websites that mimic the feel of video games. To do this, I had to learn and utilize the necessary 3D tools, Three.js, WebGL, and Blender. Native WebGL is very low level therefore to produce proper results I used the power of Three.js which is a graphics library that helps with rendering 3D rendering and many other features. Working with Three.js means working at the intersection of computer graphics and web development. The library is open source and freely available for anyone to use. Use cases for this project can be as simple as a product viewer for businesses or can be complex simulations that showcase the trajectory of a spaceship flying into orbit. For my project, I aimed to render a custom scene I created using Blender and display it as a web page for my fellow students to view. With the help of shaders, I wanted to further improve the viewing experience. Having my scene on a web page also means I can freely explore my artistic side and display it in my portfolio.

Keywords: Three.js, renderer, scene, WebGL, mesh, shaders, vertex, fragment, particles

Dedication

I want dedicate all my efforts and struggles of the educational life to my dear father. Without him I would not have come this far in life. I owe him everything.

Acknowledgement

All praise to my father for supporting me during my undergraduate journey. Also to my family who have been very patient with me throughout.
Secondly, to my supervisor Md. Golam Rabiul Alam, PhD for his kind support and advice in my work. He encouraged me to explore this field as much as possible.

Table of Contents

List of Figures

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

FOV Field of View

$GLTF$ Graphics Language Transmission Format

GPU Graphic Processing Unit

HDR High Dynamic Range

$JSON$ JavaScript Object Notation

PBR Physically Based Rendering

$RBGA$ Red, Green. Blue, Alpha

$sRGB$ Standard Red Green Blue

Chapter 1

Introduction

1.1 Background

The front-end of websites is commonly 2D in nature and some rarely have interactive 3D elements to them. This field of utilizing 3D elements for websites is not entirely popular, but the option of this technology's existence encourages artistic exploration. WebGL and the Three.js library enable developers to create 3D immersive experiences for the web. To put it simply, WebGL is the OpenGL equivalent for the web. WebGL is a JavaScript API that renders triangles in a canvas at a remarkable speed. It's compatible with most modern browsers, and it's fast because it uses the Graphic Processing Unit(GPU) of the visitor. WebGL can draw more than triangles and can also be used to create 2D experiences, but it is better suited for 3D experiences. Unfortunately, WebGL is a low-level solution that poses problems such as high code density, production time, etc. [1]

1.2 Aims and Objectives

The aim is to create a 3D scene which can be used as a web-page. It should have interactive elements so that the user feels in control of the experience.

1.3 Methodoly

The final project here took many trials to complete. The first was to understand the basics of JavaScript and then learn how 3D operates on the web. Three.js offers many features, and with the help of other libraries, features such as animations, physics, etc can be added. After finishing these steps, I created 'hello world' applications with Three.js to get a visual understanding of 3D. The first application had 3 cubes floating in 3D space, each having a rotation animation of its own. That application was published on Github Page. The second application was more fleshed out. The had features for interactivity and was also responsive for both phone and desktop. For interactivity, there was zooming in and out, scrolling up and down, using left-click and drag to rotate the camera, right-click and drag to displace the camera, double click for full screen and it had a controls tab on the top right which allowed the user to change some properties of the world and materials. I moved on to modeling custom scenes using the 3D software - Blender. Lastly, I created a

project which consisted of a custom 3D model I made with Blender, "Fireflies" using Three.js Particles and Shaders, and "Water" and "One way Portal" using Perin Noise Shaders.

Chapter 2

Three.js

2.1 Introduction

The Three.js module contains many classes and variables, but the most important variable is the THREE variable. It contains most of the necessary classes required to properly instantiate Three.js objects such as a scene and renderer. In addition to the Three.js module, we also require an HTML file where render is displayed onto an HTML canvas tag. To display 3D graphics and effects on web browsers,Three.js requires 4 main components

- Scene
- Camera
- Renderer
- Elements that need to be displayed

To elaborate on these components, a scene is like a container where the elements such as objects, lights, and particles are placed. The camera is the one that views objects within the scene and the renderer is the one that renders and displays what the camera is seeing within the scene. All this is displayed on the HTML canvas using a renderer, more specifically the WebGLRenderer. [4]
However, without objects(referred to as meshes), there will be nothing visible on the webpage; Only a space with a dark background will be displayed. The dark background is the 3D space within which meshes and other objects are added to populate the area. In order to populate the space, meshes need to be constructed by getting the coordinates of the vertices and adding colors and other extra features to them. Finally combining the two into a single mesh object. The technical and proper way to refer to this process of creating objects is by constructing geometry and adding material to it, then combining the two into a single mesh. The mesh is then added to the scene, and the renderer renders the scene into the HTML canvas. There are multiple cameras and renderers within the Three.js, but for my project, I used perspectiveCamera and WebGLRenderer. The Three.js components will be further explained below and with how I used them along with code and images.

2.2 Scene

The first step is to create a scene where everything will be added. To create a Three.js scene you need to call the scene class within the THREE variable and store it in a different variable, usually called scene. This variable scene will now be able to store all objects, cameras and other extra elements. Three.js scenes have extra properties such as environment, background, fog, etc. For my project, I utilized the fog property which is explained within the dedicated section.

2.3 Camera

We will require something to view the scene and the objects contained. The camera is the viewing tool that allows the renderer to display objects within the scene. It is an abstract class that inherits common properties and methods from the Object3D class. The camera itself is not viewable. When rendering the scene, the elements displayed are from the camera's point of view. Usually, one camera is enough, but multiple cameras can be added to perform transitions from one camera's point of view to another. Three.js has many camera classes such as StereoCamera which renders the scene through two cameras that mimic the eyes in order to create what's called a parallax effect that will lure the brain into thinking that is depth. These cameras are used in Virtual Reality applications. Another camera class is the OrthographicCamera class. It is used to create orthographic renders of a scene without perspective, meaning elements will have the same size on the screen regardless of their distance from the camera. Real-time strategy(RTS) games such as Age of Empire used this sort of camera. For my project, I used a PerspectiveCamera class from the Three.js library. The PerspectiveCamera class is specially designed to simulate a real-life camera with a perspective like the human eye. It is also one of the most common cameras used in 3D projects. The class needs two essential parameters, the field of view and the aspect ratio. The field of view is the angle of vision the camera will have. When a high value is provided, more of the scene can be viewed but there will be a lot of distortion. Lower values will create the effect of zooming in. The field of view (or FOV) values are expressed in degrees and correspond to the vertical vision angle. The aspect ratio is usually the width of the display divided by the height.
Three.js cameras have controls that enable them to rotate, displace, zoom, etc. Instead of manually controlling the way the camera operates, I used Three.js's built-in controls. There are many of these controls such as FlyControls which enable moving the camera like on an aircraft. It can rotate the camera on all 3 axes and can go forward and backward. For my project I used OrbitControls. It enables the camera to rotate using the left mouse while there is a focus on a point. Lateral translation of the camera's position can be done with the right mouse and it can also zoom in and out using the mouse wheel. The OrbitControls need to be imported from the examples folder within the Three.js module as it is not available within the THREE variable. The controls require two parameters, the camera, and the canvas. The rotation provided by this class is instant but enabling the damping property will create a smoother and more enjoyable effect when rotating the camera.
After instantiating the camera, I moved the position to a specific angle. The values for the position property are arbitrary. For the controls, I enabled the damping

```
/**
 * Camera
 */
// Base camera
const camera = new THREE.PerspectiveCamera(45, sizes.width / sizes.height, 0.1, 100)
camera.position.x = 4
camera.position.y = 2
camera.position.z = 4
scene.add(camera)

// Controls
const controls = new OrbitControls(camera, canvas)
controls.enableDamping = true
controls.maxDistance = 10
controls.maxPolarAngle = Math.PI * 0.4
```

Figure 2.1: Code for camera and additional properties in the project

for smooth rotation and added some restrictions so that the viewer does not see elements that are not meant to be seen. The maxDistance property restricts the user from zooming out of the limit, and the maxPolarAngle prevents users from going below the plane of the scene.

2.4 Renderer

The renderer renders the scene from the camera's point of view and the result is drawn into the canvas. For my project, I used the WebGLRenderer. There are many different types of renderers but the WebGLRenderer is the standard renderer most Three.js projects use. The renderer object is the one that requires the HTML element into which it will display the results.

```
/**
 * Renderer
 */
const renderer = new THREE.WebGLRenderer({
    canvas: canvas,
    antialias: true
})
renderer.setSize(sizes.width, sizes.height)
renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
renderer.outputEncoding = THREE.sRGBEncoding
```

Figure 2.2: Code for the renderer

The WebGLRenderer takes the HTML canvas as input since it needs to target for rendering the scene. I enabled the anti-aliasing because I wanted my scene to remain smooth and without jagged edges. The setSize and setPixelRatio methods are used for responsive rendering. The outputEncoding property is used for accurate color management of the scene rendered.

2.5 Responsiveness

For immersive experiences, it is best that the website is responsive to all displays therefore it is a good idea to make sure the 3D space takes the whole screen into account. The HTML canvas is the element that takes all the space available and it should also fit the user's window when a resize occurs.
To fit the viewport, the innerWidth and innerHeigth of the window need to be stored

in an object. This ensures that whichever device the user is using, the window size will always be available to the renderer. But there are default settings of browsers that take away from the proper size, such as a white margin and a scroll bar. Browsers all have default stylings like more significant titles, underlined links, spaces between paragraphs, and paddings on the page. To fix this issue I used CSS to remove the stylings. It is important to make the overflow property hidden from the body element.
In order to handle the resizing, we need to listen to events that trigger the 'resize' property of browsers. When the resize occurs we store the new values into a 'sizes' object. We also need to update the camera's aspect ratio by changing the aspect property. Cameras have an initial projection of the meshes present in the scene, but when a resize takes place we also need to update the projection of the camera by calling its updateProjectionMatrx(). Finally, we need to update the renderer's size by passing in the new values from the sizes object.

```
208  const sizes = {
209      width: window.innerWidth,
210      height: window.innerHeight
211  }
212
213  window.addEventListener('resize', () =>
214  {
215      // Update sizes
216      sizes.width = window.innerWidth
217      sizes.height = window.innerHeight
218
219      // Update camera
220      camera.aspect = sizes.width / sizes.height
221      camera.updateProjectionMatrix()
222
223      // Update renderer
224      renderer.setSize(sizes.width, sizes.height)
225      renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
226
227      // Update fireflies
228      firefliesMaterial.uniforms.uPixelRatio.value = Math.min(window.devicePixelRatio, 2)
229
230  })
```

Figure 2.3: Code for the responsiveness

Some extra matter to handle is the pixel ratio. Depending on the device, the render may seem blurry and there may be artifacts shaped like stairs on the edges called aliasing. This occurs when the pixel ratio of the device is greater than 1. The pixel ratio corresponds to how many physical pixels you have on the screen for one pixel unit on the software part. A few years back all screens had a pixel ratio of 1, but that caused limitations for how precise images were and how thin fonts could be. Apple saw an opportunity to construct screens with a pixel ratio of 2 called the retina, and predictably many constructors are doing it with even higher pixel ratios than 2. While this is a good thing for the image quality, a pixel ratio of 2 means 4 times more pixels to render and a pixel ratio of 3 means 9 times more pixels to render. This puts a huge load on a device's GPU because of the increased number of calculations required to render out the scene. To get the screen pixel ratio I used the window.devicePixelRatio method within the browser and used that to update the pixel ratio of the renderer by calling its method setPixelRatio.

2.6 Mesh

Meshes are the objects used to populate the scene to create wonderful effects. Creating meshes, requires two major components, geometry, and material.

2.6.1 Geometry

In Three.js components are composed of vertices(point coordinates in 3D spaces) and faces(triangles that join those vertices to create a surface). These geometries are used to form meshes but it is also used to make particles. Three.js has many built-in geometries such as SphereGeometry, PlaneGeometry, CylinderGeometry, etc. These classes have built-in methods like translate(), rotateX(), normalize(), etc. There are also ways to create custom geometries if the built-in geometries do not suffice. For the basic scene, I used BoxGeometry to create a cube. Taking in the parameters for width, height, depth, widthSegments, heightSegments, depthSegments. This cube is for testing purposes to see if the scene has loaded in properly. The cube will be later removed when updating the scene with the model.

2.6.2 Material

Materials are used to put a color on each visible pixel of the geometries. The algorithms that decide on the color of each pixel are written in programs called shaders. Writing shaders is one of the challenging parts of WebGL and Three.js. Thankfully Three.js has many built-in materials with premade shaders. I also wrote custom shaders for particles which are explained later below.

2.6.3 Combining geometry and material

To finalize these two classes are needed to construct meshes.

```
36  /**
37   * Object
38   */
39  const cube = new THREE.Mesh(
40      new THREE.BoxGeometry(1, 1, 1),
41      new THREE.MeshBasicMaterial()
42  )
43
44  scene.add(cube)
```

Figure 2.4: Code for constructing a mesh using Three.js built-in Geometry and Material

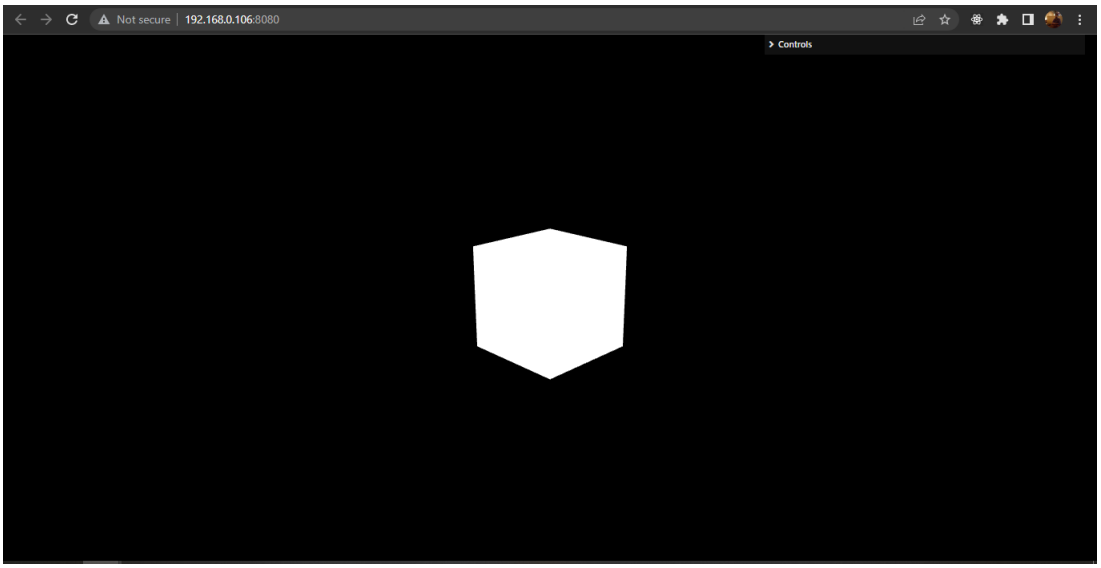

Figure 2.5: Default cube added to the scene

2.7 Animation

The animation is done manually with a custom JavaScript function. Animation, when using Three.js is like stop motion, the object moves and there is a new render. The cycle of objects moving and displaying a new render is the basis of how the animation works. Screens have a usual frame rate which is 60 frames per second(fps) but there are limitations depending on the device. Some screens can run much faster and when devices have difficulties processing things, they run slower. To avoid such issues I used a native JavaScript method called window.requestAnimationFrame(). This function executes the function provided on the next frame, basically a recursive function. The tick function is an infinite loop that keeps calling itself. In order to ensure that all devices have consistent frame rates regardless of device, I used a timestamp to calculate the previous time and the current time on the frame after to get the difference in time. The difference is used to establish a consistent framerate in all devices. While the code for this is simple, Three.js provides a simpler solution, the Clock class. This Clock class has a built-in method called getElapsedTime() which does exactly as stated above.

```
286  const clock = new THREE.Clock()
287
288  const tick = () =>
289  {
290      const elapsedTime = clock.getElapsedTime()
291
292      // Update material
293      fireflies.material.uniforms.uTime.value = elapsedTime
294      portalLightMaterial.uniforms.uTime.value = elapsedTime
295      waterMaterial.uniforms.uTime.value = elapsedTime
296
297      // Update controls
298      controls.update()
299
300      // Render
301      renderer.render(scene, camera)
302
303      // Call tick again on the next frame
304      window.requestAnimationFrame(tick)
305  }
306
307  tick()
```

Figure 2.6: Code to enable animations

The code shows how the elaspedTime is being used with Shaders. As a new render occurs every frame, we need to remind the controls of the perspective camera to update itself, otherwise, it would be stuck in its place.

2.8 Webpack

A bundler is a tool in which assets such as JavaScript, CSS, HTML, images, TypeScript, Stylus, and other languages. The bundler will handle those assets, apply potential modifications, and output a 'bundle' composed of web-friendly files like HTML, CSS, images, and JavaScript.
Webpack is currently the most popular bundle. It can handle the needs and provide extensive documentation and a constructive community. The following codes are used to utilize Webpack.

- npm install - installs necessary node modules using the package.json reference
- npm run dev - to start the server
- Ctrl+C - for terminating the batch job
- npm run build - deployed build and output it in the /dist/ folder

The build command is later used during the hosting and deployment process.

Chapter 3

Blender - Modeling and Exporting

3.1 Introduction

There are many software applications like Cinema4D, Maya, 3DS Max, Blender, ZBrush, Marmoset Toolabg, Substance Painter, etc. These are great and they differ on diverse criteria like the UX, the performance, the features, the compatibility, the price, etc.
Blender is a free open-source tool that enables users to create 3D experiences. The barrier of entry for this software is comparatively low and it serves the purpose of this project very well. The splash screen gives access to some useful links, templates, and recently opened files. Different parts of the interface are called areas. Areas are very flexible, and it allows users to create any layout they need. The main time spent is one the 3D Viewports where the modeling happens
Some key feature required for the model creation was to control the movement of objects within the correct axes and how to morph them. Blender has a different orientation of the x,y, and z-axis than regular mathematical ideology. The z-axis is the one that exists on the vertical plane whereas the x and y-axis both exist on the horizontal plane. The shortcut key 'G' is used to displace objects and when the key X, Y, or Z is pressed then the object will move along in that axis only. Similarly, the shortcut key 'S' is used to scale objects.
By default, Blender is on Object Mode, where users can create, delete, and transform objects. There are many other modes but for my project, only the Edit Mode was required. It is similar to Object Mode but we can edit the vertices, the edges, and the faces.
Blender has 3 main render engines

- Eevee - A real-time render engine. It uses the GPU just like Three.js, it's very performant, but it has limitations like realism, light bounce, reflection, and refraction.
- Workbench - A legacy render engine is not used anymore. It has good performance but the results are not very realistic.
- Cycles - A raytracing engine. It's very realistic. It handles light bounce, deep reflection, deep refraction, and many other features, but it is very sluggish and might cause the user to wait for hours or even days to render the scene.

By default, the render engine is Eevee but for my project, I went with the Cycles rendering engine to create accurate textures for my model. By pressing F12 the software opens a new window and produces a render. The next step is to create a model and bake the textures of the render into an HDR file and later pass it through the compositor to get a web-friendly texture which will be used to create the scene on the web.

3.2 Modeling

To create the baked scene, we need to go through multiple steps:

- Create the scene in 3D software.
- Optimize all the objects because we need clean geometries and only the surfaces that can be seen.
- UV unwrap everything.
- Bake the render into the texture.
- Export both the scene and the texture.
- Import everything in Three.js and apply the texture to the mesh.

I created a simple scene using Blender. It is a low poly model meaning there are very few vertices and structures. The scene is not complex, consists of trees, trunks, fences, pole lights, rocks, a house, a puddle, a portal, and a circular plane.

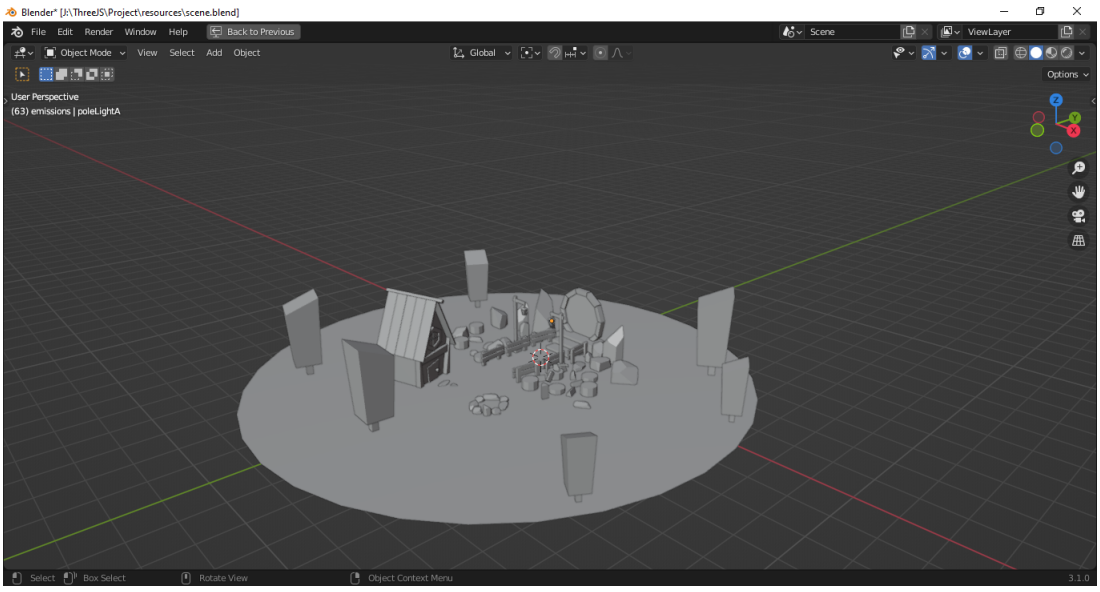

Figure 3.1: Low poly scene created using Blender

The materials used in the scene have a PBR property which stands for Physical Based Rendering. The PBR property is applied to all the objects except the lights. The portal, window, and pole lights have emissive material so that light emits and interacts with other surfaces. The light emissions and lights which are bounced from different surfaces are all occurring because of the PBR property.

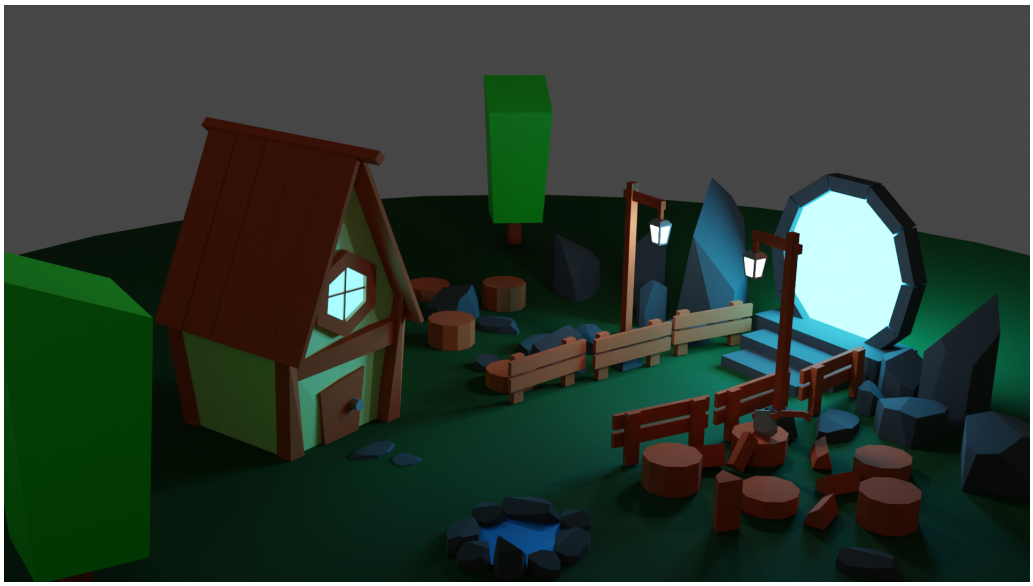

Figure 3.2: Final render of scene from Blender

3.3 Optimization

I had to optimize the scene in different ways so that the overall file was not heavy.

3.3.1 Removing faces from objects

I had to optimize the scene in different ways so that the overall file was not heavy. The first thing I needed to do is remove all the hidden faces. That would mean faces directly on the floor like the bottom of the trunks. And, it would also mean faces against other faces like the back of the stair steps.

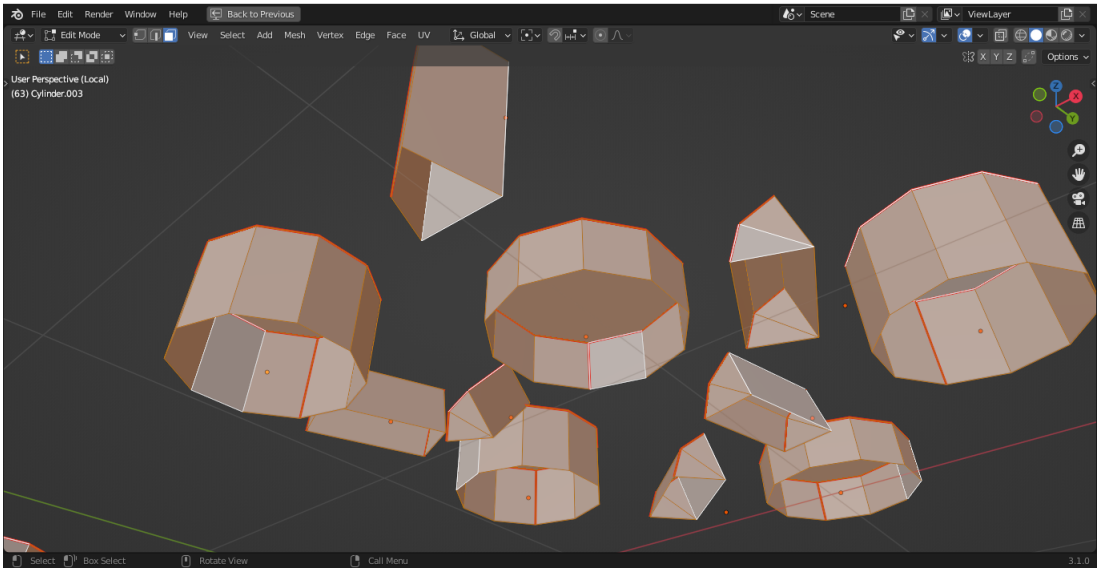

Figure 3.3: Optimizing scene by removing faces so that users will never see

The inside of these objects is hollow and the bottom faces are removed. There was no point in applying materials to these faces because they would never be seen in

the final scene. Faces that would be unseen from the house, portal, etc were all removed.

3.3.2 Face orientation

One issue we cannot see has to do with the orientation of faces. All faces have a front and a back. While this is not a problem when doing a render, it might create bugs during the baking process and some of the faces might become black. Those faces will not even be rendered in Three.js because, by default, the backsides are hidden.
This orientation is a matter of normals. When doing extrudes, insets, and other operations like that, we might have flipped the normals unintentionally.

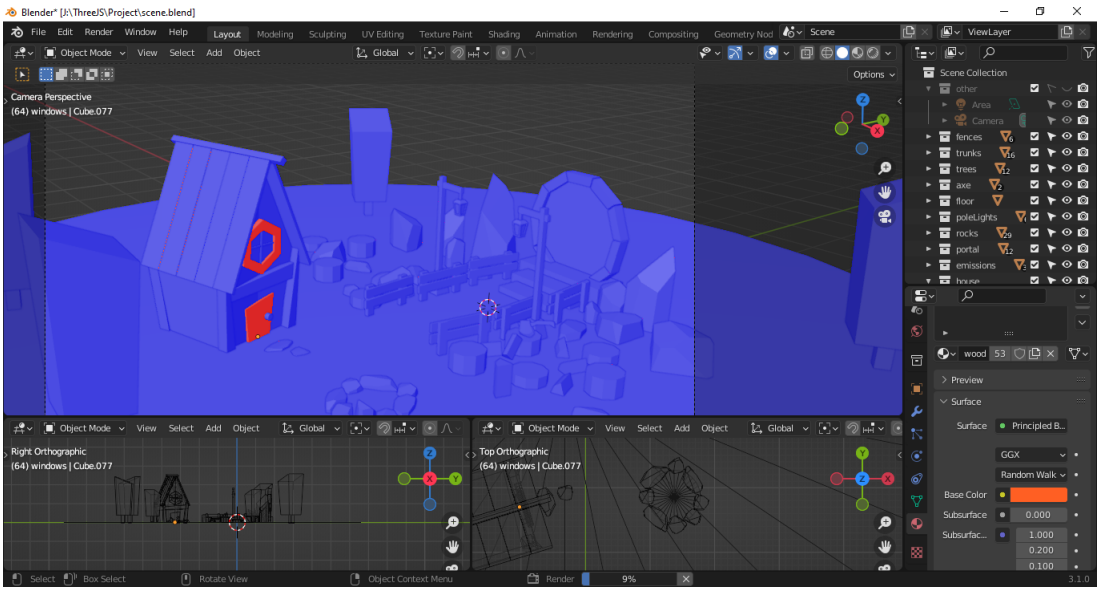

Figure 3.4: Face orientation - red faces indicate not oriented properly

3.3.3 UV unwrapping

This is more of an engineering approach rather than a creative one. The idea is to unfold all the geometry composing our scene into a square. Later, this will enable us to bake the render into a texture that will follow the unwrapping we set. There are multiple ways of unwrapping a geometry. We can use a "smart" automatic version that will do a pretty good job. Or, we can do a manual version to create a more optimized version, but at the price of more effort. I used both techniques depending on the objects. Only for the rocks, I used the "smart" unwrap feature in Blender because the rocks were irregular in shape. The rest was done manually by creating seams.
The bottom left shows the UV Editor window. Different colors indicate that the shapes are not in scale. Therefore, the goal is to scale all the unwraps to that they are of color.
I repeat the whole process for every object other than the pole lights, portal, and window on the house. Those objects have emissive properties; Therefore, they are not going to be included in the final texture. In the end, they are uniform colors

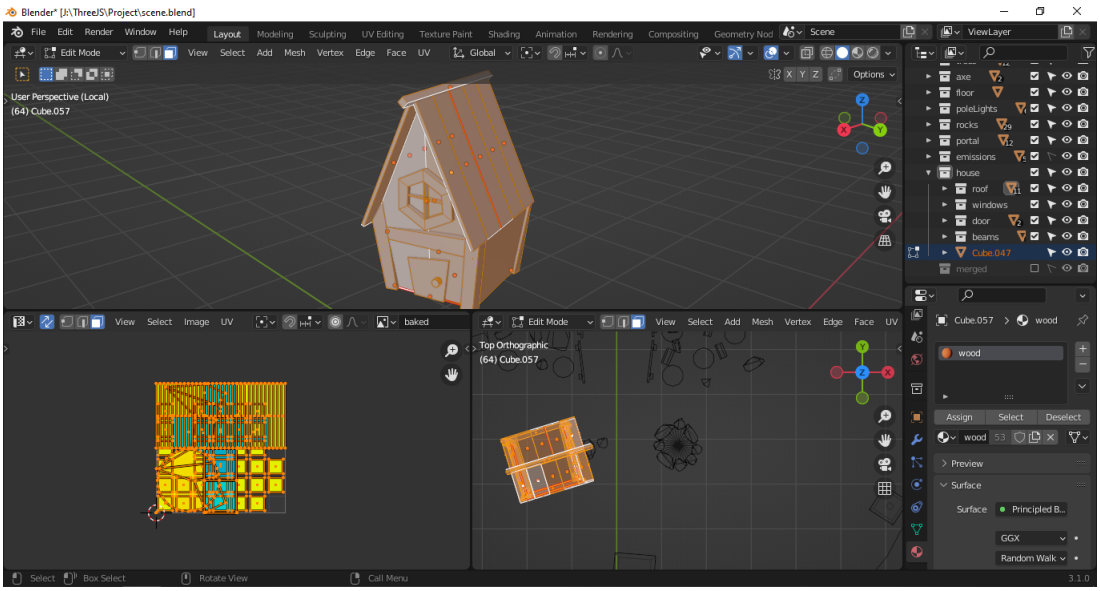

Figure 3.5: Manual unwrapping process of the house.

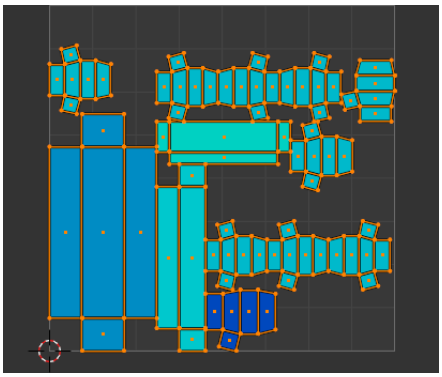

Figure 3.6: Unwrapping of portal rocks and stairs

and we are going to create materials within Three.js to give them those colors. I could have baked those objects, but it's a waste of space in the texture.
After unwrapping all the objects, they must be put into the UV window so that the baking can be applied.

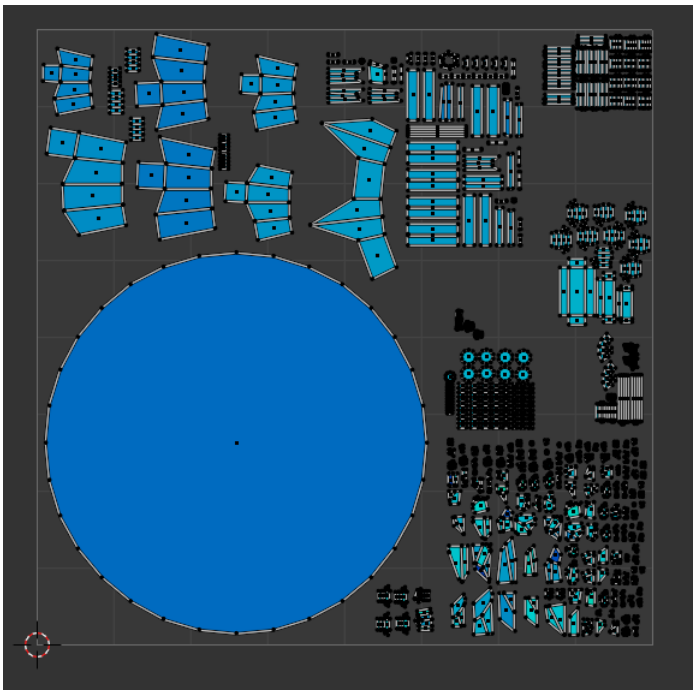

Figure 3.7: Final unwrap

3.4 Baking

When we do render in a 3D software like Blender, it usually looks better than the model imported into Three.js, no matter how much one tries to get the exact same lighting and colors. This is because of the technique used while making the render. Ray Tracing consists of casting multiple rays toward each pixel of the render. These rays start with the geometry we are rendering. Next, they test the direction of each light in the scene to see which part of the geometry is illuminated as well as to test whether the directions of the light bouncing off the geometry are colliding with other objects in the scene. Then, for each of these collisions, more rays are cast as they bounce off other objects. And it goes on and on like this multiple times. All of the information collected by these collisions is then computed to define the final color of that pixel.
The goal is to simulate real-life lighting and enable visual effects like indirect lighting and soft shadows. As an example, if you place a red object close to a white object, you'll see the white object being tinted with red because rays are bouncing from the red surface to the white surface. In the same way, you'll see that the red object looks brighter when the surface is close to the white object. This process results in a beautiful realistic render, but doing one render can take many minutes, even hours.
When we are doing renders with WebGL, we need to do it as fast as possible in order to get a good frame rate. We don't have the luxury to spend minutes on just one render. For this reason, rendering in WebGL uses cheaper techniques that do not look as good but at least keep a decent frame rate.
The idea of baking is that we save the Ray Tracing renders into textures that we then use in WebGL instead of using the classic render techniques provided by Three.js.There is no light, no real-time shadows. It's just the texture you see below

being placed on the geometries.
This way, we will see the Ray Tracing renders directly on the Meshes. And when we move around the scene, the performance will be great because all we did was display a texture on a geometry.
Unfortunately, there are some drawbacks. I had to bake everything in the 3D software and it was a long process. I had to load the textures and if the scene was a complex scene with a lot of objects, then I would need a lot of textures. This is bad for loading but can also result in a short freeze at the beginning of the experience because we need to load those textures into the GPU. The lights aren't dynamic. We can't move the lights, and we cannot change their intensity or color in real-time. We have to do it in the 3D software and re-bake everything.
Because we can have multiple textures in my project, we need a way to tell Blender that the texture we created is the one in which each object has to be baked. This information is provided on each material.

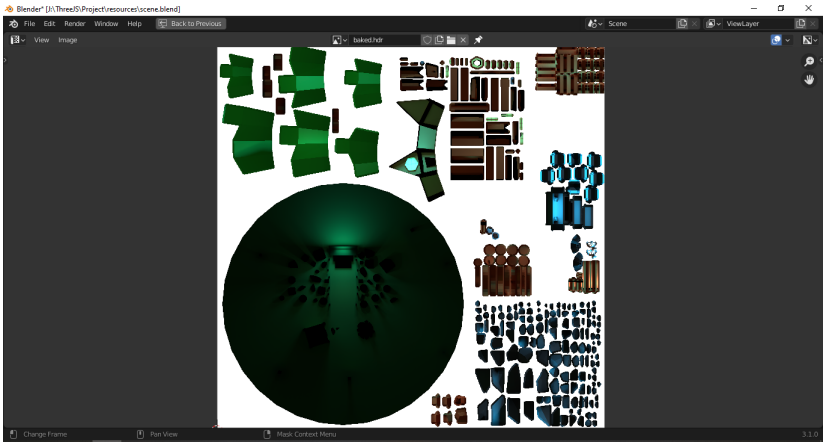

Figure 3.8: Textures saved into HDR file to preserve all the data

3.5 Saving and Exporting

Before baking anything in the texture, I had to save it as an HDR file. This is not the texture used in the project. When saving this file, I chose to check the 32-bit float. This made the texture data much more precise than a classic texture. In a way, we made an HDR texture. The problem with HDR textures is that they are too heavy. We are baking our scene inside an HDR texture in order to keep all the data.
As you can see, the colors are all burned out. If you compare it to the render, the colors will have less contrast. It is as if they are being toned in the render and that is exactly the case. When we do a render in Blender, a color manager named Filmic is used. This tells us that, when baking in Blender, we are losing Filmic.
Another problem is the visual noise. When we do a render and if we have checked the denoise parameter, we can see a smooth render without noise. As with Filmic, Blender seems to ignore the denoise parameter.
Finally, we want to export a compressed image that we can use in Three.js. We want it to be light and because we don't need transparency, we can use JPEG. Also, we want to apply the sRGB encoding to improve the color quality.

But currently, all we have is our fancy HDR image.
All of these problems can be fixed with the Blender Compositor. The idea, here, is to create some nodes that will take our texture, apply a denoise on it and then apply Filmic. Then we output that texture to the render by sending it to the Composite node.

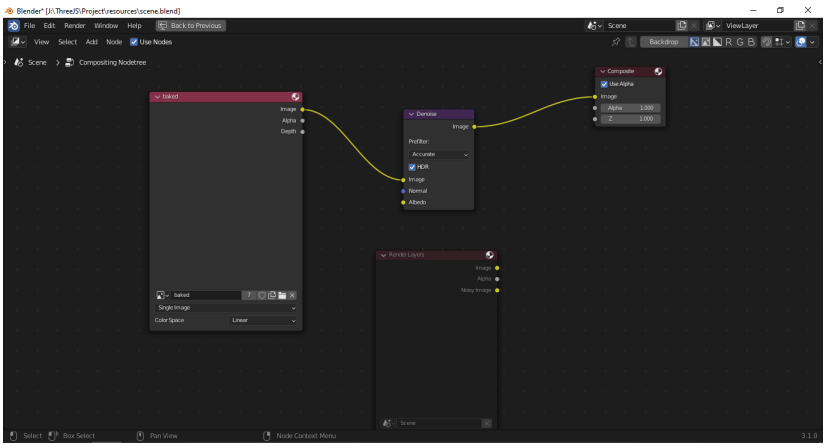

Figure 3.9: Compositor setup which allows conversion from HDR to JPEG

The HDR file was around 18MB and the JPEG file was 1MB, which is a huge difference in file size. This greatly improves the performance of the website. The time required to load the texture onto the geometry is reduced significantly.

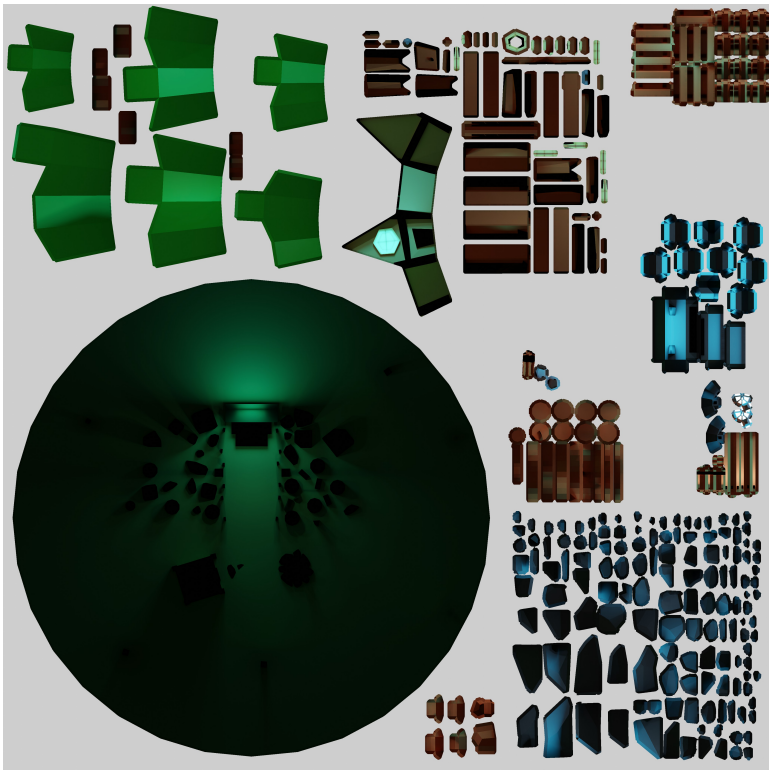

Figure 3.10: Compositor setup which allows conversion from HDR to JPEG

The blender file is exported as scene.glb file. This contains all the data points for the geometry. This compressed file needs to be decompressed when loading the

website, which will be explained below. Also, the materials used in Blender will not be carried into the GLTF file, because custom Three.js materials will be applied to them.
The file format in which the data points are stored is known as GLTF-binary. GLTF stands for GL Transmission Format. It's made by the Khronos Group (the people behind OpenGL, WebGL, Vulkan, Collada, and with many members like AMD / ATI, Nvidia, Apple, id Software, Google, Nintendo, etc.)
GLTF has become very popular these past few years. It supports very different sets of data. It can contain data like the geometries and the materials but it can also have data like cameras, lights, scene graph, animations, skeletons, morphing, and even multiple scenes. It also supports various file formats like JSON, binary, and embedded textures.
GLTF has become the standard when it comes to real-time. And because it's becoming a standard, most 3D software, game engines, and libraries support it. That means that we can easily have similar results in different environments.

Chapter 4

Importing model into Three.js

4.1 Draco compression

In my project, I used DRACO compression for better load time. The decoder is available in native JavaScript but also in Web Assembly (wasm), and it can run in a worker. Those two features significantly improve performances, but they imply having a wholly separate code.
While it may seem that the Draco compression is a win-win situation, it is not. Yes, the geometries are lighter, but first, we have to load the DRACOLoader class and the decoder. Secondly, it takes time and resources for the computer to decode a compressed file that can result in a short freeze at the start of the experience, even if we are using a worker and Web Assembly code. If we had one model with a 100kB geometry, then we would not need Draco.

```
38  /**
39   * Loaders
40   */
41  // Texture loader
42  const textureLoader = new THREE.TextureLoader()
43
44  // Draco loader
45  const dracoLoader = new DRACOLoader()
46  dracoLoader.setDecoderPath('draco/')
47
48  // GLTF loader
49  const gltfLoader = new GLTFLoader()
50  gltfLoader.setDRACOLoader(dracoLoader)
51
52  /**
53   * Textures
54   */
55  const bakedTexture = textureLoader.load('baked.jpg')
56  bakedTexture.flipY = false
57  bakedTexture.encoding = THREE.sRGBEncoding
58
```

Figure 4.1: Code for loaders in the project

The textureLoader is responsible for loading the baked.jpg file which contains the textures for the geometry. The GLTF loader loads in the scene.glb file. As you can see the bakedTexture.flipY is equal to false because the textures you see in the JPEG file were upside down. To apply the texture properly, the flipY is equal to false, and the color encoding is set to sRGBencoding for accuracy.
The above code shows the importing of the GLTF model. The GLTF file has many children and one of them is the gltf.scene. This scene contains the objects that were

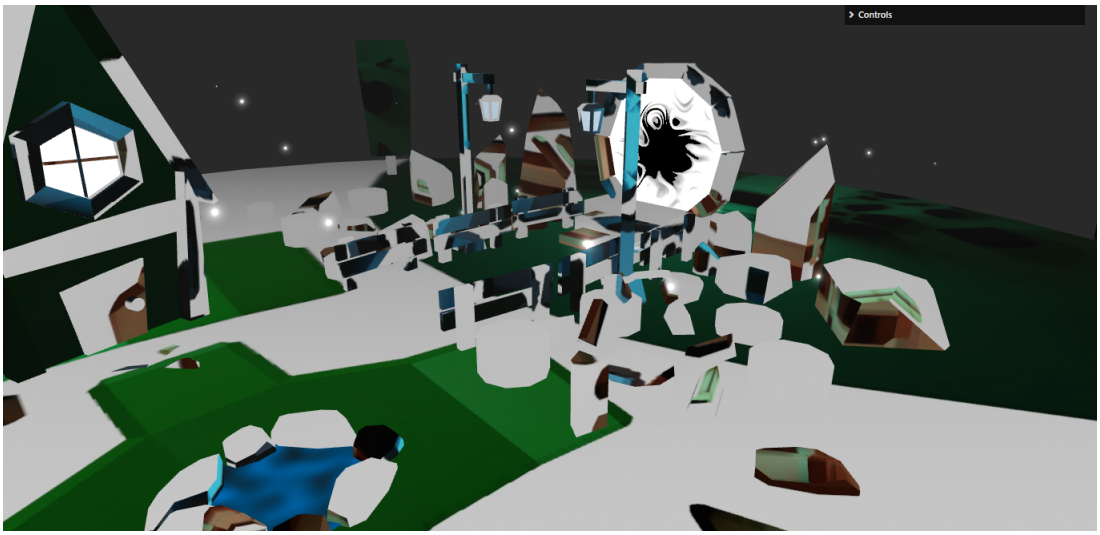

Figure 4.2: Texture loaded in where bakedTexture.flipY is equal to true

```
gltfLoader.load(
    'scene.glb',
    (gltf) =>
    {
        const bakedMesh = gltf.scene.children.find(child => child.name === 'baked')

        const portalLightMesh = gltf.scene.children.find(child => child.name === 'portalLight')
        const poleLightAMesh = gltf.scene.children.find(child => child.name === 'poleLightA')
        const poleLightBMesh = gltf.scene.children.find(child => child.name === 'poleLightB')
        const water = gltf.scene.children.find(child => child.name === 'water')
        const windowLight = gltf.scene.children.find(child => child.name === 'windowLight')

        bakedMesh.material = bakedMaterial

        poleLightAMesh.material = poleLightMaterial
        poleLightBMesh.material = poleLightMaterial
        portalLightMesh.material = portalLightMaterial
        water.material = waterMaterial
        windowLight.material = windowMaterial

        scene.add(gltf.scene)
    }
)
```

Figure 4.3: Code for loading the GLTF model.

designed in Blender. It is through this that we can access the objects. Custom shaders made with Three.js are applied to the objects which had emissive materials assigned to them and the texture that was exported is applied to the overall scene. This new gltf.scene is added to the Three.js scene.

4.2 Measuring GPU calls

Spector.js is a library used to monitor how many different calls are made to the GPU to produce the render. The value on the Command tab on the top right shows how many calls are made to the GPU. For this project, it made 237 calls during the rendering process. The goal is to reduce the number of calls made to the GPU. In order to do this, all the Blender geometries, except the emissive geometries, had to be merged to form 1 geometry.

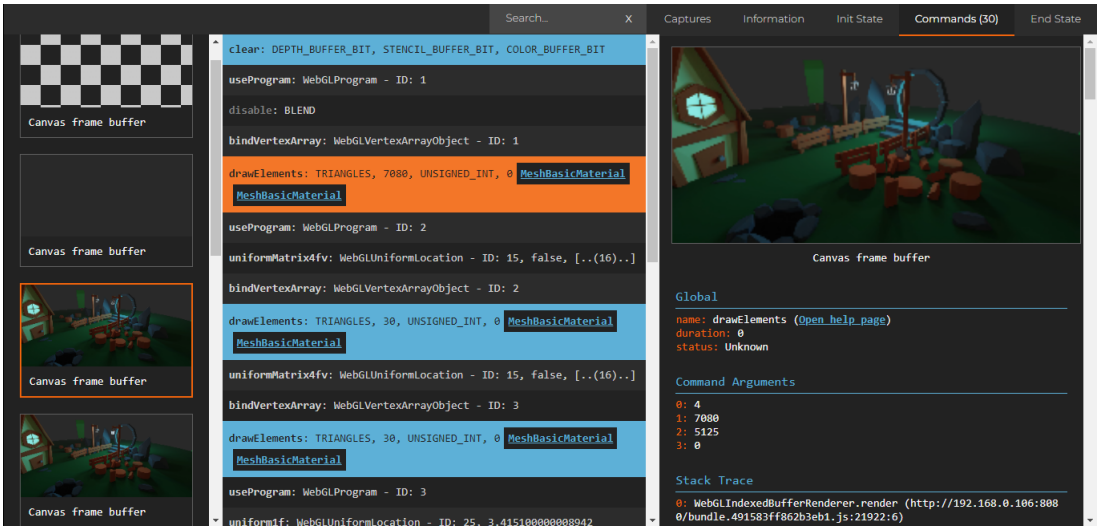

Figure 4.4: Code for loading the GLTF model.

The above figure shows that the number of calls made to the GPU decreased from 237 to 30. This significantly improved load times. Since everything object other than the emissive objects were merged, traversing for emissive objects within the scene is made shorter.

Chapter 5

Shaders

5.1 Introduction

A shader is, in fact, one of the main components of WebGL, but it is one of the main reasons 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 geometry and to colorize each visible pixel of that geometry. The term "pixel" is not accurate because each point in the render doesn't necessarily match each pixel of the screen and this is why the term "fragment" is used. Then we send a lot of data to the shader such as the vertices coordinates, the mesh transformation, information about the camera and its field of view, and parameters like the color, 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, one is the vertex shader and the other is the fragment shader. [3]

5.2 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), and the camera information (like its position, rotation, and field of view). Then, the GPU will follow the instructions in the vertex shader to process all of this information in order to project the vertices on a 2D space that will become the render.
When using a vertex shader, its code will be applied to every vertex of the geometry. But some data like the vertex position will change between each vertex. This type of data, the one that changes between vertices, is called an attribute. But some data does not need to switch between each vertex like the position of the mesh. Yes, the location of the mesh will impact all the vertices, but in the same way. This type of data, the one that doesn't change between vertices, is called a uniform. The vertex shader happens first. Once the vertices are placed, the GPU knows what pixels of the geometry is visible and can proceed to the fragment shader.
The `gl_Position` variable already exists in the vertex shader. We need to assign it. This variable will contain the position of the vertex on the screen. The goal of the instructions in the main function is to set this variable properly.
When setting the values, we do not truly move the plane in a 3D space as if we were playing with the position in Three.js. We only move the projected plane into a 2D

```glsl
void main()
{
    vec4 modelPosition = modelMatrix * vec4(position, 1.0);
    vec4 viewPosition = viewMatrix * modelPosition;
    vec4 projectionPosition = projectionMatrix * viewPosition;

    gl_Position = projectionPosition;
```

Figure 5.1: Code of vertex shader used in the window light of the scene.

space. We need 4 values for the `gl_Position` because its final goal is to position vertices on a 2D space. It's actually because of the coordinates or not precisely in 2D space; they are in what is called clip space which needs 4 dimensions.
Clip space is a space that goes in all 3 directions (x, y, and z) in a range from -1 to +1. It's like positioning everything in a 3D box. Anything out of this range will be "clipped" and disappear. The fourth value (w) is responsible for the perspective. All of this is done automatically. The same code applies to every vertex of the geometry. Attributes are the only variable that will change between the vertices. The same vertex shader will be applied for each vertex and the position attribute will contain the x, y, and z coordinates of that specific vertex. Then this is converted from vec3 to vec4.
Each matrix will transform the position until we get the final clip space coordinates. There are 3 matrices in our code, and because their values are the same for all the vertices of the geometry, we retrieve them by using uniforms.
Each matrix will do a part of the transformation:

- The modelMatrix will apply all transformations relative to the Mesh. If we scale, rotate, or move the Mesh, these transformations will be contained in the modelMatrix and applied to the position.
- The viewMatrix will apply transformations relative to the camera. If we rotate the camera to the left, the vertices should be on the right. If we move the camera in direction of the Mesh, the vertices should get bigger, etc.
- The projectionMatrix will finally transform our coordinates into the final clip space coordinates.

5.3 Fragment shader

The fragment shader's purpose is to color 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. The most straightforward instruction in a fragment shader can be to color all the fragments with the same color. We get the equivalent of the MeshBasicMaterial, if we had set only the color property. Or we can send more data to the shader, for instance, a light position. We can then color the fragments according to how much the face is in front of the light source. We would get the MeshPhongMaterial equivalent if we had one light in the scene.

```
1  uniform vec3 uColor;
2
3  void main()
4  {
5      gl_FragColor = vec4(uColor,1.0);
```

Figure 5.2: Code of fragment shader used in the window light of the scene.

The `gl_FragColor` is like the `gl_Position` but for the color. It's already declared, and we need to assign it in the main function. It's a vec4 with the first three values being the red, green, and blue channels (r, g, b) and the fourth value is the alpha (a).
Attributes are values that change between each vertex. We already have one attribute named position that contains a vec3 of the coordinates of each vertex.There is a way of sending data from the vertex shader to the fragment shader called varyings.
Uniforms are a way to send data from JavaScript to the shader. That can be valuable if we want to use the same shader but with different parameters, and it's also the occasion to have parameters that can change during the experience. We can use uniforms with both vertex and fragment shaders, and the data will be the same for every vertex and every fragment. We already have uniforms in our code with projectionMatrix, viewMatrix, and modelMatrix but we didn't create these because Three.js did that automatically. The fragment shader code has a Uniform being passed through the JavaScript which is stored in the variable uColor. uColor contains the RGB values which will be used to paint the window light in the scene. The figure below showcases the result.

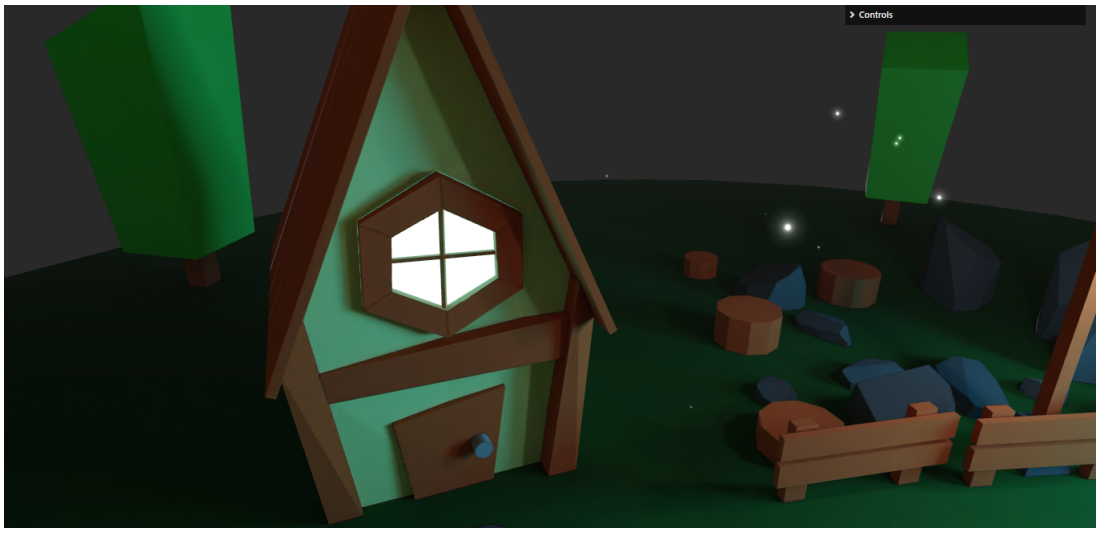

Figure 5.3: Mimicking the effect of light-emitting through the window using shaders.

5.4 Particles

Particles are precisely what you expect from that name. They are very popular and can be used to achieve various effects such as stars, smoke, rain, dust, fire, and

many other things. The good thing with particles is that we can have hundreds of thousands of them on screen with a reasonable frame rate. The downside is that each particle is composed of a plane (two triangles) always facing the camera. Creating particles is as simple as making a Mesh. We need a BufferGeometry, a material that can handle particles (PointsMaterial), and instead of producing a Mesh, we need to create a Points. Each vertex of the geometry will become a particle. We need a special type of material called PointsMaterial. This material has multiple properties specific to particles like the size to control all particle's size and the site attenuation to specify if distant particles should be smaller than close particles. When creating the particles, rather than using a Mesh, we use the Points class.

```
/**
 * Fireflies
 */
// Geometry
const firefliesGeometry = new THREE.BufferGeometry()
const firefliesCount = 40
const firefliesPosition = new Float32Array(firefliesCount * 3)
const firefliesScale = new Float32Array(firefliesCount)



for(let i = 0; i < firefliesCount; i++)
{
    firefliesPosition[i * 3 + 0] = (Math.random() - 0.5) * 10
    firefliesPosition[i * 3 + 1] = Math.random() * 1.5
    firefliesPosition[i * 3 + 2] = (Math.random() - 0.5) * 10

    firefliesScale[i] = Math.random()
}

firefliesGeometry.setAttribute( 'position', new THREE.BufferAttribute(firefliesPosition, 3))
firefliesGeometry.setAttribute( 'aScale', new THREE.BufferAttribute(firefliesScale, 1))
```

Figure 5.4: Fireflies geometry code

The fireflies' geometry was a custom geometry using THREE.BufferGeometry(). The position for each firefly requires x,y, and z coordinates, thus in the Float32Array, every firefly needs to be multiplied by 3, so that the entire variable can store the coordinates. In the for loop, I used Math.random() to randomly generate the x,y, and z coordinates and stored them in the array. The position and scale of the fireflies are custom attributes, so the setAttribute was used to apply these properties.

```
// Material
const firefliesMaterial = new THREE.ShaderMaterial(
    {
        uniforms:
        {
            uTime: { value: 0 },
            uPixelRatio: { value: Math.min(window.devicePixelRatio, 2) },
            uSize: { value: 200 }
        },
        vertexShader: firefliesVertexShader,
        fragmentShader: firefliesFragmentShader,
        transparent: true,
        blending: THREE.AdditiveBlending,
        depthWrite: false
    }
)
```

Figure 5.5: Fireflies materials code

The above code shows the uniforms passed into the vertex shader. uTime is used to perform positional animation of the fireflies, uPixelRatio is to make sure the fireflies remain consistent regardless of which display it is rendered in, and the uSize is to simply adjust the size of the fireflies. When fireflies are behind one another, the light needs to pass through from the one behind to the one in front. The effect needs to

be additive, therefore the blending mode is set to AdditiveBlending. In order to see this effect, the transparency is set to true. The depthWrite is set to false because it causes a clipping effect when particles are behind one another.

```
uniform float uTime;
uniform float uPixelRatio;
uniform float uSize;

attribute float aScale;

void main()
{
    vec4 modelPosition = modelMatrix * vec4(position, 1.0);
    modelPosition.y += sin(uTime + modelPosition.x * 100.0) * aScale * 0.2;
    vec4 viewPosition = viewMatrix * modelPosition;
    vec4 projectionPosition = projectionMatrix * viewPosition;

    gl_Position = projectionPosition;
    gl_PointSize = uSize * aScale * uPixelRatio;
    gl_PointSize *= (1.0 / -viewPosition.z);
```

Figure 5.6: Vertex shader code for the fireflies

There are 3 uniforms beings passed and 1 attribute. Inside the main function, the uTime is multiplied with the modelPosition inside a sin function to create a floating animation. Then this value is multiplied with the aScale to reduce the amplitude of the fireflies' movement. The smaller fireflies move less while the bigger ones move more. `gl_PointSize`simply controls the size of the fireflies and to activate the size attenuation the last formula was used.

```
void main()
{
    float distanceToCenter = distance(gl_PointCoord, vec2(0.5));
    float strength = 0.05 / distanceToCenter - 0.05 * 2.0;
    gl_FragColor = vec4(1.0, 1.0, 1.0, strength);
```

Figure 5.7: Fragment shader code for the fireflies

The distanceToCenter variable calculates the center value between `gl_PointCoord` and vec2(0.5). This value is used to calculate the alpha value, which is then passed into the `gl_FragColor` to apply colors to the fireflies.

5.5 Perlin Noise - Water and Portal

Perlin Noise algorithm was an Oscar-winning noise algorithm created for the movie "Tron". It was created by Ken Perlin during the 1980s. The Perlin noise is instrumental in recreating nature shapes like clouds, water, fire, and terrain elevation but it can also be used to animate the grass or snow moving in the wind. There are many Perlin noise algorithms with different results, and different dimensions (2D, 3D, and even 4D), some that repeat themselves, others more performant, etc. The noise produces random values which have relationships with other values. The relational values allow a smoothness effect. We enter two float values, usually, x and y, in the algorithm, and a float value is returned which is used in shaders.[2]
The vertex shader code for the water is similar to the house light shader code. There is a varying vec2 variable (vUv) which allows the vertex shader to pass values of the UV coordinates into the fragment shader.

```
varying vec2 vUv;

void main()
{
    vec4 modelPosition = modelMatrix * vec4(position, 1.0);
    vec4 viewPosition = viewMatrix * modelPosition;
    vec4 projectionPosition = projectionMatrix * viewPosition;

    gl_Position = projectionPosition;
    vUv = uv;
```

Figure 5.8: Vertex code for the water

```
void main()
{
    float strength = cnoise(vec3(vUv * uWaterPattern, uTime * uWaterSpeedMultiplier));

    vec3 color = mix(uWaterColorStart, uWaterColorEnd, strength);

    gl_FragColor = vec4(color, 1.0);
}
```

Figure 5.9: Fragment code for the water

For the water, the Perlin noise algorithm is used to form patterns. The strength variable contains a float value which is calculated by the cnoise function within the Perlin noise algorithm. It takes in two values and the returned float is used as the alpha when mixing in the colors. The mix values are then placed in the `gl_FragColor` to apply the colors in the render.
The vertex shader for the Portal is the same as the water, but the fragment shader is written differently. For the Porta; the 3D Perlin noise was used just like the water, but the patterns are drastically different.
The UV coordinates passed from the vertex shader are used to displace the original UV coordinates along with the addition of the noise produced by the Perlin noise algorithm. The result is multiplied by the uTime to allow animation at any desired speed. The effect is enhanced more by adding the displaced value with another Perlin noise. To create the outer glow, the distance of the center is calculated. The desired gradient is pushed to the edges by multiplying and offsetting with satisfactory values. Then we clamp the values between 0.0 to 1.0 to ensure that the edges are completely white. Also a step() function is applied to get the value of the strength. The first parameter of the step() function is a limit (also called edge). When the value of the second parameter is above this limit, we will get 1.0 and if the value is below this limit, we get 0.0. Instead of replacing the strength with that step() function, we are going to add it to the initial strength. We can also multiply it by a lower value to dim the step effect a little. The two uniforms uColorStart and uColorEnd are passed into a final color variable along with the new calculated strength. The final color is used to render the colors on the pattern for the scene.

5.6 Debugging Tool

An essential aspect of every creative project is making debugging easy and tweaking the code. The developer (in this case, me) and other actors working on the project

```glsl
void main()
{
    // Displace uv
    vec2 displaceUv = vUv + cnoise(vec3(vUv * 5.0, uTime * 0.1));

    // Perlin noise
    float strength = cnoise(vec3(displaceUv * 5.0, uTime * 0.2));

    // Outer glow
    float outerGlow = distance(vUv, vec2(0.5)) * 5.0 - 1.4;
    strength += outerGlow;

    // Apply sharpeness
    strength =  strength + step(-0.2, strength) * 0.8;

    // Clamp the value
    strength = clamp(strength, 0.0, 1.0);

    // Final color
    vec3 color = mix(uColorStart, uColorEnd, strength);

    gl_FragColor = vec4(color, 1.0);
}
```

Figure 5.10: Fragment shader code for Portal

(like designers or even the client) must be able to change as many parameters as possible. We have to take this into account for them to find the perfect color, speed, quantity, etc. for the best experience. There might even be unexpected results that look great. For this project, I used lil-gui to enable the tweaking process. To add an element to the panel, we must use gui.add(). The first parameter is an object and the second parameter is the property of that object we want to tweak. We need to set it after you created the concerned object. When it comes to colors, we need to use addColor() instead of add(). This is due to lil.gui not knowing if we want to tweak a text, a number, or a color just by the type of the property. Since we are using lil-gui, we can use addColor() directly on the material. There is a color picker in the panel that appears for the created object. The problem is that changing this color does not affect the material. It does change the color property of the parameter variable, but we do not even use that variable in our material. To fix that, we need lil.gui to alert us when the value changes. We can do that by chaining the onChange() method and updating the material color using material.color.set(). This method is very useful because of how many formats we can use like '#ff0000', '#f00', 0xff0000 or even 'red'.

```javascript
// Tweaks
gui.add(firefliesMaterial.uniforms.uSize, 'value' ).min(20).max(500).step(1).name('Firefly size')
gui.addColor(portalLightMaterial.uniforms.uColorStart, 'value').name('Portal start')
gui.addColor(portalLightMaterial.uniforms.uColorEnd, 'value').name('Portal end')
gui.add(waterMaterial.uniforms.uWaterSpeedMultiplier, 'value' ).min(0.1).max(20).step(0.1).name('Water speed')
gui.add(waterMaterial.uniforms.uWaterPattern, 'value' ).min(0.0).max(20).step(1).name('Water pattern')
gui.addColor(waterMaterial.uniforms.uWaterColorStart, 'value').name('Water start')
gui.addColor(waterMaterial.uniforms.uWaterColorEnd, 'value').name('Water end')
gui.addColor(windowMaterial.uniforms.uColor, 'value').name('Window color')
gui
    .addColor(debugObject, 'poleLightColor')
    .onChange(() =>
    {
        poleLightMaterial.color = new THREE.Color(debugObject.poleLightColor)
    })
```

Figure 5.11: Code for tweaks - lil.gui

5.7 Hosting and Deployment

In order to share the project online, I had to host and deploy it. The project is hosted using Vercel. Vercel is one of those "modern" hosting solutions and features continuous integration (automatization of testing, deployment, and other development steps like this). It is very developer-friendly and easy to set up. It can be used for complex projects, but also very simple "single page" websites. Also, other good alternatives should be mentioned like Netlify and GitHub Pages. Vercel directly links up with the Github repository which makes it very easy to update the page. The service needs instructions on where to place the build files for deployment. In this project, everything is built using 'npm run build' and the resulting files are placed in the 'dist' folder.

Figure 5.12: Final rendered scene along with debugging tools

Chapter 6

Conclusion

6.1 Limitations

Due to my lack of knowledge of object orientated programming with JavaScript, the code base can be considered as 'spaghetti' code. When it came to rendering high fidelity models, 3D software took a toll on my current hardware. Therefore, the overall application was made to be simple.

6.2 Final thoughts

The main take away from this project was that learning and applying this new technology was an incredible experience. It's uncommon and not widely implemented, but the option of this technology's existence encourages artistic exploration. I learned that there is a lot of trickery involved when it comes to making immersive applications. The overall application, though not realistic, shows that immersive game-like websites are possible to make. This technology is always improving. Major companies such as Google, Facebook, Microsoft are investing in this tech.

Bibliography

[1] D. Cantor and B. Jones, *WebGL beginner's guide*. Packt Publishing Ltd, 2012.

[2] I. McEwan, D. Sheets, M. Richardson, and S. Gustavson, "Efficient computational noise in glsl," *Journal of Graphics Tools*, vol. 16, no. 2, pp. 85–94, 2012.

[3] P. G. Vivo and J. Lowe, "The book of shaders," *Dosegljivo: https://thebookofshaders.com*, 2015.

[4] E. Angel and E. Haines, "An interactive introduction to webgl and three. js," in *ACM SIGGRAPH 2017 Courses*, 2017, pp. 1–95.