ThreeJS Assignment report

Desarrollo de una aplicación web “Galería 3D” basada en ThreeJS

Marcos Jesús Barrios Lorenzo, 27/12/2023
Sistemas y Tecnologías Web: Cliente subject of *Universidad de La Laguna*.

Index

Image Index

Assignment description

A webpage with 3D scenes and interaction is to be built. The requirements are:

- Create more than one ThreeJS scene and use them on different places.
- Have selectable 3D models in the scenes which trigger webpage content changes on click.
- Be able to download the models used.
- Use light with shadows casted to a surface.
- Have at least one animated 3D Model on one scene.
- Make a report explaining it’s technical aspects and functionality. It is also asked for it to be in the english language.

Local installation

Download the latest release from the github repository or perform a *git clone* to obtain the folder with the proyect.

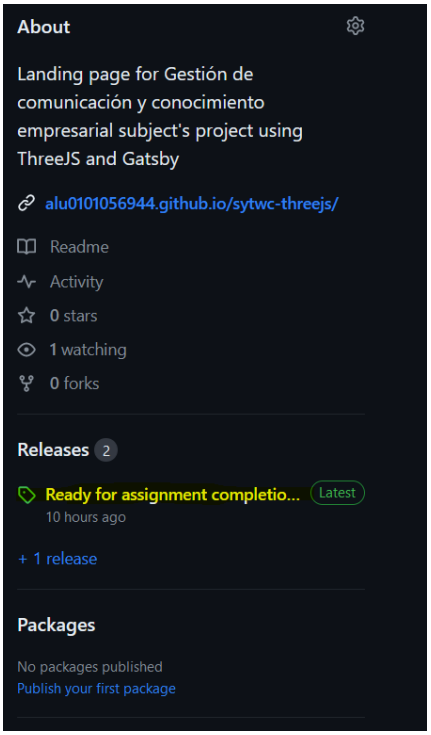

Figura 1: https://github.com/alu0101056944/sytwc-threejs, right sidebar screenshot

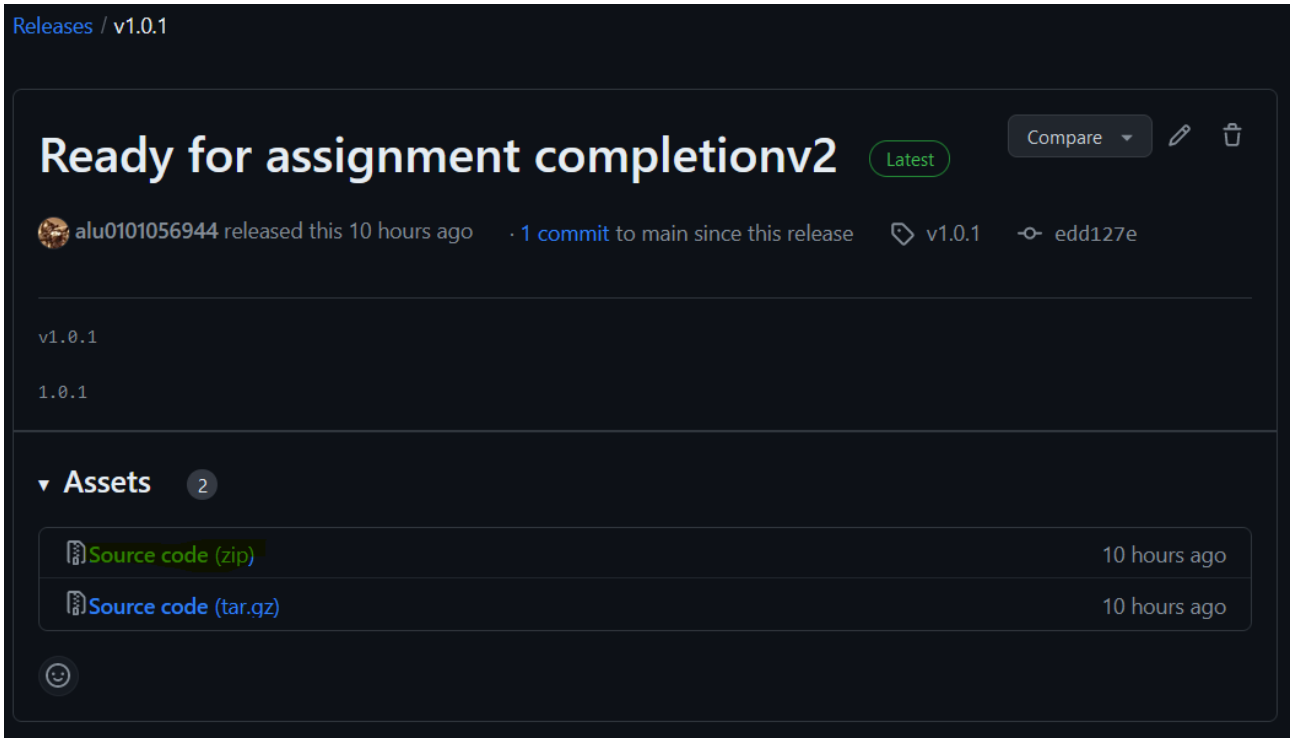

Figura 2: Screenshot of releases section, source code (.zip) highlighted

Then, open the folder from a terminal and execute npm install first, then npm run build and finally npm run serve to open a local server, allowing webpage access on http://localhost:9000

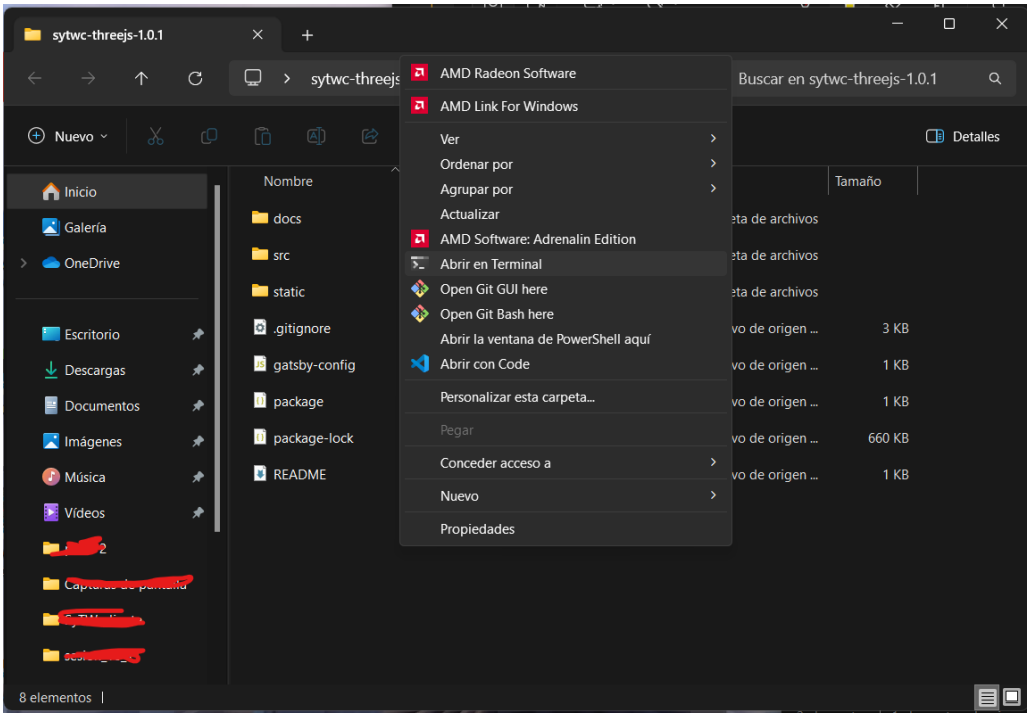

Figura 3: Open in terminal option (windows 11) from the folder

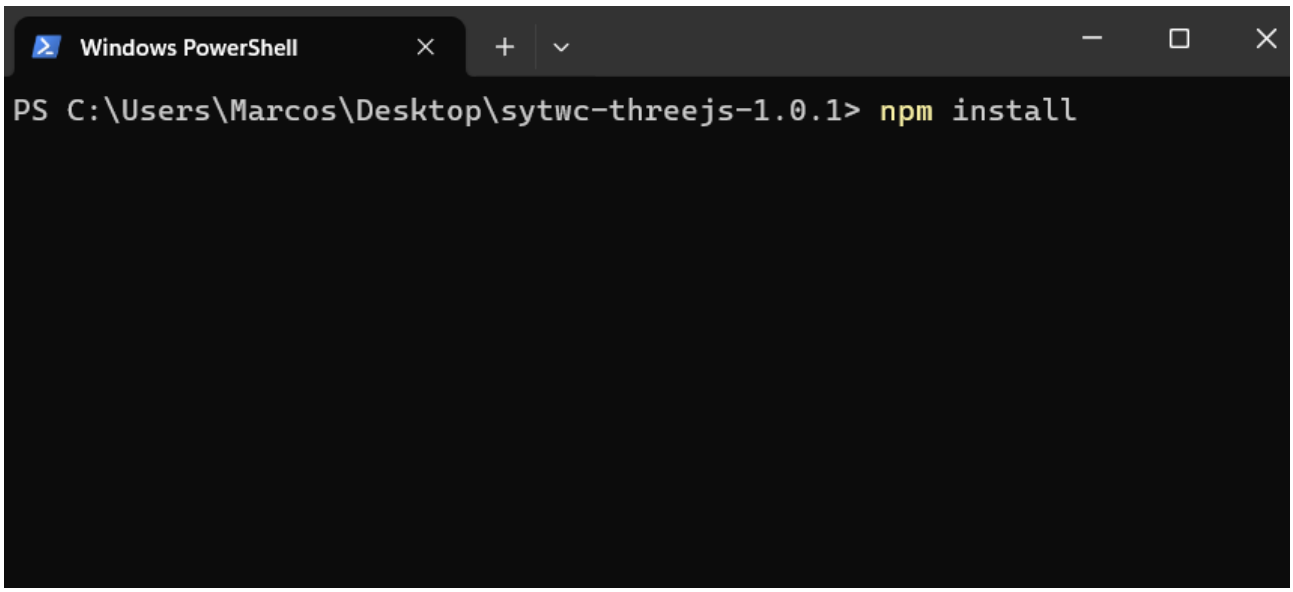

Figura 4: screenshot of npm run install cli command typed in terminal

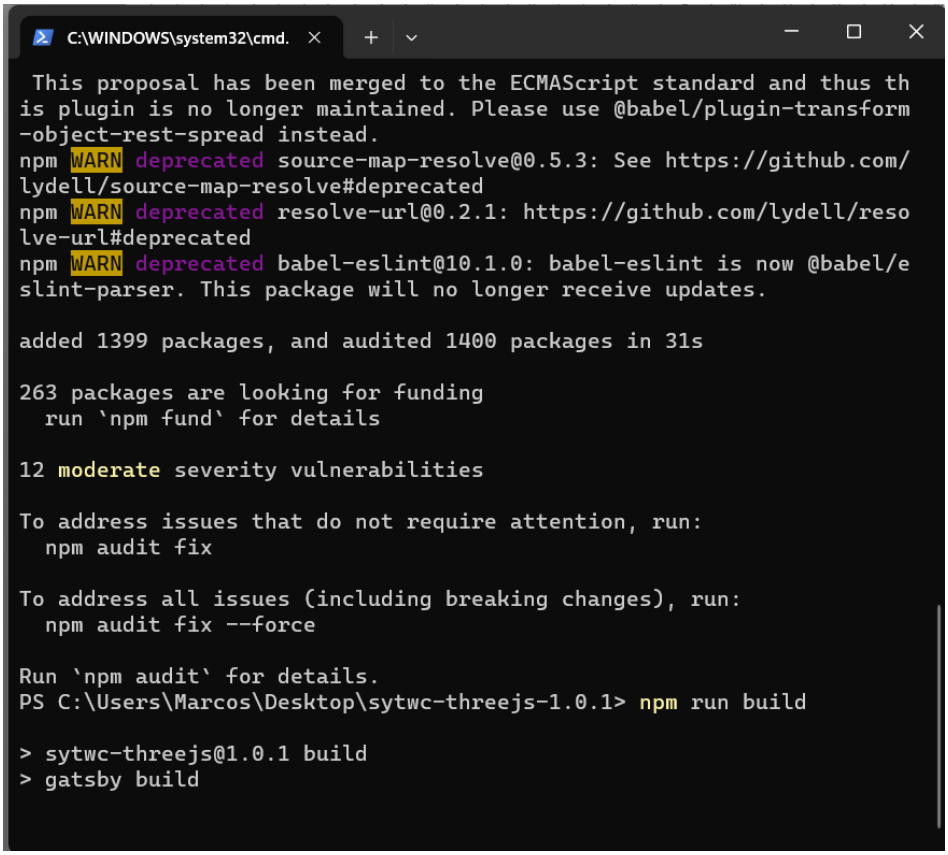

Figura 5: screenshot after npm install execution and after pressing npm run build (the later still in process)

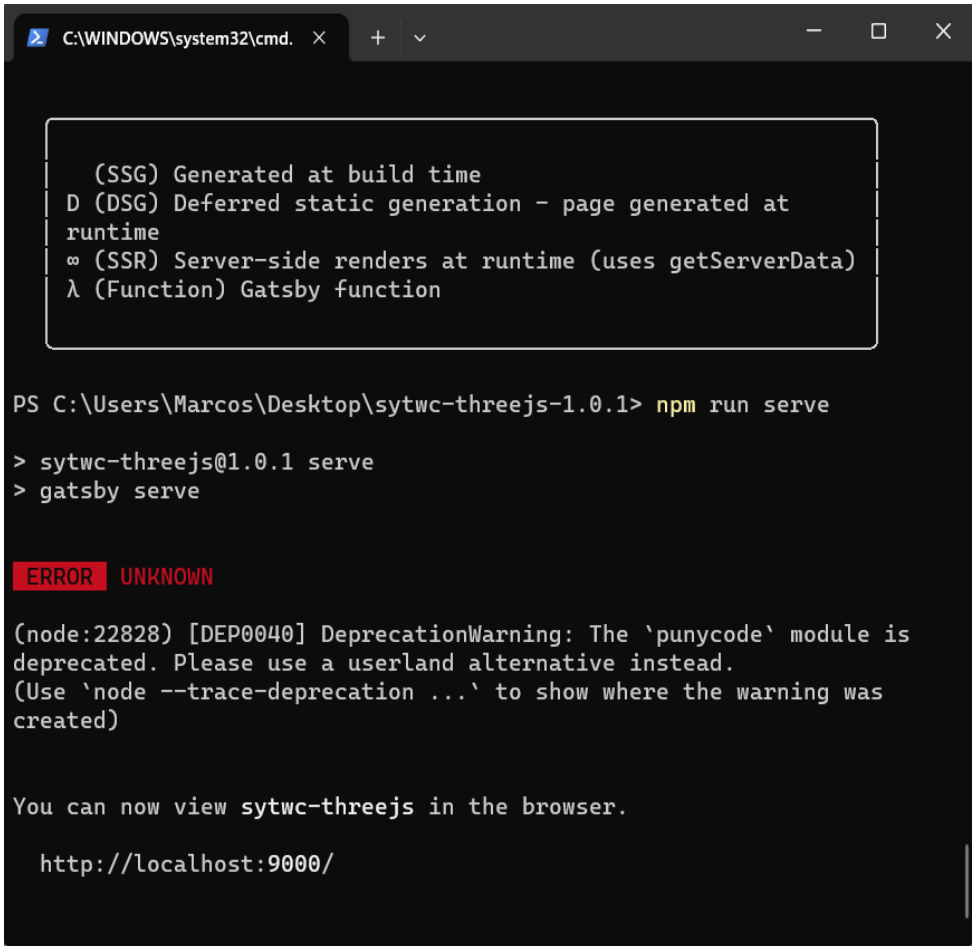

Figura 6: screenshot after npm run build command finish and after npm run serve command execution

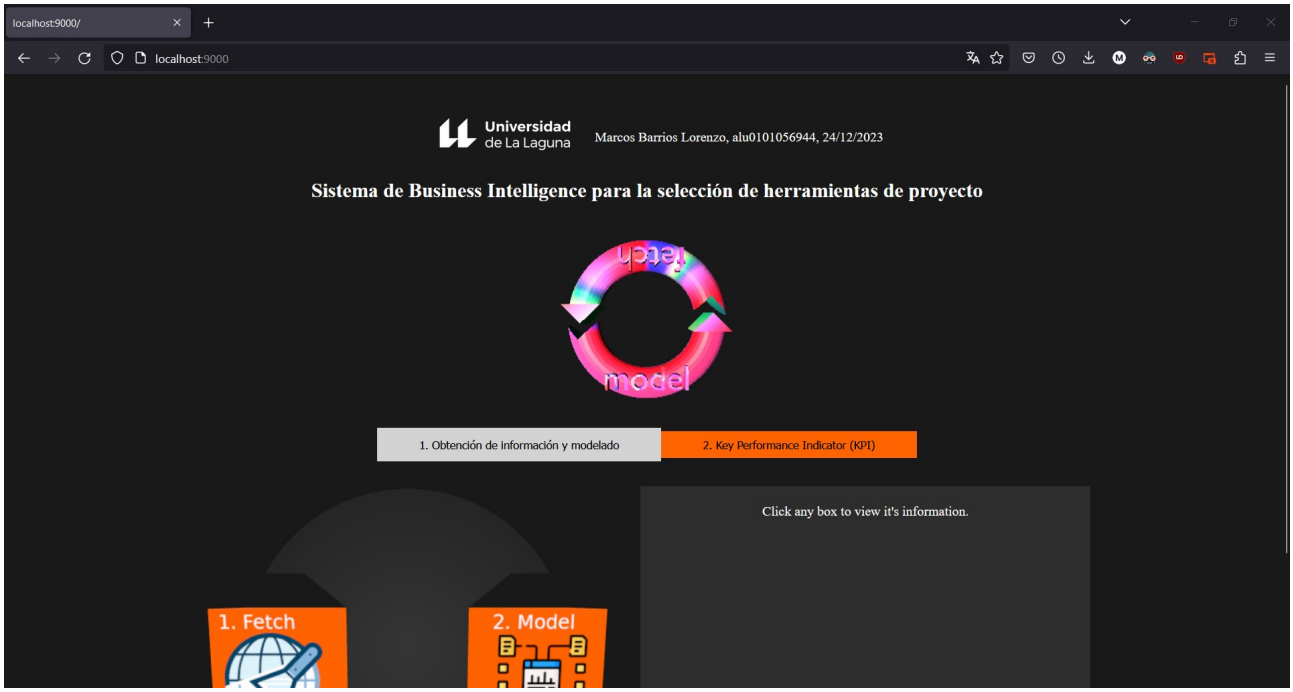

Figura 7: screenshot of localhost:9000 navigation result after executing the cli command npm run serve

Functionality

The main functionality is the tabs component, the first tab contains a scene with orange boxes that can be clicked and the right sidebar will change accordingly. Can select the active tab.

There is something in the webpage for each requirement:

- **Create more than one ThreeJS scene and use them on different places:** There are three scenes. The rotating torus on top of the tabs section, then on the first tab there is another on the left side, then on the second tab there is another again on the left side.
- **Have selectable 3D models in the scenes which trigger webpage content changes on click.:** On the first tab, the orange boxes can be clicked and the content of the right side changes.
- **Be able to download the models used:** Links were added as content on the right side of each tab when pressing the orange boxes or directly in the case of the second tab.
- **Use light with shadows casted to a surface:** The first tab's scene has shadows casted from orange boxes and at the background white boxes.
- **Have at least one animated 3D Model on one scene:** Both the top scene and the first tab's scene have animations.
- **Make a report explaining it's technical aspects and functionality. It is also asked for it to be in the english language:** The present document fulfils this requirement.

Technical features

Tecnologies used

Nodejs programming language and Gatsbyjs framework are used for building the webpage. Gatsbyjs allows the creation of React components declaratively and it follows the JAMStack methodology, so the build process involves first performing a server-side static rendering step of the webpage. To do that, gatsbyjs uses react to render the components into dom nodes and webpack to minify and package all the dependencies into as small as possible files. These dependencies are obtained through node's npm package manager, one of them being the 3D library, threejs. Finally, git version control and github repository hosting are also used and the repository can be accessed through https://github.com/alu0101056944/sytwc-threejs. Thanks to github pages the webpage can be accessed directly from https://alu0101056944.github.io/sytwc-threejs/.

A list of all the dependencies can be viewed on the *package.json* file's dependencies section, on the github repository.

Folder structure

```
│
├─────docs/  assets for the README.md file
│
├─────src/  webpage source code and images
│    ├─────images/ images, in this case only the ull logo is present
│    │
│    ├─────pages/ one .js file per webpage route. Contains an index.js for the '/' route.
│    │
│    ├─────styles/ contains sass files. Contains main.scss
│    │    │
│    │    └─────templates/ .scss files for some components
│    │
│    └─────templates/ one .js file per component
│
└─────static/ empty, due to prefix requirements of gh-pages everything is under sytwc-threejs/
     └─────sytwc-threejs/ glTF assets and other .js exports shown after box click on the webpage
```

Components explained

In the pages/ folder each .js represents a route. Index.js for '/', 404.js for '/404' (left as gatsby

default generated). Gatsby automatically converts each of those .js to a page.

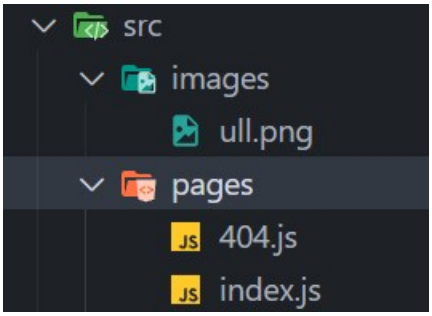

Figura 8: folder structure pages/

On each of those, there is a default export that returns a JSX component:

```
11  const IndexPage = () => {
12    const tabsContent = [
13          {
14            title: '1. Obtención de información y modelado',
15            content: <FirstGLBlock key='ap1'/>
16          },
17          {
18            title: '2. Key Performance Indicator (KPI)',
19            content: <SecondGLBlock key='ap2'/>
20          },
21        ];
22
23    return (
24      <div className='mainDiv'>
25        <div className='logoDiv'>
26            <StaticImage src={'../images/ull.png'} alt='Gastby logo' width={200} height={100}/>
27            <div style={{display: 'flex', justifyContent: 'center', marginLeft: 20}}>
28              <p>Marcos Barrios Lorenzo, alu0101056944, 24/12/2023</p>
29            </div>
30        </div>
31
32        <h1 className='mainTitle'>
33          Sistema de Business Intelligence para la selección de herramientas de proyecto
34        </h1>
35
36        <div className='animationDiv'>
37          <MainAnimation>dasda</MainAnimation>
38        </div>
39
40        <Tabs allContent={tabsContent}/>
41      </div>
42    );
43  }
44
45  export default IndexPage;
```

Figura 9: src/pages/index.js Ver https://github.com/alu0101056944/sytwc-threejs/blob/main/src/pages/index.js

Notice how there are special html tags being used: <StaticImage>(line 26), <MainAnimation> (line 37), <Tabs> (line 40). Those are what gastbyjs calls “templates”, and each one is either in a

separate .js file (src/templates/) or comes from a gatsby plugin (*gastby-plugin-image* for StaticImage), but here I call them “components” because they can be reused on other components and pages/.js files. All that is needed is an import:

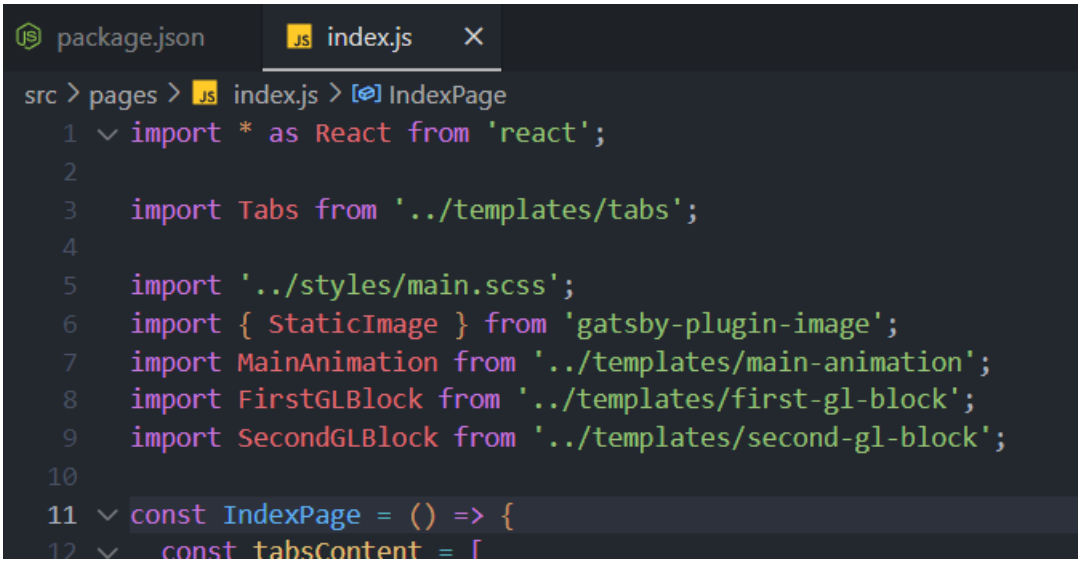

Figura 10: imports at src/pages/index.js

Notice how there is an import for ‘../styles/main.scss’, gatsbyjs allows .scss file imports and it automatically compiles them into .css, but the styles it contains are global; they are applied to all the components and pages, so each className used in html tags must be unique when wanted. There are many ways to make modular styling in gastby but it is not applied in this assignment.

Back to the components, there are four components programmed by me. **<Tabs>**, **<MainAnimation>**, **<FirstGLBlock>** (which internally uses another component I created, <ContentAndSidebar>) and **<SecondGLBlock>** (internally uses <ContentAndSidebar> too)

The first represents a tabs content (just like for example, in a webbrowser, where each tab is a webpage visited) and receives an array with the title of the tab and the content of the tab. In this case two tabs are created, one for the first scene where I use the **<FirstGLBlock>** component, and another for the second scene where I use the **<SecondGLBlock>** component. The remaining component, **<MainAnimation>**, is directly inserted into index.js.

Figura 11: <MainAnimation> component inserted directly on index.js page

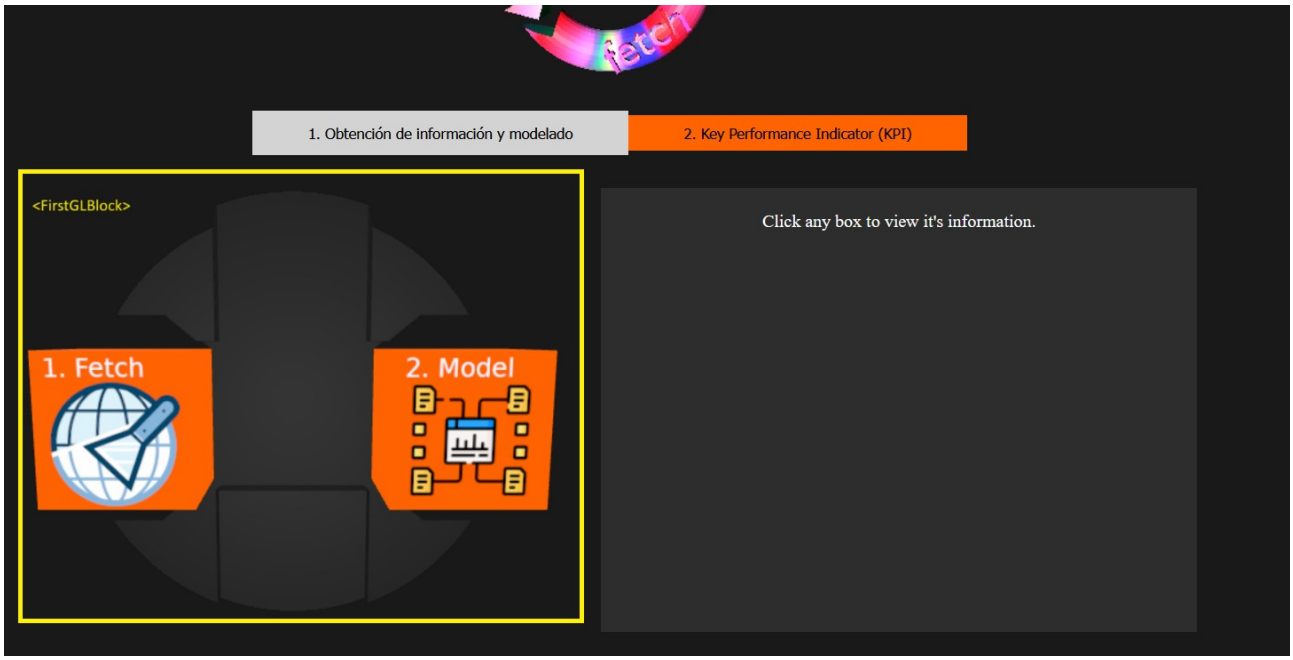

Figura 13: <FirstGLBlock> component within <Tabs> component

Figura 14: <SecondGLBlock> component within <Tabs>

Lets explain the **<Tabs>** component.

<Tabs> component explanation

```
const Tabs = ({allContent = [{title: 'default'}]}) => {
  const [nameOfActiveTab, setNameOfActiveTab] = useState(allContent[0].title);
  Tabs.buttonAmount ??= 0; // for keys

  return (
    <div className='tabsContainer'>
      <nav>
        {
          allContent.map((content, index) => {
                Tabs.buttonAmount++;
                return <button
                        className={nameOfActiveTab === content.title ? 'activeTab' : 'inactiveTab'}
                        key={`${content.title}${index}${Tabs.buttonAmount}`}
                        onClick={() => setNameOfActiveTab(content.title)}>
                      {content.title}
                    </button>
              })
        }
      </nav>
      <main>
        {
          allContent.filter(content => content.title === nameOfActiveTab)
              .map(content => content.content)
        }
      </main>
    </div>
  );
}
```

Figura 15: <Tabs> component source code. See https://github.com/alu0101056944/sytwc-threejs/blob/main/src/templates/tabs.js

With the array content as parameter (remember, each element is an object with a title and a content property), for each content element it inserts a <button> into <nav> with some attributes I'll explain after describing some fundamental concepts in gastbyjs.

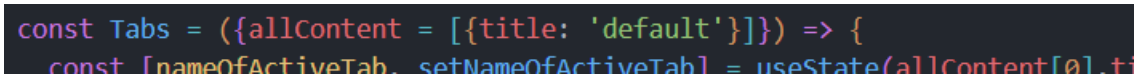

Figura 16: Parameters of <Tabs> component

```
const IndexPage = () => {
  const tabsContent = [
        {
          title: '1. Obtención de información y modelado',
          content: <FirstGLBlock key='ap1'/>
        },
        {
          title: '2. Key Performance Indicator (KPI)',
          content: <SecondGLBlock key='ap2'/>
        },
      ];

  return (
```

Figura 17: content array passed as allContent attribute to <Tabs> component from index.js. Note: key is required for gatsbyjs to distinguish elements within list, it is just a string representing a unique key for that element.

Note: In Gatsby, the *className* attribute is exactly the same as *class* in html.

In gastbyjs the function that returns the component is called many times, and between each call It is sometimes wanted to have persistent values between calls. For that, gastbyjs allows defining a state variable: "*const [nameOfActiveTab, setNameOfActiveTab] = useState(allContent[0].title);*", the first variable *nameOfActiveTab* is the value of the state, while the second variable is a function that is to be called each time I chaneg the state. The argument passed to *useState* is the initial value of *nameOfActiveTab*.

And an important thing to understand is that each time setNameOfActiveTab is called then the component is queued for re-rendering with the new variable's state.

In this case, the button will call that *setNameOfActiveTab* and trigger a rerender of the component so the component will be created again with the new state, that's why the *classsName* atribute of the button depends on the *nameOfActiveTab* variable; depending on which value it has the button will have one look or another.

```
return (
  <div className='tabsContainer'>
    <nav>
      {
        allContent.map((content, index) => {
              Tabs.buttonAmount++;
              return <button
                      className={nameOfActiveTab === content.title ? 'activeTab' : 'inactiveTab'}
                      key={`${content.title}${index}${Tabs.buttonAmount}`}
                      onClick={() => setNameOfActiveTab(content.title)}>
                    {content.title}
                  </button>
            })
      }
    </nav>
    <main>
```

Figura 18: onClick attribute on <Tabs>'s <nav>

Then there is the <main> tag. Each render the .content property of the input content array is inserted into the <main> tag. So, because the input array content has two elements, one with .content **<FirstGLBlock>** and the other with .content **<SecondGLBlock>**, then the **<Tabs>** component will insert one or the other depending on the *nameOfActiveTab* state variable.

```
</nav>
<main>
  {
    allContent.filter(content => content.title === nameOfActiveTab)
        .map(content => content.content)
  }
</main>
/div>
```

Figura 19: <main> inserts only content input array element's whose title matches the current name of active tab

MainAnimation explanation

Back to index.js, there is a <MainAnimation> component:

```
return (
  <div className='mainDiv'>
    <div className='logoDiv'>
        <StaticImage src={'../images/ull.png'} alt='Gastby logo' width={200} height={100}/>
        <div style={{display: 'flex', justifyContent: 'center', marginLeft: 20}}>
          <p>Marcos Barrios Lorenzo, alu0101056944, 24/12/2023</p>
        </div>
    </div>

    <h1 className='mainTitle'>
      Sistema de Business Intelligence para la selección de herramientas de proyecto
    </h1>

    <div className='animationDiv'>
      <MainAnimation>dasda</MainAnimation>
    </div>

    <Tabs allContent={tabsContent}/>
  </div>
);
```

Figura 20: index.js return component with the <MainAnimation> component

This is it’s definition:

```
const MainAnimation = () => {
  React.useEffect(setupScene, []);

  return (
    <div className='mainAnimDiv'>
      {/* setupScene function attaches the renderer as child of this node */}
    </div>
  )
}
```

Figura 21: Definition of the <MainAnimation> component

It’s just a div. But the important thing here is the *React.useEffect(setupScene, []);* sentence. First, *useEffect* executes the first argument function once per *MainAnimation()* call (commonly described as component mount) because the second argument, which should contain variable references whose change would trigger the first argument function to execute, is just an empty array.

The *setupScene* function uses ThreeJS to setup an scene and attach it to the div with *className=’mainAnimDiv’*. Lets explain the function:

```
function setupScene() {
    const VIEWPORT_WIDTH = 250;
    const VIEWPORT_HEIGHT = 200;

    const scene = new three.Scene();
    const camera = new three.PerspectiveCamera(45, VIEWPORT_WIDTH / VIEWPORT_HEIGHT,
        0.1, 1000);

    const renderer = new three.WebGLRenderer();
    renderer.setClearColor(new three.Color(0x191919));
    renderer.setSize(VIEWPORT_WIDTH, VIEWPORT_HEIGHT);
    renderer.shadowMap.enabled = true;
    renderer.shadowMap.type = three.PCFSoftShadowMap

    const spotLight = new three.SpotLight(0xFFFFFF);
    spotLight.position.set(0, 10, 2);
    spotLight.angle = Math.PI / 3;
    spotLight.castShadow = true;
    spotLight.intensity = 200;
    spotLight.distance = 15;
    spotLight.penumbra = 0.2;
    spotLight.shadow.mapSize = new three.Vector2(1024, 1024);
    scene.add(spotLight);

    const ambientLight = new three.AmbientLight(0x353535);
    scene.add(ambientLight);

    const planeGeometry = new three.PlaneGeometry(60, 20);
    const planeMaterial = new three.MeshLambertMaterial({
            color:0x353535
        });
    const plane = new three.Mesh(planeGeometry,planeMaterial);
    plane.position.set(0,0,0);
    scene.add(plane);

    const loader = new GLTFLoader();
```

Figura 22: Part 1 of the setupScene function

After importing three (*import * as three from 'three';*) a scene is instanced, a camera is setup and then the renderer is setup.

This scene is meant to have a "Torus" 3D Model with a rotating animation that represents the "Fetch → Model" procedure in which information is gathered from website scraping techniques and then it is modeled into a database (The idea comes from the *Gestion de la comunicación y Conocimiento Empresarial* subject's business intelligence project, which is as of writing this a work in progress)

Figura 23: Torus 3DModel with a texture. (The texture may look incorrectly placed but it is intentional)

Note: the plane is unused. It was accidentaly left in place but it is not shown in the final webpage, so please ignore it.

To achieve that scene in *Figura 23* the camera, a Spolight and an Ambientlight were carefully placed. That and clearColor adjustments made the look current to be seen in the webpage. **But more importantly, a process involving blender's glTF2.0 export file format and threejs's GLTFLoader took place**. See the second part of the code:

```
new Promise((resolve, reject) => {
    loader.load(
        'torus_final.glb',
        gltf => {
          resolve();
          const torus = gltf.scene.getObjectByName('TorusFinalWithText');
          torus.position.set(0, 5, 0);
          spotLight.target = torus;
          scene.add(gltf.scene.children[0]);
          camera.position.set(0, 8.2, 0);
          camera.lookAt(torus.position);
          camera.rotation.z = Math.PI / 2;

          const mixer = new three.AnimationMixer(torus);
          gltf.animations.forEach(clip => mixer.clipAction(clip).play());
          const clock = new three.Clock();
          animate();
          async function animate() {
            requestAnimationFrame(animate);
            var delta = clock.getDelta();
            if (mixer) mixer.update(delta);
            renderer.render(scene, camera);
          }

          // clear container first
          const container = document.querySelector('.mainAnimDiv');
          if (container.children.length > 0) {
            for (const child of container.children) {
              child.remove();
            }
          }
          container.appendChild(renderer.domElement);

          renderer.render(scene, camera);
        },
        undefined,
        (error) => reject(error)
      );
```

Figura 24: Second and final part of the setupScene code

So the idea here is to import the .glb file format (exported from blender), which includes materials, animations and all the 3D Models in the scene, and load them into the ThreeJS scene.

A promise is created and solved when threejs GLTFLoader.load() sucessfully loads the scene. This is done because the next sentences are related to the exported scene and it needs to be loaded before continuing. That's why the first sentence in the callback is a "*resolve()*".

The "*// clear container first*" part is done because ThreeJS uses imperative programming while

The *animate()* would be the equivalent of the *gameloop*, but because this threejs scene only animates a single object, then it was called *animate()* and treated as a separate async function.

Note: Gatsbyj is declarative, and because gatsbyjs mounts the component by calling it's function, and it could happen many times due to page changes, then many ThreeJS canvas could get accumulated overtime, then some sentences for dom node cleanup are added to prevent it from happening. It would be ideal to use threejs's react fiber for a declarative approach instead but for the assignment's correctness then the classic ThreeJS library is used.

The model itself has the following texture:

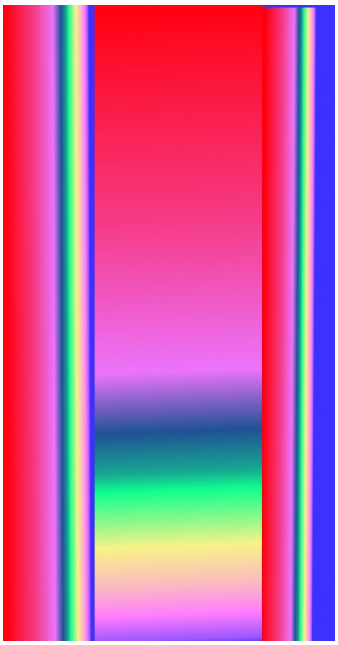

Figura 25: torus texture (less width here than the real image)

And is the result of many blender boolean modifiers. The animation was setup at object level by changing the rotation while putting keyframes on the bottom animation bar.

FirstGLBlock

Like I mentioned previously, both this and <SecondGLBlock> uses a <ContentAndSidebar>, which is a simple structure which is changed by css to make it look like a split div (content on left, sidebar on right, but both are same width in this case):

```jsx
const FirstGLBlock = () => {
  React.useEffect(setupScene, []);

  return (
    <ContentAndSidebar
        key='content1'
        sidebarContent={<p style={{color: 'white'}}>Click any box to view it's information.</p>}>
      <div className='glBlock'>
        {/* The canvas is attached here */}
      </div>
    </ContentAndSidebar>
  );
}
```

Figura 27: <FirstGLBlock> component. See at https://github.com/alu0101056944/sytwc-threejs/blob/main/src/templates/first-gl-block.js

The <ContentAndSidebar> has a *sidebarContent* attribute that receives the JSX to display at the right side. By default it is a simple paragraph. The children content is put on the left side, in this case a div with a *glBlock* class that the ThreeJS canvas attaches to.

There is again a *setupScene* function that creates the ThreeJS scene. This is the longest *setupScene* in the webpage, so multiple parts will be shown on images. Let’s check it out:

```
function setupScene() {
  const VIEWPORT_WIDTH = 500;
  const VIEWPORT_HEIGHT = 400;

  const scene = new three.Scene();
  const camera = new three.PerspectiveCamera(45, VIEWPORT_WIDTH / VIEWPORT_HEIGHT,
      0.1, 1000);

  const renderer = new three.WebGLRenderer({ antialias: true });
  renderer.setClearColor(new three.Color(0x232323));
  renderer.setSize(VIEWPORT_WIDTH, VIEWPORT_HEIGHT);
  renderer.shadowMap.enabled = true;

  const ambientLight = new three.AmbientLight(0x373737);
  scene.add(ambientLight);

  const loader = new GLTFLoader();
  new Promise((resolve, reject) => {
      loader.load(
          '/animation1_emission.glb',
          gltf => {
            resolve();
            scene.add(gltf.scene);
            gltf.scene.traverse(object => {
                  object.castShadow = true;
                  object.receiveShadow = true;
                });

            const centerCube =
```

Figura 28: First part of the setupScene of <GLFirstBlock>.

We see a scene instance, a camera instance, a renderer instance (with antialias on to improve the graphical rendering). Then an ambient light, which is used to match the unlit parts on the scene with the background color of the webpage for appearance reasons. Then, we load /animation1_emission.glb, which is the following scene:

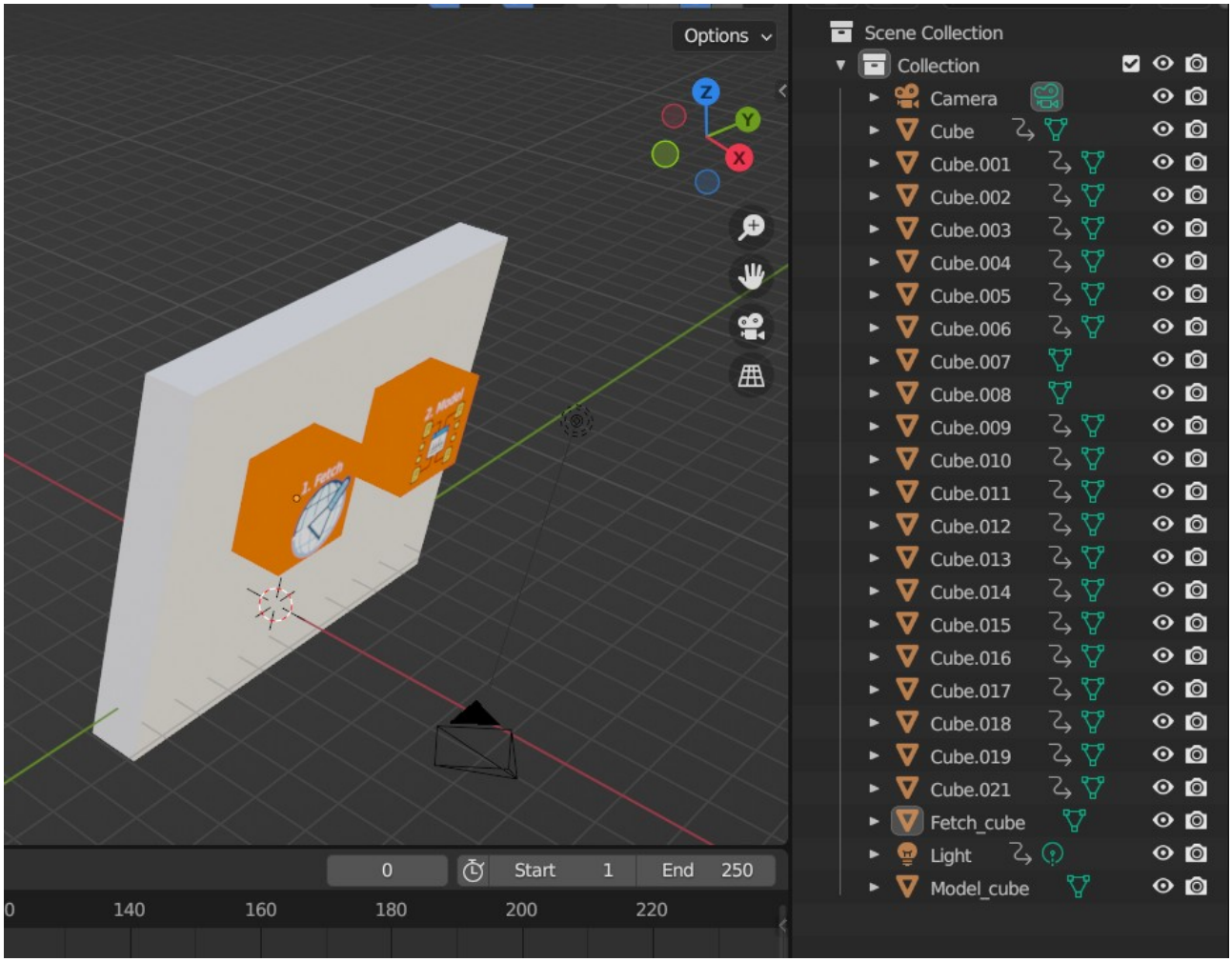

Figura 29: animation1_emission.glb on blender

The scene is a wall of boxes with two orange boxes highlighted:

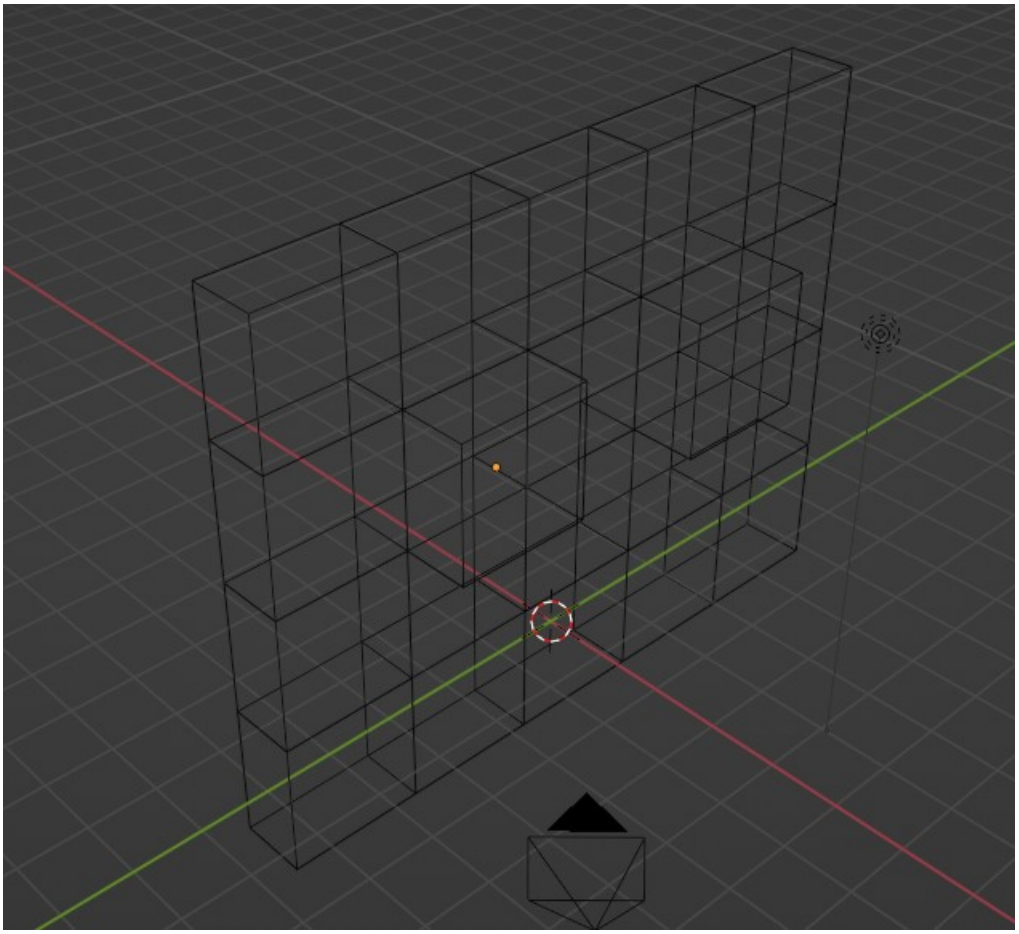

Figura 30: Wirefram eof animation1_emission.glb scene.

All the white boxes have been animated to move a bit forward and then go back in different phases and different speeds:

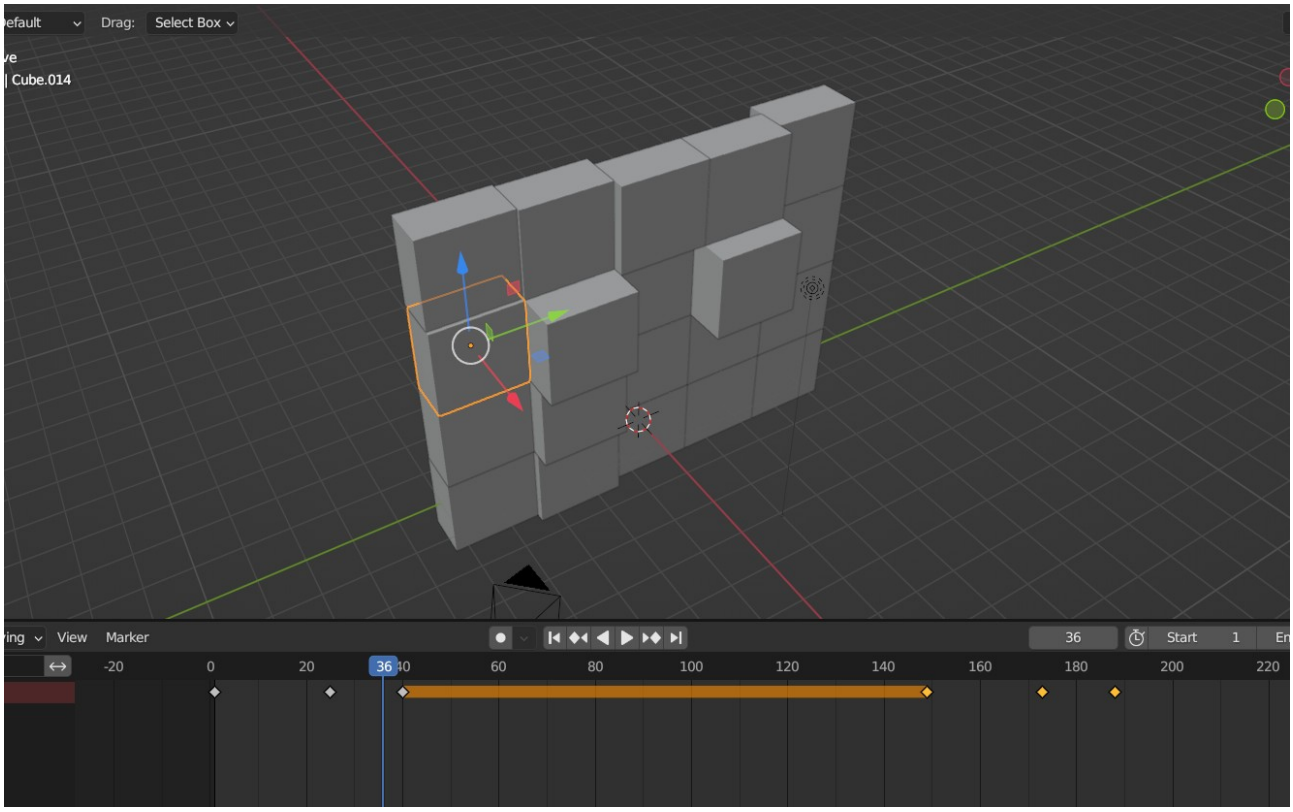

Figura 31: Animation window wth a white box selected. The orange boxes were not animated.

The orange boxes have a special “Emission” material that does not react to light, which is wanted in this case because they are meant to be interaction boxes like button that can be clicked.

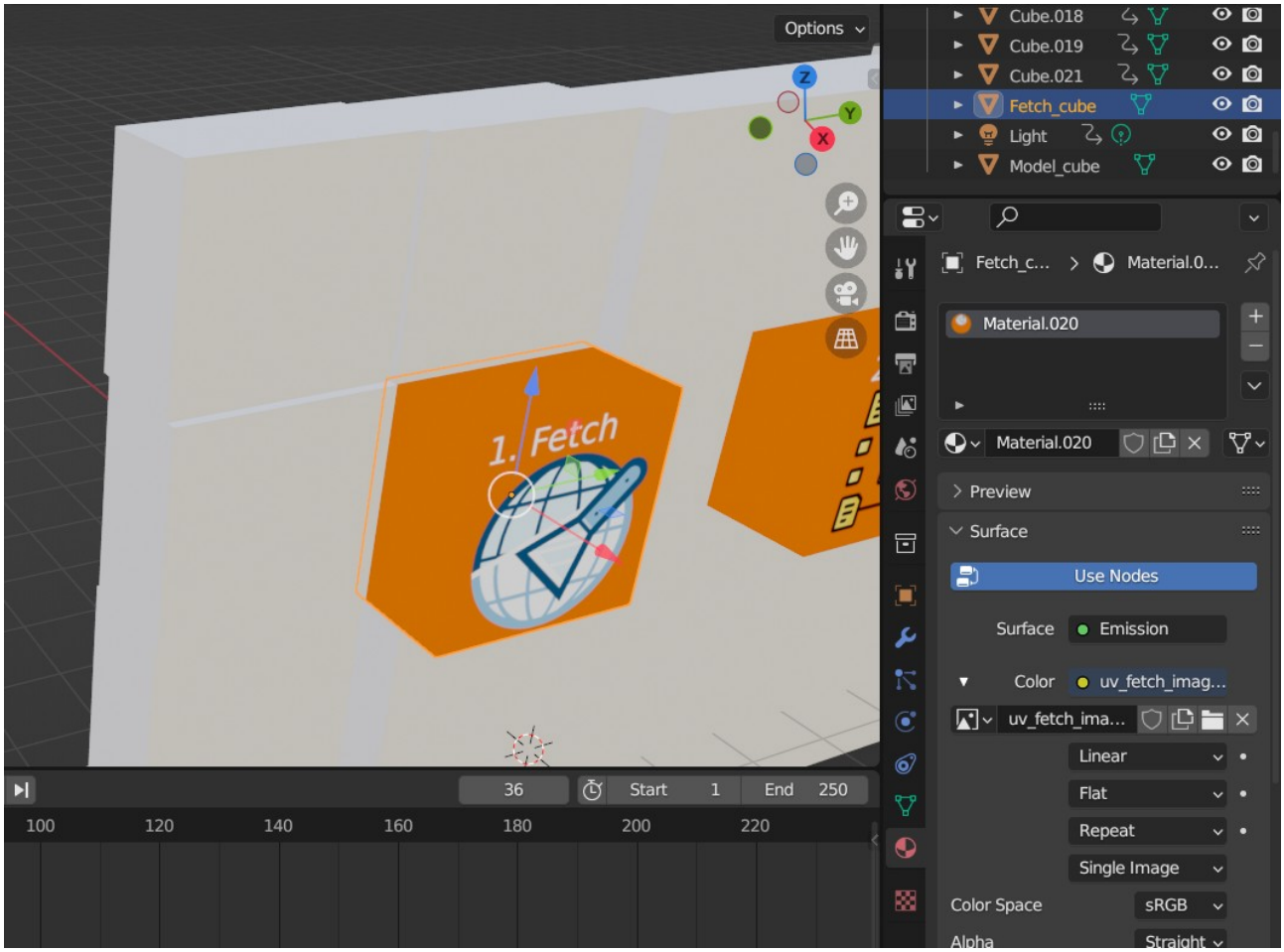

Figura 32: Material of orange boxes

A UV map layout was exported and then, using GIMP image editing, the texture was setup:

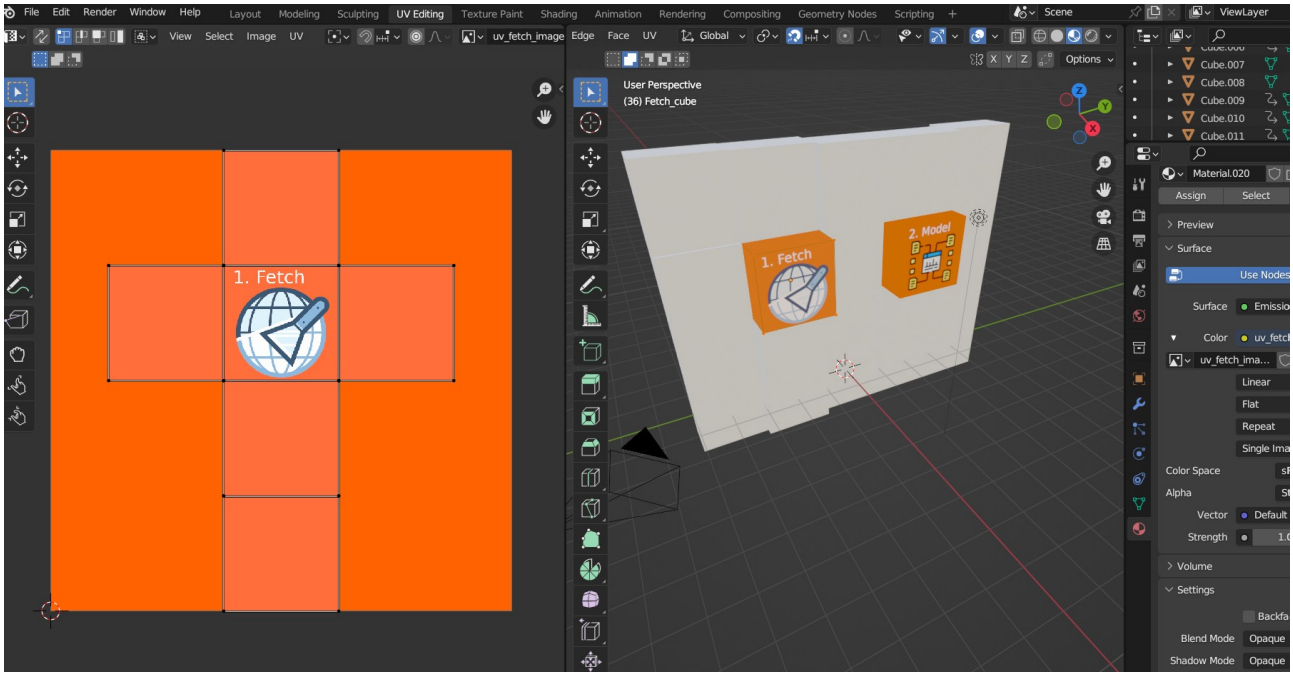

Figura 33: UV Map of fetch orange box

The whole scene was exported as glTF2.0 format:

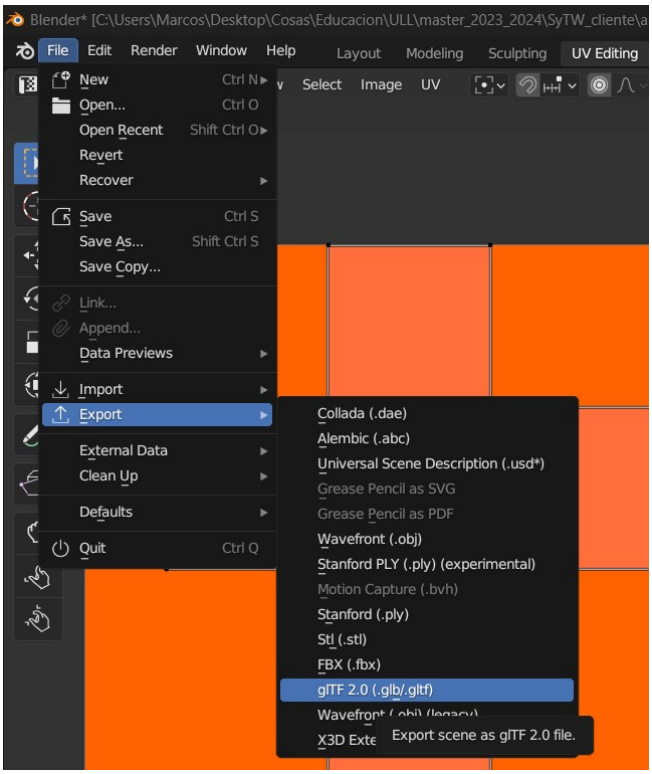

Figura 34: Exporting the scene as glTF2.0 format

Then the ThreeJS was setup with a Spotlight so that it lights the center of the box wall, complementing the Ambient light.

Both castShadows and receiveShadow was enabled on all objects of the scene by using *traverse()*:

```
const loader = new GLTFLoader();
new Promise((resolve, reject) => {
    loader.load(
        '/animation1_emission.glb',
        gltf => {
          resolve();
          scene.add(gltf.scene);
          gltf.scene.traverse(object => {
                object.castShadow = true;
                object.receiveShadow = true;
              });
```

Figura 35: enable all shadows using traverse

Some camera setup was done:

```
38
39          const centerCube =
40                gltf.scene.getObjectByName('Cube007');
41          const spotLight = new three.SpotLight(0xFFFFFF);
42          spotLight.position.set(4, 4, 0);
43          spotLight.angle = Math.PI / 4.5;
44          spotLight.target = centerCube;
45          spotLight.castShadow = true;
46          spotLight.shadow.bias = -0.0005;
47          scene.add(spotLight);
48
49          camera.rotation.z = Math.PI / 2;
50          camera.position.set(7.8, 5, 0);
51          const center = new three.Vector3(scene.position.x,
52                scene.position.y + 4, scene.position.z);
53          camera.lookAt(center);
54
```

Figura 36: Camera and light setup. Cube007 is the one in the middle of the white box wall.

And also the animation setup:

```
const mixer = new three.AnimationMixer(gltf.scene);
gltf.animations.forEach(clip => mixer.clipAction(clip).play());
const clock = new three.Clock();
```

Figura 37: Animation setup on setupScene of <GLFirstBlock>. The clock variable is used on the update() of gameloop()

Now comes the mouse interaction and the gameloop() function:

```
            // clear container first
            const container = document.querySelector('.glBlock');
            if (container.children.length > 0) {
              for (const child of container.children) {
                child.remove();
              }
            }
            container.appendChild(renderer.domElement);

            const modelCube = scene.children[1].getObjectByName('Model_cube');
            modelCube.scale.set(0.9, 0.9, 0.9);
            const fetchCube = scene.children[1].getObjectByName('Fetch_cube');
            fetchCube.scale.set(0.9, 0.9, 0.9)

            let mousePosition = new three.Vector2();
            window.addEventListener('pointermove', (event) => {
              const rect = renderer.domElement.getBoundingClientRect();
              mousePosition.x =
                  ((event.clientX - rect.left) / (rect.right - rect.left)) * 2 - 1;
              mousePosition.y =
                  - ((event.clientY - rect.top) / (rect.bottom - rect.top)) * 2 + 1;
            });

            function gameloop() {
              const delta = clock.getDelta();
              if (mixer) {
                mixer.update(delta);
              }
              update(scene, camera, mousePosition);
              renderer.render(scene, camera);
              requestAnimationFrame(gameloop);
            }
            gameloop();
          },
          undefined,
          (error) => reject(error)
        );
      });
```

Figura 38: More of setupScene() of GLFirstBlock

Ignoring the same dom node cleanup I did earlier with <MainAnimation>, the objective of the next code is to have the **orange boxes scale down when the mouse is hovering over them**. For that, the mouse position is registered and passed to is registered and calibrated for the ThreeJS scene position. Then the gameloop is called and inside it the mixer receives a delta to advance the animations. The highlight here is the *update()* function, which handles the scaling of the orange boxes on hover but also something new; it adds a new click event listener so that when an orange box is clicked it updates the sidebar content (it also changes mouse appearance just like a button):

```
function update(scene, camera, mousePosition) {
  document.body.style.cursor = "auto";
  const fetchCube = scene.children[1].getObjectByName('Fetch_cube');
  fetchCube.scale.set(0.9, 0.9, 0.9);
  const modelCube = scene.children[1].getObjectByName('Model_cube');
  modelCube.scale.set(0.9, 0.9, 0.9);
  window.removeEventListener('click', setFetchContent);
  window.removeEventListener('click', setModelContent);

  const raycaster = new three.Raycaster();
  raycaster.setFromCamera(mousePosition, camera);
  const intersects = raycaster.intersectObjects(scene.children);
  for (let i = 0; i < intersects.length; i++) {
    if (intersects[i].object.id === fetchCube.id) {
      fetchCube.scale.set(0.7, 0.7, 0.7);
      window.addEventListener('click', setFetchContent, { once: true });
      document.body.style.cursor = "pointer";
    } else if (intersects[i].object.id === modelCube.id) {
      modelCube.scale.set(0.7, 0.7, 0.7);
      window.addEventListener('click', setModelContent, { once: true });
      document.body.style.cursor = "pointer";
    }
  }
}
```

Figura 39: update of <GLFirstBlock>

The idea is to reset the scale first and then cast a Raycast from the camera according to current mouse position, and if either of the orange boxes are intersected, then change the scale of the box, add a click event listener (and always clear the click event listener by default, so that when the mouse moves away it aways gets removed) and change the mouse to pointer appearance. Both *setFetchContent* and *setModelContent* are similar:

```
123
124   // meant for event listener click on the fetch box
125   function setFetchContent() {
126     const sidebarDOMNode = document.querySelector('.sidebar');
127     if (sidebarDOMNode) {
128       for (const child of sidebarDOMNode.children) {
129         child.remove();
130       }
131       const fetchTabContainer = document.createElement('div');
132       fetchTabContainer.style.color = 'white';
133       fetchTabContainer.innerHTML = fetchInnerHTML;
134       sidebarDOMNode.append(fetchTabContainer);
135     }
136   }
137
138   // meant for event listener click on the mode box
139   function setModelContent() {
140     const sidebarDOMNode = document.querySelector('.sidebar');
141     if (sidebarDOMNode) {
142       for (const child of sidebarDOMNode.children) {
143         child.remove();
144       }
145       const fetchTabContainer = document.createElement('div');
146       fetchTabContainer.style.color = 'white';
147       fetchTabContainer.innerHTML = modelInnerHTML;
148       sidebarDOMNode.append(fetchTabContainer);
149     }
150   }
151
```

Figura 40: callbacks of click event listener at <GLFirstBlock>

Here we query the sidebar's div to insert content. That content is at *static/model_innerHTML.js* and similar:

```
static > sytwc-threejs > model_innerhtml.js > modelInnerHTML
  1  export const modelInnerHTML = `
  2  <div>
  3    <h1>Descargas</h1>
  4    <a href="/model_box.glb">Descargar el modelo model_box</a>
  5    <a href="/animation1_emission.glb">Descargar la escena entera</a>
  6    <hr>
  7    <h1>Estructura de los Datos</h1>
  8
  9    <h2>Dimensiones</h2>
 10
 11    <h3>Employee</h3>
 12    <ul>
 13        <li>Employee id.</li>
 14        <li>Name</li>
```

Figura 41: static/model_innerhtml.js. See at https://github.com/alu0101056944/sytwc-threejs/blob/main/static/sytwc-threejs/model_innerhtml.js

It just exports the dom nodes to insert as string. All are imported directly from static/:

```
 1  import * as React from 'react';
 2
 3  import ContentAndSidebar from './content-and-sidebar';
 4
 5  import * as three from 'three';
 6  import { GLTFLoader } from 'three/addons/loaders/GLTFLoader.js';
 7
 8  import { fetchInnerHTML } from '../../static/sytwc-threejs/fetch_innerhtml';
 9  import { modelInnerHTML } from '../../static/sytwc-threejs/model_innerhtml';
10
11  function setupScene() {
12    const VIEWPORT_WIDTH = 500;
```

Figura 42: Imports of <GLFirstBlock>

Then when assigned to the corresponding div's *innerHTML*, the browser renders will render it properly. And because this can be called many times then the parent dom node needs to be cleared each time, thats why *sidebarDOMNode*'s children are removed each time a click happens.

The strings also contain links for downloading scene models.

Thats the gist of <GLFirstBlock>.

<GLSecondBlock>

This one is simpler than <GLFirstBlock>. It simply displays a sample helmet model and adds in ThreeJS's Orbit controls to be able to rotate around the model. And because there is no interaction besides that with this one, then the sidebar's content is added directly:

```
65  const SecondGLBlock = () => {
66    React.useEffect(setupScene, []);
67
68    React.useEffect(() => {
69          const sidebarContentDiv =
70              document.querySelector('.sidebarContent2');
71          sidebarContentDiv.innerHTML = kpiInnerHTML;
72        }, []);
73
74    return (
75      <ContentAndSidebar
76          key='content2'
77          sidebarContent={
78              <div className='sidebarContent2'>
79                {/* The content is inserted here by JS */}
80              </div>
81            }>
82        <div className='glBlock2'>
83          {/* The canvas is attached here */}
84        </div>
85      </ContentAndSidebar>
86    );
87  }
```

Figura 43: <SecondGLBlock> main function

The second *useEffect* is the one inserting the content into the right sidebar.