Figure 10: Front body

Figure 11: Back body

Figure 12: Body bird view

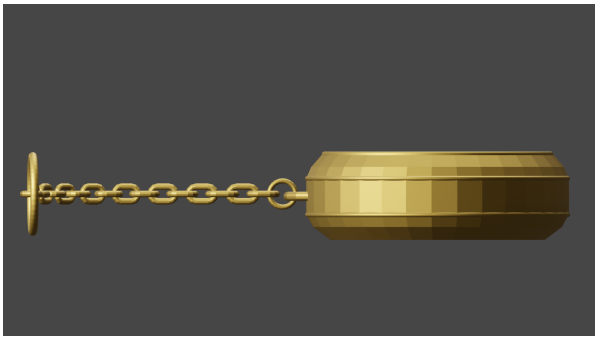

Figure 13: Body side view

ii. The chain

> During the Hackathon we thought we wanted to showcase the Barometer, in a contemporary way, so we agreed on the idea of adding a chain to the device. The chain was created using a Torus and an Array Modifiers.

Figure 14: The chain

iii. The spring

> The spring, which has the function of opening from the privacy flap, the barometer itself, at the push of a button. It could be easily created in Blender. You start creating a circle, then add the "screw" modifier to the object. This creates a circular tube that can be shaped as desired. With the modifier menu you can increase the "Itaration". As soon as you have done this, you can already recognize a spring and, if necessary, process it further.

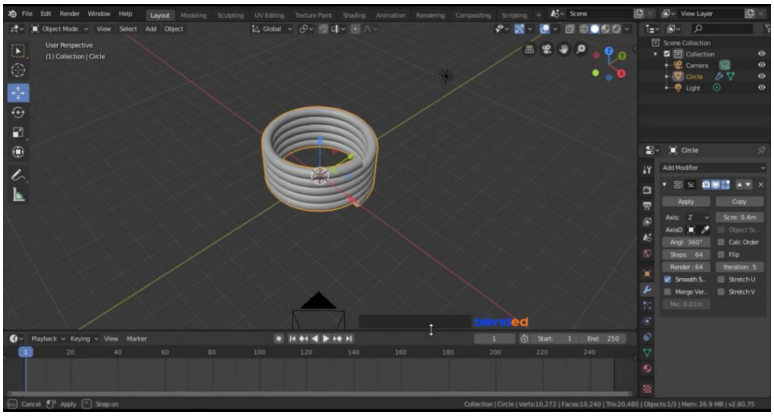

Figure 15: Modelling & Animation

iv. Insides

> From multiple sources we figured out the mechanism within the barometer. So we shaped a logical component unit from data which represents the insides. Multiple parts of that unit were animated seperately to represent the final mechanism. This was already done in mile 1.

Figure 16: Barometer insides in mile 1

iii. Mile 2

1. The chain was removed
2.

i. Case

Figure 17: Case open, black

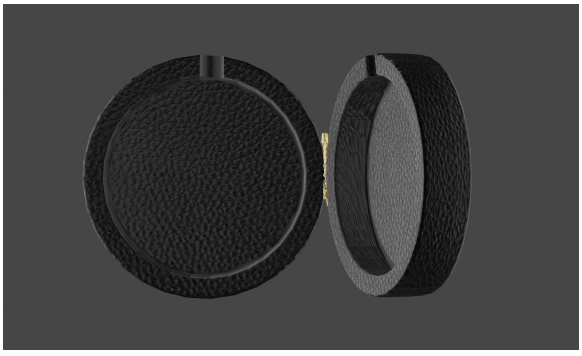

Figure 18: Case open, black, top view

Figure 19: Case open, black, side view

ii. Barometer

Figure 20: Finished barometer at the end of mile 2

iii. Insides

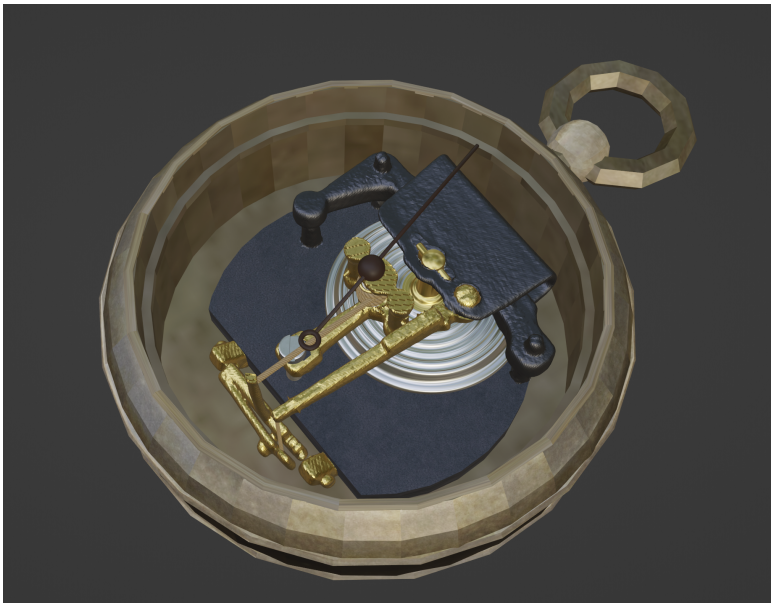

Figure 21: Barometer insides at the end of mile 2

iv. Textures

> The UV Atlas of the original Barometer were given by the DSM to texture the remodeled Barometer. Therefore the texture was UV mapped through the unwrap on the model.

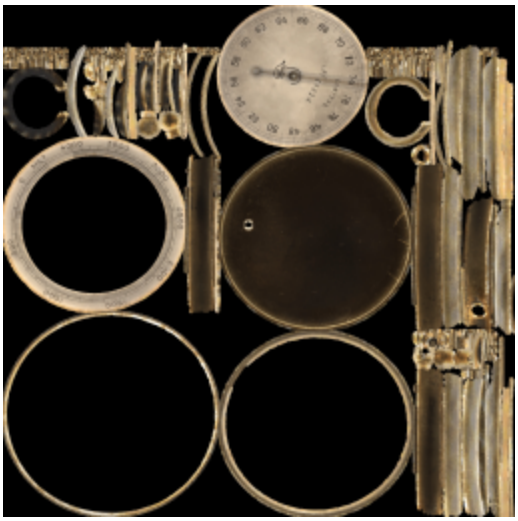

Figure 22: Texture use for barometer case in mile 2

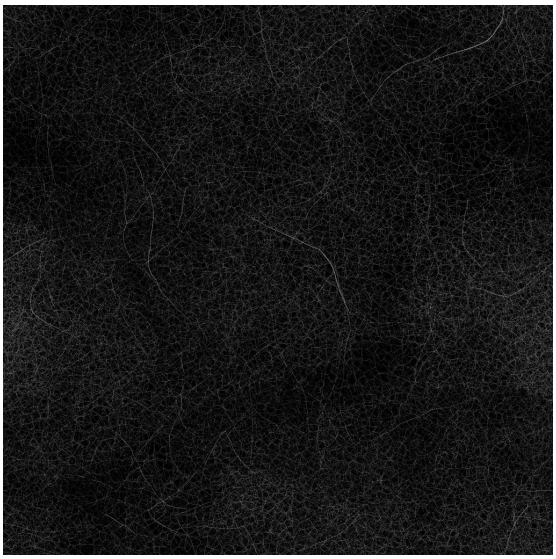

Figure 23: Leather texture for the case in mile 2

Figure 24: Texture used for barometer in mile 2

xvii. WEB-APP-USER-INTERFACE

i. Mile 1

i. GUI

Three different views should be implemented for the expanded interaction of the web application. A number was assigned to each of these views and thus made available directly under the representation of the respective component [ie the barometer case].

Figure 25: GUI mile 1

The clickable buttons are the outer rectangles with the arrow icons. The views switch automatically if one of the numbers is not pressed. Within a view there is a selection of animations that can be played. Further interaction options shall be implemented in the future.

ii. Mile 2

i. GUI

The graphical user interface evolves from a basic implementation to a more user friendly one suited for the purpose of an exhibition. We don’t have buttons with views anymore. Instead we have animations with their corresponding names placed as buttons in the middle of the UI. Framed by two arrows which make it possible for the user to choose between more animations than listed up front. The animations are ordered in sequence from left to right. A button to toggle x-ray vision on the displayed component was also added.

Figure 26: GUI mile 2

ℹ️ For **this milestone 2** we have one main component “The barometer in its case”.

- The button on the left is for x-ray mode which enables us to look inside the barometer so we can watch the mechanic there.
- The button on the right is the “reset” button which enables us to bring the barometer back to its starting point of presentation.
- The arrow buttons enables us to switch between animations.
- In the middle of the UI you find all different animations available.
- Light-grey colored are the buttons that are currently pressed.
- You click once on a certain animation available for a particular component which is in the present mile 2 the whole barometer. When you press the same animation name again, it shall be played in reverse.

ii. Optimized for different screen sizes

In mile 2 the GUI was optimized for different screen sizes. This was done with multiple predefined canvas which swap accordingly to the available screen automatically.

Figure 27: GUI in screensize XS

Figure 28: GUI in screensize XS with hint

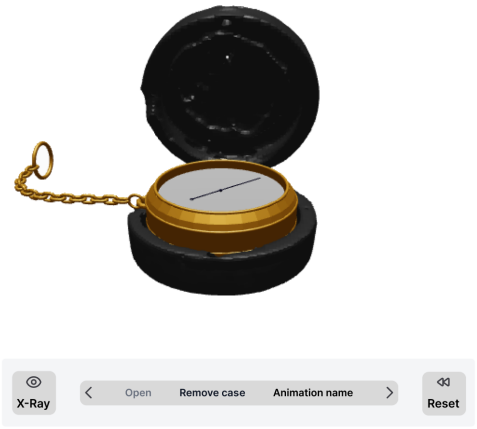

Figure 29: GUI in screensize for desktops

xviii. CODE

In this project we are making use of WebGL library three.js, CSS framework tailwind.css and bundler Webpack. For the list of all node packages used in the project see `package.json`

ii. Mile 1

i. Basic setup

ii. Load model

iii. Play the animation

iii. Mile 2

i. New application architecture

The need of a new architecture (Fig. 1) became evident as we started to develop the graphic user interface (GUI), further referred to as dashboard. Splitting the app into components makes it easier to scale and maintain the software. New design suggested that we will need at least two components – the 3D scene, containing HTML5 canvas and the dashboard, holding buttons and button carousel. Furthermore, all buttons and carousel could also be implemented as components, but we had to reject it for now since the effort would outweigh the use at this stage of project.

Figure 30: Wireframe schematics of the application

ii. Entry point

The entry point of the web application is by convention `script.js`:

```
import './style.scss'
import { scene, dashboard } from './layouts'

scene.loadScene()
dashboard.loadDashboard()
```

Here we are enqueue our styles and two main app components – 3D scene containing the barometer model and a dashboard, then instantiate a single object of each respective components.

iii. Scene component

The scene component is defined in `src/layouts/scene.js` as a class `Scene` exported through dynamic interface in `src/layouts/index.js`. Before defining the class we make sure to leverage three.js and stats.js modules:

```
import * as THREE from 'three'
import { OrbitControls } from 'three/examples/jsm/controls/Orbi
import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoade
import { DRACOLoader } from 'three/examples/jsm/loaders/DRACOLoa
import Stats from 'stats.js'
```

This class consists of a single method `loadScene()` which contains 3D scene setup, GLTF loader inclusive DRACO decompressor, method to keep track of time and update scene objects and a method to initialise several event listeners connected to dashboard.

Selecting document objects and global variable declaration

```
// Select dashboard buttons
const xray_button = document.querySelector('#xray')
const reset_button = document.querySelector('#reset')

const open_button = document.querySelector('#open_case')
const remove_case_button = document.querySelector('#remove_case
const action_button = document.querySelector('#in_action')
const liftup_button = document.querySelector('#lift_up')

// Select canvas
const canvas = document.querySelector('canvas.webgl')

// Define global objects
let model, animation_clips, mixer, open_case_action, lift_up_act
let etui_parts = []
```

```
// X-Ray Shader variables
let uniforms = {
                        isXRay: { value: false },
            rayAng: { value: 0.975 },
            rayOri: { value: new THREE.Vector3() },
            rayDir: { value: new THREE.Vector3() }
            }

let raycaster = new THREE.Raycaster()
let mouse = new THREE.Vector2()
```

Optional statistics

Stats.js is a small library which helps with performance evaluation and should be removed in final product. We create and update an object `stats` to plot some parameters like frame-per-second or latency of our scene (Fig.31)

```
// Stats
const stats = new Stats()
stats.showPanel(0)
document.body.appendChild(sta
```

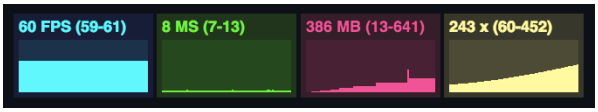

Figure 31: Graphic interface of Stats.js

Basic scene components

```
// Scene
const scene = new THREE.Scene()

// Base camera
const camera = new THREE.PerspectiveCamera(60, sizes.width / siz
camera.position.x = 0
camera.position.y = 1.5
```

```
camera.position.z = 4
scene.add(camera)

// Renderer
const renderer = new THREE.WebGLRenderer({
            canvas: canvas,
            antialias: true,
            alpha: true
      })
 renderer.setClearColor(0xfbfbfb, 1)
 renderer.setSize(sizes.width, sizes.height)
 renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
 renderer.outputEncoding = THREE.sRGBEncoding
 renderer.toneMapping = THREE.ACESFilmicToneMapping
 renderer.toneMappingExposure = 1.2
 renderer.shadowMap.enabled = true
 renderer.shadowMap.type = THREE.PCFSoftShadowMap
```

Resizing

```
const sizes = {
            width: window.innerWidth,
            height: window.innerHeight
        }

window.addEventListener('resize', () => {
            // Update sizes
            sizes.width = window.innerWidth
            sizes.height = window.innerHeight

            // Update camera
            camera.aspect = sizes.width / sizes.height
            camera.updateProjectionMatrix()

            // Update renderer
```

```
            renderer.setSize(sizes.width, sizes.height)
            renderer.setClearColor(0xfbfbfb, 1)
            renderer.setPixelRatio(Math.min(window.devicePixelRa
        })
```

Lights

The scene is illuminated by two types of light – low-intensity white ambient light and a brighter directional light, pointing to the object. As seen in code below, we redefined certain properties of a directional light, to reduce the shadow scope of the light, therefore slightly improving performance.

```
// Lights
    // ambient
  scene.add(new THREE.AmbientLight(0xffffff, 0.1));

  //directional
  const parametersDirLight = {
            color: 0xffffff,
            intensity: 1.5
        }
  const directionalLight = new THREE.DirectionalLight(parameters
  // Set up shadow properties for the light
  directionalLight.receiveShadow = false
  directionalLight.shadow.mapSize.width = 128
  directionalLight.shadow.mapSize.height = 128
  directionalLight.shadow.camera.near = 0.5
  directionalLight.shadow.camera.far = 1
  scene.add(directionalLight)
```

Orbit controls

```
// Controls
const controls = new OrbitControls(camera, canvas)
controls.enablePan = false
controls.autoRotate = true
```

```
controls.autoRotateSpeed = 1
controls.enableDamping = true
controls.dampingFactor = .5
controls.minDistance = 1.5
controls.maxDistance = 3
```

EventListeners - events() method

```
// let case_model = null
        let clips = null
        let mixer = null
        // let hingeAction = null


        // Model Loader

        // Instantiate DRACO Loader
        const dracoLoader = new DRACOLoader()
        dracoLoader.setDecoderPath('/draco/')

        // Load model with GLTF Loader
        const gltfLoader = new GLTFLoader()
        gltfLoader.setDRACOLoader(dracoLoader)
        gltfLoader.load('/models/last_textured_baro.glb', (gltf
            console.log(gltf)
            scene.add(gltf.scene)
            clips = gltf.animations
            mixer = new THREE.AnimationMixer(gltf.scene)

            // finally, start the rendering
            tick()

            // Debug
            // folderObject.add(gltf.scene.children[35].materia
```

```
        })
```

Here we are still in the class Scene and within its loadScene() method. We instantiated the dracoLoader for decoding purpose and as nescessary requirement to build the GLTF Loader which is needed to load our blender files in .glb format into the program with the use of its threejs framework. The method tick() does the rendering.

```
/**
* Animate
*/
const clock = new THREE.Clock()
let lastElapsedTime = 0

const tick = () => {

     // Clock
     const elapsedTime = clock.getElapsedTime()
     const deltaTime = elapsedTime - lastElapsedTime
     lastElapsedTime = elapsedTime

     // Update controls
     controls.update()

     // Render
     renderer.render(scene, camera)

    // Update the animation mixer
    if (mixer) mixer.update(deltaTime)

    stats.update()

    // Call tick again on the next frame
    window.requestAnimationFrame(tick)
```

```
}
```

ii. Dashboard component

Creation of a camera for the orbital view on the displayed barometer component.

Add an eventListener to the windwows object who is updating the camera inside the scene. The renderer does his prework on the windows and scene.

xix. BAROMETER

> This section contains object related details provided by the DSM.

i. Description

- Barometers are used to precisely measure the atmospheric air pressure so that information about the weather situation can be obtained.
- A barometer is a measuring device for determining the static absolute air pressure. The change in air pressure is measured.
- In particular, the use of the barometer makes it possible to recognize the short-term trend at the local level and to react to possible weather changes.
- Our model is the Vidi Barometer. Containing a can with vacuum.

i. Other

Observing air pressure with a barometer alone is not enough to reliably predict the weather. For this reason, a barometer is often used together with other measuring devices: an anemometer to determine wind speed, a thermometer to detect temperature development, a wind vane to determine wind direction or a hygrometer to measure humidity.

ii. Technique

- A can barometer consists of a metal can (pressure can) with minimal pressure inside. This dose is connected to a lever mechanism via a spring. The mechanism results in a pointer that can move in front of a scale.

- The pressure inside the dose, the pressure from the spring and the air pressure are in balance. At normal air pressure, the pointer is set to 1,013 hPa (760 Torr). If the air pressure changes, the membrane of the dose is deformed to a greater or lesser extent. The respective value of the air pressure can be read on the scale.
- The air pressure indicated by the barometer changes as the weight of the surrounding air changes. It is measured in hectopascals (hPa), sometimes in millibars (mb).

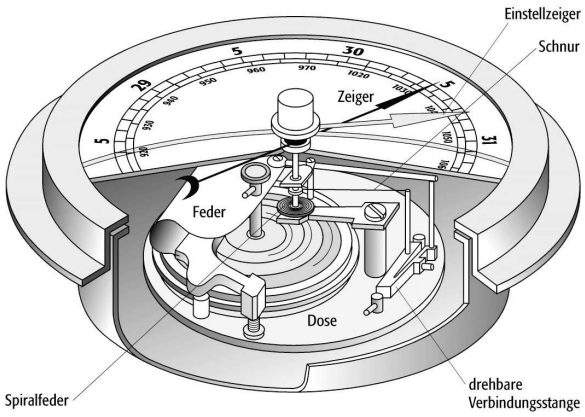

Figure 32: Mechanism inside

iii. Details

- Measurements: 2.8cm; Ø 5.3 cm (without case)
- Weight: 120g
- Material:
  - Barometer: Brass;
  - Case: leather, textile
- Clockface:
  - Outer ring inscribed 'MET' and descending scale from 4000 in 500 increments down to 0.
  - Inner ring: with a scale from 46 (at approx. 2 o'clock) to 78 in increments of 2.

xxi. PROBLEMS & OPTIONS & SOLUTIONS

Many problems appeared during the process of development. The questions it gave us and the solutions we chose are collected and represented in these tables.

Figure 33: Managing problems around modeling

Modeling	**Mile 0**	**Mile 1**	**Mile 2**
Problem(s)	Raw data	Model file size	The 3D model was running very slow on the Three.js Application
Options /Details	a) Don’t make use of given data at all. b) Try to make a use of it somehow.	a) Choose another file format. b) Delete less important components. c) Don’t use native objects given from raw-data.	1. Removal of components. 2. Reducing the number of Geometry on the Mesh.
Solution(s)	Seperate selected objects from sticky polygon mass and construct components from it.	1. Reducing polygons with tools given by Blender. 2. Recreated certain chosen component parts from scratch.	More components had to be recreated from scratch completely. 1. Using Instant Meshes software for Retopology of some components. 2. Using the initial scan for reference remodel the original ring 3. Removing of the chain and sticking to the original form.

iii. App architecture

Figure 34: Managing problems around animation

Animation	**Mile 1**	**Mile 2**
Problem(s)	Animations couldn’t be well implemented on certain native models	Animations couldn’t be found during phase of implementation into web-app.

Options / Details	a) Remodel the barometer components from scratch inside blender discarding the raw data from the DSM. b) Don’t animate certain models.	Renaming the Barometer objects that have Animations, and mentioning that they have animations in their title i.e. Animation_open_case / Animation_Lift up /Animation_Measuring
Solution(s)	Refurbished or remodeled each problematic components parts so they can be used for animation.	Rename animations inside blender to successfully import into the web-application.

Figure 35: Managing problems around texture

Texture	**Mile 0**	**Mile 1**	**Mile 2**
Problem(s)		Adding the colors on the Remodeled Barometer	1. X-Ray. 2. Real Barometer Texture
Options / Details			1. Make and X-Ray Shader. 2. UV unwrap the object to the Original UV Atlas.
Solution(s)			Retexture

Figure 36: Managing problems around webtransfer

Webtransfer	**Mile 1**	**Mile 2**
Problem(s)	Web application loading times, FPS, loading Blender file formats into the web.	Where to apply the lighting and shadows?
Options / Details		1. Inside Blender 2. Inside web application.
Solution(s)		Shadows and lightning inside web application using three.js framwork.

xxii. TESTINGS

i. White-box & black-box testing

i. Mile 1

Figure 37: White-box & black-box testing for mile 1

Aspects tested	Prototype	Iteration 1	Iteration 2	Iteration 3
Animations working on components	-	-	high	low
Visualizing animations for a presentation	high	low	high	low
Rotation of the component through an animation	high	low	low	-

ii. Mile 2

Figure 38: White-box & black-box testing for mile 2

Aspects tested	Prototype	Iteration 1	Iteration 2	Iteration 3
Animations working on components	low	medium	low	-
Animations import into application	high	medium	low	-
Components centered	high	low	low	-
Barometer component import into application	high	high	-	-
Rotation of the component through application	high	low	-	-

Lighting- & shadow effects	high	-	-	-
X-ray-vision	high	low	low	-

xxiii. CONCLUSION

This document describes the development of a web UI for a barometer using Webpack and Three.js. It covers the project's milestones, including basic setup, loading the model, and playing the animation. The document also discusses the new application architecture, entry point, scene component, dashboard component, and problems encountered during development. Finally, the document includes testing results and a conclusion.

The document provides detailed descriptions of the code for each component, including selecting document objects and global variable declaration, optional statistics, basic scene components, resizing, lights, orbit controls, and event listeners.

The developers encountered problems during the development process, including issues with modeling, animation, texture, and web transfer. They considered various options and chose the best solutions to address each issue.

The testing section of the document includes white-box and black-box testing results for each iteration of the project. The results show that some aspects of the project, such as animations and visualization, improved over time, while others, such as importing animations into the application, remained a challenge.

Overall, the document provides a comprehensive overview of the development process and the challenges encountered along the way. It demonstrates the importance of careful planning, problem-solving, and testing in creating a successful web UI.