Technische Software-Dokumentation (Readme)

Machbarkeitsstudie für die Nutzung von 3D Rendering im Web für die Auswahl von Veranstaltungsräumen

Datum: 17.02.2022

Von: Vladimir Brazhnik (vladimir.brazhnik@ost.ch)

Inhaltsverzeichnis

Bemerkung
In dieser Software-Dokumentation wird ausschliesslich zugunsten der besseren Lesbarkeit auf eine geschlechtsspezifische Formulierung von personenbezogenen Bezeichnungen verzichtet. Die maskuline Form soll sich stets gleichermaßen auf männliche, weibliche und diverse Personen beziehen.

1 Vorwort

Das Repository zeigt eine Machbarkeitsstudie der Nutzung von 3D Rendering im Web für die Auswahl und Konfiguration von Veranstaltungsräumen anhand des Beispiels „KKL Luzern“. Für die Realisierung des interaktiven 3D Modelles von dem Gebäude und dessen konfigurierbaren Räumen wurden die Packages `React-Three-Fiber` und `React-Drei` verwendet. `React-Three-Fiber` ist ein Renderer, welcher die Verwendung von `ThreeJS` Objekten in React deutlich vereinfacht, indem diese bereits als vorgefertigte Komponenten verwendet werden können. `React-Drei` bietet zusätzlich eine Kollektion an nützlichen Helfer-Klassen, Funktionen und bereits voll funktionalen, vorgefertigten Abstraktionen für `React-Three-Fiber`. Beide Packages sind speziell für React entwickelt worden und bauen auf `ThreeJS` und der `WebGL` auf. Das 3D Modell wurde mit Hilfe der 3D-Computergrafik Software Blender entwickelt und reagiert in Echtzeit auf die Auswahl und Konfigurierung der Räume im 3D Modell oder auch in der Raumliste. Zusätzlich wurde mit `MeshMaterials`, deren `color` und `opacity` Werten gearbeitet (Kapitel 6.5.1), als auch eine fortgeschrittene Logik einer für jeden Raum individuelle Kamerafahrt entwickelt (Kapitel 6.6).

2 Installation

1. Das Repository klonen
   `git clone https://github.com/4realDev/KKL-3D-Prototype.git`

2. NPM packages installieren
   `npm install`

3. Das Repository lokal ausführen unter http://localhost:3000/
   `npm run start / yarn start`

4. Das Repository bauen
   `npm run build / yarn build`

3 Debugging

Im Projekt wurden während der Entwicklung einige Debug Optionen im UI entwickelt, welche mit Hilfe von Checkboxen aktiviert und deaktiviert werden können (Abbildung 1). Diese Debug Optionen werden durch die Klasse `DebugControlPanel.tsx` gesteuert. Falls diese nicht benötigt wird und ausblendet werden soll, muss die `Zeile 178-180` in `RoomSelection.tsx` auskommentiert werden (Code Ausschnitt 1).

```
<div className={styles.card}>
    <DebugControlPanel />
</div>
```

Code Ausschnitt 1: Initialisierung von der DebugControlPanel.tsx Komponente in Zeile 178-180 in RoomSelection.tsx

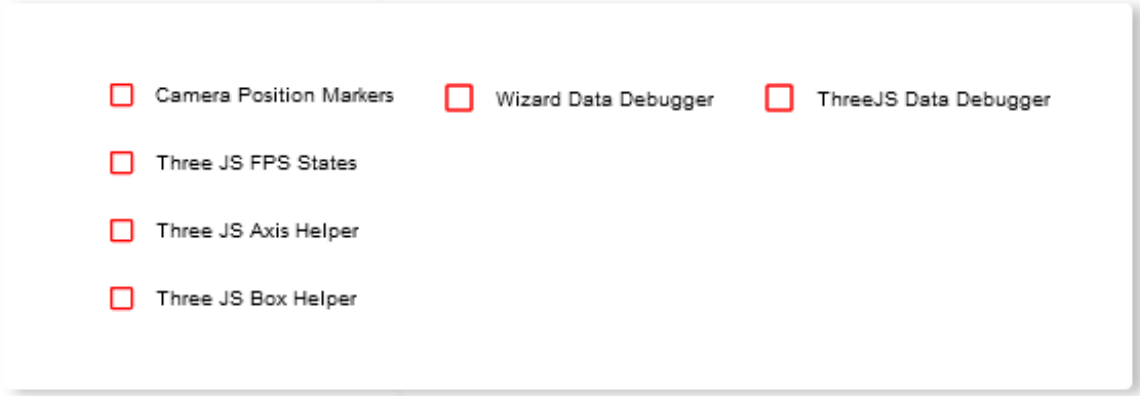


Abbildung 1: DebugControlPanel mit Checkboxen zum Aktivieren und Deaktivieren von Debug Optionen im UI der Applikation

3.1 Debug Optionen

3.1.1 ThreeJS Data Debugger

`ThreeJsDataDebugger.tsx`: Zeigt Informationen, welche in `useCameraStore.ts` als States gehalten werden und global verwendet und angepasst werden können in einer Card Komponente in Echtzeit. Die Abbildung 2 zeigt, wie diese Informationen im Modell dargestellt werden.

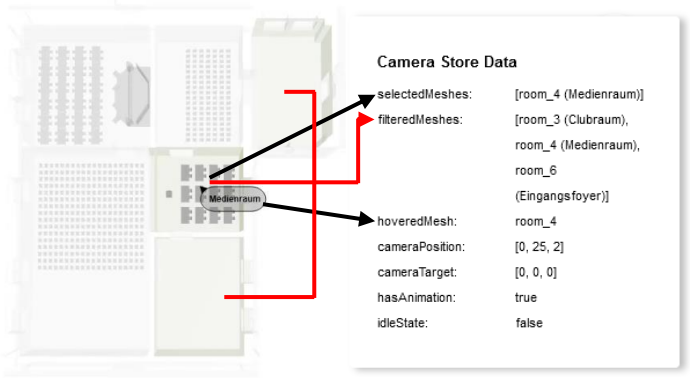


Abbildung 2: Gegenüberstellung von einem Modell Zustand und dessen useCameraStore Daten

3.1.2 Wizard Data Debugger

`WizardDataDebugger.tsx`: Zeigt die Informationen, welche in `useWizardStore.ts` als States gehalten, global verwendet und angepasst werden können in einer Card Komponente in Echtzeit. Die Abbildung 3 zeigt, wie diese Informationen im Wizard dargestellt werden.

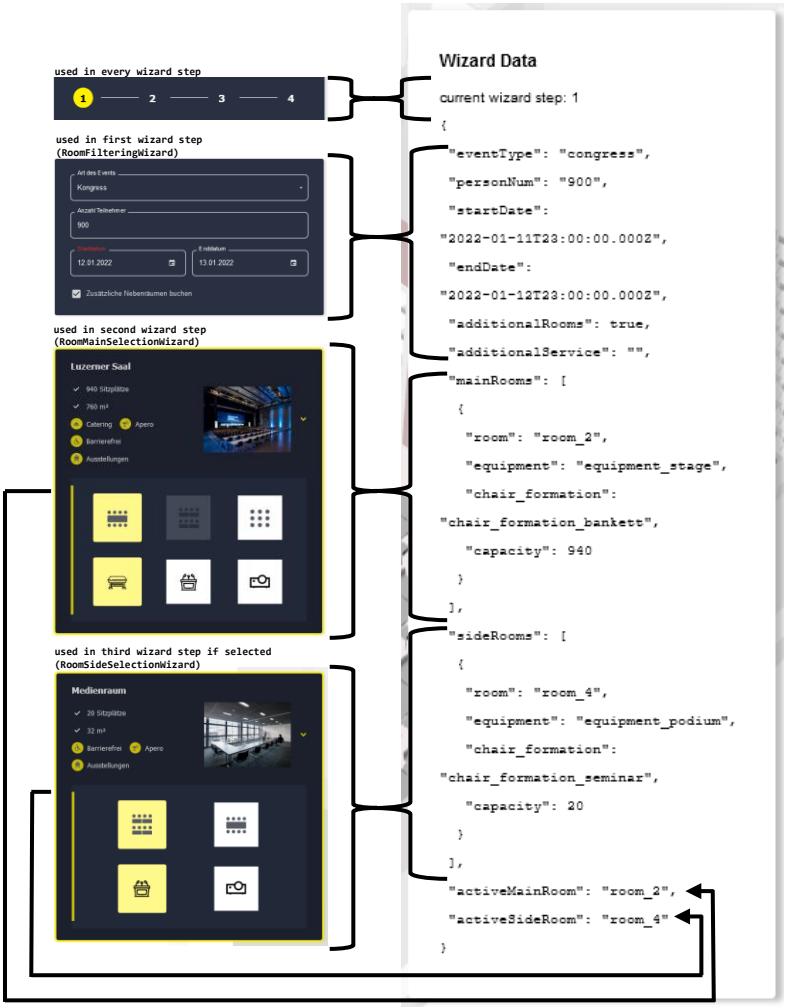


Abbildung 3: Gegenüberstellung von unterschiedlichen Wizard Screens, deren Schritten und den useWizardStore Daten

3.1.3 3D Modell Debugger

1. **`CameraPositionMarkers.tsx`**: Rendert die Kamera Positionen als schwarzes Rechteck und deren Fokus Punkt als orangene Kugeln für die `camPos` und `camTarget` Werte jedes Raumes (Abbildung 4). Das ist hilfreich bei der Konfigurierung der einzelnen, individuellen Kameraansichten / Nahansichten der Räume, wenn sie ausgewählt sind.

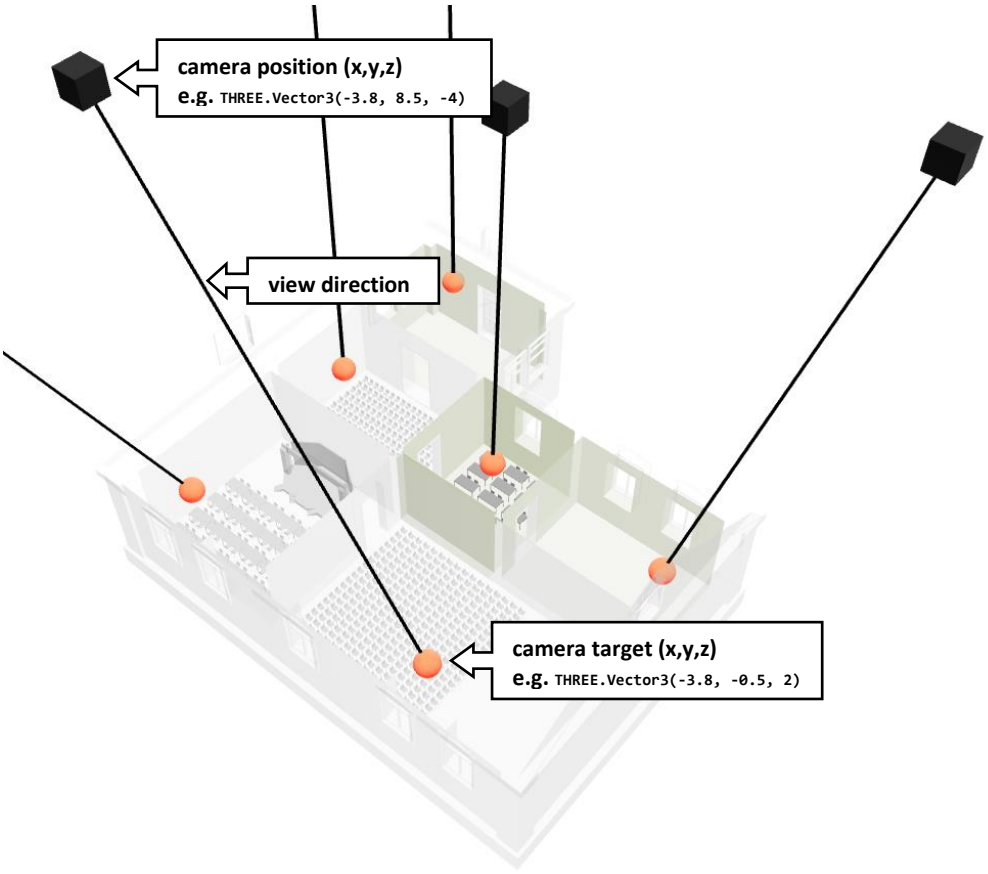
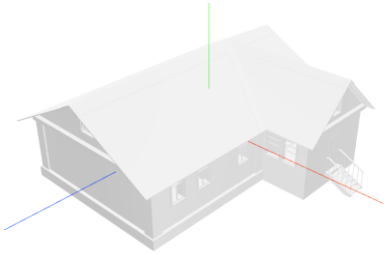


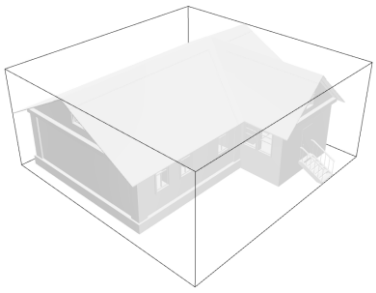
Abbildung 4: 3D Modell mit aktiven CameraPositionMarkers

2. **ThreeJS Axis Helper**:

Zeigt die einzelnen Achsen im Koordinatensystem an (untere Abbildung). Hilfreich zur Erkennung, wo der Mittelpunkt im Koordinatensystem ist.

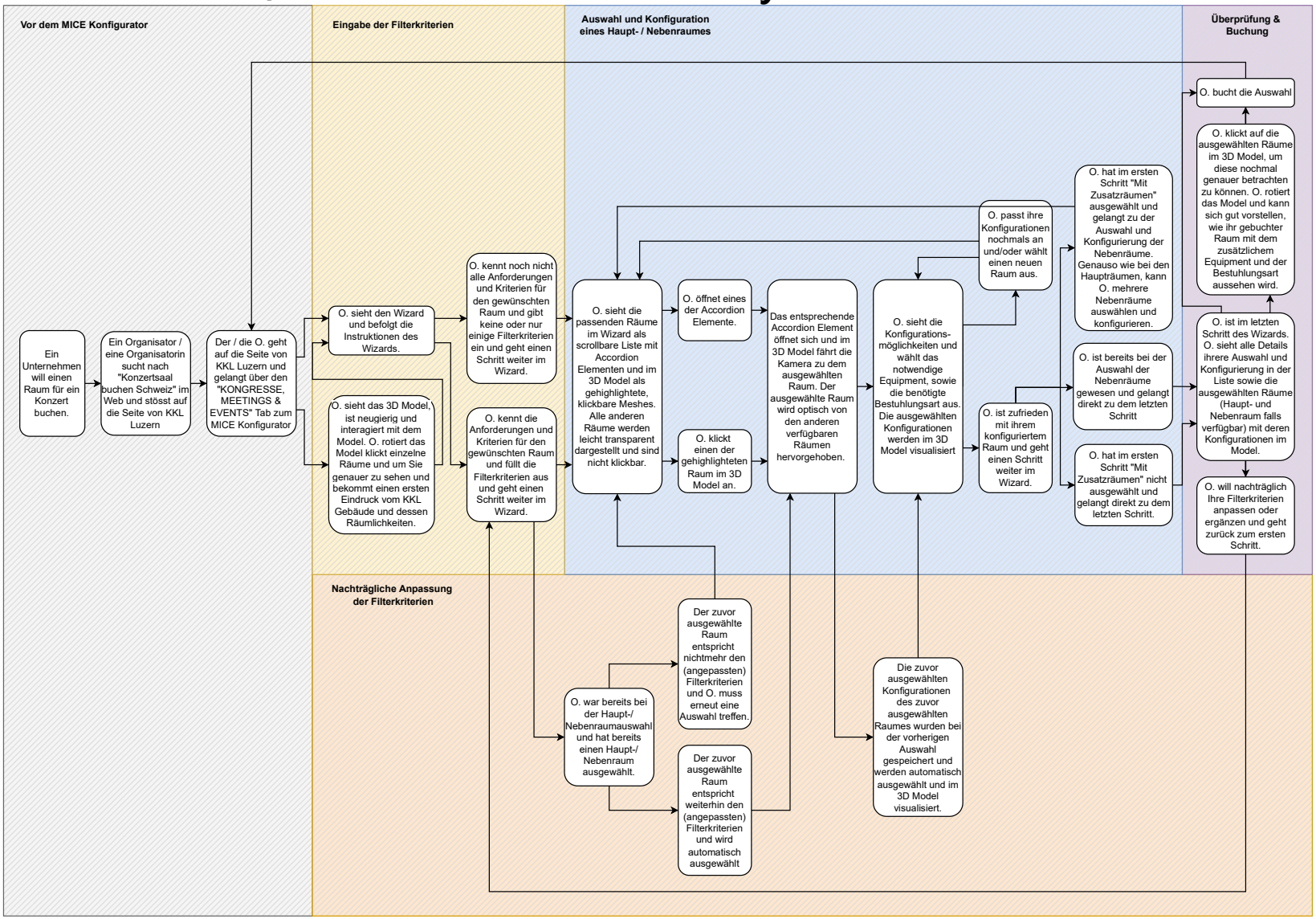


3. **ThreeJS Box Helper**:
Zeigt eine Bounding Box um das 3D Modell an (untere Abbildung). Hilfreich zur Einschätzung, wie gross das Modell ist.



4. **ThreeJS FPS States:**
Zeigt eine einfache Infobox in der linken, oberen Ecke an, mit der die Leistung des ThreeJS Codes überwacht werden kann (untere Abbildung). Auch bei häufigem und schnellem Wechseln, Rotieren und Konfigurieren des Modelles fiel die FPS Rate nicht unter 40 frames per second.

4 Vereinfachter, Technischer Customer Journey



5 Aufbau und Filestruktur

- In **`public`** sind die Bilder als komprimierte JPEG Files, das 3D Modell als GLB File und die im readme verwendeten Dateien hinterlegt.
- In **`src`** befinden sich die einzelnen Komponenten unter `components` die unterteilt sind in `blocks`, `debug`, `icons`, `threeJs`, `ui` und `wizard`.
  - **`blocks`** sind die beiden Blöcke, aus denen die Applikation aufgebaut ist. Diese sind: `RoomSelection.tsx`, in welcher sich die gesamten Komponenten von `debug`, `icons`, `ui` und `wizard` befinden und `ModelCanvas.tsx`, welche die ThreeJs Logik in sich trägt, die Canvas, die Kamera und das 3D Modell initialisiert und somit nur die `threeJs` Komponenten verwendet.
  - **`debug`** sind die Komponenten, welche für das eigene Debugging verwendet wurden.
  - **`icons`** sind alle `<svg>` Icons, die in dem gesamten Projekt verwendet wurden als React Arrow Function Komponenten.
  - **`ui`** sind die kleineren User Interface Komponenten, die im Wizard und in der Raumliste eingesetzt wurden.
  - Im **`wizard`** befinden sich die vier Wizard Screens, welche abhängig von dem aktuellen Wizard Schritt gerendert werden und die `ui` und `icons` Komponenten nutzen.
- In **`data`** befindet sich das statische Datenmodell, das die geladenen Daten vom Backend mocken soll.
- In **`hooks`** befinden sich zwei wiederverwendbare Hooks, welche zum einen dabei helfen, langen Press-Events zu erkennen und zum anderen die Ermittlung der Viewport Breite im Code zu erleichtern.
- In **`store`** befinden sich alle global zugänglichen States und deren Methoden. Diese sind unterteilt nach den Wizard Daten, den Kameradaten, den Meshdaten und den (optionalen) Debuggingdaten.
- In **`styles`** befinden sich die globalen `colors.scss` und die `viewport.scss` Variablen.
- In **`utils`** befinden sich global geteilte Getter Methoden für Raumdaten und deren Mappings.

```
.
│   .gitignore
│   package-lock.json
│   package.json
│   tsconfig.json
│   yarn.lock
│
├──public
│   │   favicon.ico
│   │   index.html
│   │   logo192.png
│   │   logo512.png
│   │   manifest.json
│   │
│   ├──images
│   │       Auditorium.jpg
│   │       BusinessMedienraeume.jpg
│   │       Clubraeume.jpg
│   │       EingangsFoyer.jpg
│   │       Konzertsaal.jpg
│   │       LuzernerSaal.jpg
│   │
│   └──model
│           house-model.glb
│
└──src
    │   App.js
    │   App.scss
    │   index.css
    │   index.js
    │   react-app-env.d.ts
    │   theme.tsx
    │
    ├──components
    │   ├──blocks
    │   │   ├──ModelCanvas
    │   │   │       ModelCanvas.module.scss
    │   │   │       ModelCanvas.tsx
    │   │   │
    │   │   └──RoomSelection
    │   │           RoomSelection.module.scss
    │   │           RoomSelection.tsx
    │   │
    │   ├──debug
    │   │   ├──DebugControlPanel
    │   │   │       DebugControlPanel.module.scss
    │   │   │       DebugControlPanel.tsx
    │   │   │
    │   │   ├──ThreeJsDataDebugger
    │   │   │       ThreeJsDataDebugger.module.scss
    │   │   │       ThreeJsDataDebugger.tsx
    │   │   │
    │   │   └──WizardDataDebugger
    │   │           WizardDataDebugger.module.scss
    │   │           WizardDataDebugger.tsx
    │   │
```

```
│   ├──icons
│   │       Accessibility.tsx
│   │       Apero.tsx
│   │       Beamer.tsx
│   │       Catering.tsx
│   │       ChairFormationBankett.tsx
│   │       ChairFormationConcert.tsx
│   │       ChairFormationSeminar.tsx
│   │       CheckMark.tsx
│   │       Chevron.tsx
│   │       Edit.tsx
│   │       EmptySearch.tsx
│   │       Exhibition.tsx
│   │       NoSeats.tsx
│   │       Notification.tsx
│   │       Overview.tsx
│   │       Podium.tsx
│   │       Seats.tsx
│   │       Stage.tsx
│   │
│   ├──threeJs
│   │       CameraControls.tsx
│   │       CameraPositionMarkers.tsx
│   │       Lights.tsx
│   │       Model.tsx
│   │
│   ├──ui
│   │   ├──Accordion
│   │   │       Accordion.module.scss
│   │   │       Accordion.tsx
│   │   │
│   │   ├──AccordionItem
│   │   │       AccordionItem.module.scss
│   │   │       AccordionItem.tsx
│   │   │
│   │   ├──Cursor
│   │   │       Cursor.tsx
│   │   │
│   │   ├──EditButton
│   │   │       EditButton.module.scss
│   │   │       EditButton.tsx
│   │   │
│   │   ├──MeshVisibilityButton
│   │   │       MeshVisibilityButton.module.scss
│   │   │       MeshVisibilityButton.tsx
│   │   │
│   │   ├──NoResults
│   │   │       NoResults.module.scss
│   │   │       NoResults.tsx
│   │   │
│   │   └──RoomCard
│   │           RoomCard.module.scss
│   │           RoomCard.tsx
│   │
│   └──wizard
│       ├──RoomFilteringWizard
│       │       RoomFilteringWizard.module.scss
│       │       RoomFilteringWizard.tsx
│       │
│       ├──RoomMainSelectionWizard
│       │       RoomMainSelectionWizard.tsx
│       │
│       ├──RoomSideSelectionWizard
│       │       RoomSideSelectionWizard.tsx
│       │
│       └──RoomSummaryWizard
│               RoomSummaryWizard.module.scss
│               RoomSummaryWizard.tsx
│
├──data
│       roomData.ts
│
├──hooks
│       useLongPress.ts
│       useWindowDimensions.ts
│
├──store
│       useCameraStore.ts
│       useDebugStore.ts
│       useMeshStore.ts
│       useWizardStore.ts
│
├──styles
│       colors.scss
│       viewport.scss
│
└──utils
        room.tsx
```

6 Issues & Lösungen

6.1 Mocken der Raum- und Modelldaten (statisches Datenmodell)

Da der Prototyp ohne eine Backend oder Server Anbindung entwickelt wurde, mussten alle Daten, welche im Normalfall vom Backend oder Server geladen werden, im Projekt auf der Client Seite „gemocket“ werden. Hierzu wurde speziell die Datei `roomData.ts` in dem `src/data` Verzeichnis angelegt. Diese Datei exportiert die Konstante `roomList`, welche ein Array mit `RoomFetchedDataType` JSON Objekten, wie im Code Ausschnitt 2 zu sehen ist. Diese `RoomFetchedDataType` Objekte spiegeln die Informationen vom Raum (`roomList[index].info`) sowie die relevanten Informationen für die 3D Visualisierung im Mesh (`roomList[index].model`) wieder. Im Folgenden werden die einzelnen Properties vom `info` und `model` erleutert:

```
export type RoomFetchedDataType = {
    model: {
        meshName: INTERACTABLE_MESH_NAMES;
        camPos: THREE.Vector3;
        camTarget: THREE.Vector3;
    };
    info: {
        title: string;
        personCapacity: number | [number, number];
        area: number;
        img: string;
        chairFormations?: { name: CHAIR_FORMATION; capacity: number }[];
        equipment?: EQUIPMENT[];
        fittings?: ROOM_FITTINGS[];
        fittingEventTypes?: EVENT_TYPES[];
        fittingSideRooms?: INTERACTABLE_MESH_NAMES[];
        bookedDates: { start: string; end: string }[];
    };
};
```

Code Ausschnitt 2: RoomFetchedDataType Interface

- **`model.meshName`**: ist ein `INTERACTABLE_MESH_NAMES` Enum, welches denselben Namen als String trägt, wie das mit dem jeweiligen Raum korrespondierende 3D Mesh im glTF-File. Der `model.meshName` ist somit ein wichtiger Verbindungspunkt zwischen dem glTF 3D Modell und der React Applikation. Mehr dazu im Kapitel 6.3.3

- **`model.camPos`**: ist ein `THREE.Vector3` Objekt, welches aus den x, y und z Koordinaten besteht und die raumspezifische Kameraposition bestimmt. Mehr dazu im Kapitel 6.6.

- **`model.camTarget`**: ist ein `THREE.Vector3` Objekt, welches aus den x, y und z Koordinaten besteht und die raumspezifische Kamera-Blickrichtung / das Kameraziel bestimmt. Mehr dazu im Kapitel 6.6.

- **`info.title:`** ist ein `String`, welcher den Namen des Raumes angibt. Vor allem relevant für die Darstellung der Rauminformationen im UI.

- **`info.area:`** ist eine `Number`, welche die Fläche des Raumes angibt. Vor allem relevant für die Darstellung der Rauminformationen im UI.

- **`info.img:`** ist ein String, welcher die Quelle für das Bild des Raumes im `public/images` Ordner angibt. Vor allem relevant für die Darstellung der Rauminformationen im UI.

- **`info.bookedDates:`** ist ein `Array` mit einer oder mehreren `{„start“: String, „end“: String}` JSON Datenstrukturen. In diesem Array wird hinterlegt, von wann bis wann der Raum ausgebucht wurde. Diese Information ist die einzige Information in dem gesamten `roomList` JSON Objekt, welche nach jeder Buchung mit beispielsweise einem POST Request auf dem Server ergänzt und angepasst werden müsste. Vor allem relevant für die Filtrierung der Haupträume bei der Hauptraumauswahl und für die Filtrierung der Nebenräume bei der Nebenraumauswahl nach dem im ersten Schritt eingegebenen Start- und Enddatum des Events (`wizardData.startDate, wizardData.endDate`).

- **`info.personCapacity:`** kann entweder eine `Number` oder ein `Array` mit zwei `Numbers` sein und entspricht der Anzahl an Personen, die im Raum sitzend oder stehend (falls der Raum keine Sitzplätze hat) platziert werden können. Ist `personCapacity` ein `Array` aus zwei `Numbers`, so ist die erste `Number` die minimale Anzahl an Personen, welche im Raum Platz haben und die zweite `Number` die maximale Anzahl. Die Anzahl variiert abhängig von der ausgewählten Bestuhlungsart (`info.chairFormation`). Ist die `info.personCapacity` nur eine `Number`, so existiert entweder keine Bestuhlungsart (nur Stehplätze) oder nur eine fixe, nicht konfigurierbare Bestuhlungsart mit der in `info.personCapacity` angegebenen Anzahl an Sitzplätzen. Diese Information ist relevant für die Darstellung der Rauminformationen im UI sowie für die Filtrierung der Haupträume bei der Hauptraumauswahl. Das `info.personCapacity Array` existiert auch in den Nebenräumen (die Räume ohne `info.fittingSideRooms`), jedoch wird die Information momentan nur für die Darstellung der Rauminformationen im UI verwendet und nicht für die Filtrierung der Nebenräume bei der Nebenraumauswahl.

- **`info.fittings:`** ist ein optionales `Array` aus `ROOM_FITTINGS String Enums`, welche zusätzliche Informationen zu den Möglichkeiten der Räume gibt. Beispielsweise hat ein Raum mit Sitzplätzen im `info.fittings Array` das `Enum ROOM_FITTINGS.seats` und ein Raum ohne Sitzplätze das `Enum ROOM_FITTINGS.noSeats`. Diese Information ist vor allem relevant für die Darstellung der Rauminformationen im UI.

- **`info.fittingEventTypes:`** ist ein optionales `Array` aus `EVENT_TYPES String Enums`, welche zusätzliche Informationen zu den für den Raum passenden Eventtypen gibt. Beispielsweise hat der Konzertsaal in seinem `info.fittingEventTypes Array` das `Enum EVENT_TYPE.concert`. Diese Information ist relevant für die Darstellung der Rauminformationen im UI sowie für die Filtrierung der Haupträume im ersten Wizard Schritt.

  Die nächsten drei Properties `info.chairFormations, info.equipment` und `info.fittingSideRooms` folgen alle einer „Naming Convetion“, da diese neben der Verwendung im UI, auch als Referenz auf deren 3D Meshes in dem glTF File des 3D Gebäudes verwendet werden. Mehr dazu im Kapitel 6.3.

- **`info.fittingSideRooms:`** ist ein optionales `Array` aus `INTERACTABLE_MESH_NAMES String Enums`, welches angibt, ob der jeweilige Raum passende Nebenräume besitzt. Da jeder Hauptraum mindestens ein passenden Nebenraum besitzt und Nebenräume keine Nebenräume besitzen (aus den Daten der KKL Webseite), wird die Existenz von der `info.fittingSideRooms` Property auch zur Überprüfung des Raumtypen (Haupt- oder Nebenraum) verwendet.
  Gleichzeitig ist jeder `INTERACTABLE_MESH_NAMES` Wert im `Array` eine indirekte Referenz auf ein `RoomFetchedDataType` JSON Objekt in der `roomList`, da jeder verwendete `INTERACTABLE_MESH_NAMES` Wert mit einem `model.meshName` Wert eines `RoomFetchedDataType` JSON Objektes in der `roomList` korrespondiert. Dadurch ist es möglich den in `info.fittingSideRooms` referenzierten Raum in der `roomList` zu finden und dessen Informationen zu erhalten. Diese Information ist relevant für die Darstellung der Nebenräume im UI und im 3D Modell sowie für die Filtrierung der Haupträume im Wizard. Mehr dazu im Kapitel 6.2.1.

- **`info.equipment:`** ist ein optionales `Array` aus `EQUIPMENT String Enums`, das angibt, welches zusätzliche Equipment der jeweilige Raum besitzt. Auch diese `Enum Strings` folgenden der zuvor erwähnten Naming Convention, sodass die korrespondierenden 3D Meshes des raumbezogenen Equipments im 3D Modell gefunden und gegeben falls ein- und ausgeblendet werden können. Diese Information ist relevant für die Darstellung des Equipments im UI und im 3D Modell. Mehr dazu im Kapitel 6.3.5 und 6.4.

- **`info.chairFormations:`** ist ein optionales `Array` mit einer oder mehreren `{name: CHAIR_FORMATION, capacity: Number}` JSON Datenstrukturen, das angibt, welche zusätzliche Bestuhlungsarten der jeweilige Raum besitzt und wie viele Personen damit im Raum platziert werden können. Eine als `name` gewählte `CHAIR_FORMATION` ist hierbei genauso wie `EQUIPMENT` ein `String Enum`, welches auch den Naming Conventions folgen muss, sodass die korrespondierenden 3D Meshes der raumbezogenen Bestuhlungsarten im 3D Modell gefunden und gegebenenfalls ein- und ausgeblendet werden können. Diese Information ist relevant für die Darstellung der Bestuhlungsart im UI und im 3D Modell. Mehr dazu im Kapitel 6.3.5 und 6.4.

6.2 Konvertierung des glTF 3D Modelles in eine JSON Datenstruktur (dynamisches Datenmodell)

Der klassische und empfohlene Weg von der `React-Three-Fiber` Dokumentation ein glTF Modell in React zu laden, ist über das Kommandozeilen-Tool `GLTFJSX`. Dieses Tool wandelt GLTF Assets in deklarative und wiederverwendbare `React-Three-Fiber` JSX-Komponenten um (https://github.com/pmndrs/gltfjsx).

Zitat aus der `React-Three-Fiber` Doku/Pitfalls https://docs.pmnd.rs/react-three-fiber/advanced/pitfalls: „Regarding GLTF's try to use *GLTFJSX* as much as you can, this will create immutable JSX graphs which allow you to even re-use full models."

Das Problem ist, dass das 3D Modell von dem Gebäude, sowie die darin enthaltenden Räume und deren Children (Equipment und Bestuhlungsarten) veränderlich sein müssen. Somit muss das gesamte 3D Modell inklusive der einzelnen Komponenten, ihren eigenen State besitzen, um auf User-Inputs reagieren können und am Rendering-Loop teilzunehmen. Informationen wie die Farbe, Transparenz und Sichtbarkeit einer interaktiven Komponente des 3D Modelles müssen somit innerhalb eines States gehalten werden, sodass bei der Änderung dieses States, die Komponente / das Modell neugerendert wird. Ein einfaches Beispiel dieser Kombination aus `React-Three-Fiber` JSX-Komponenten, welche auf State Veränderungen reagieren wird im readme.md der `React-Three-Fiber` Doku auf der Seite https://github.com/pmndrs/react-three-fiber als Beispiel demonstriert.

Der erste Ansatz folgte dem demonstrierten Beispiel und nutzte das `GLTFJSX` Kommandozeilen-Tool und ergänzte die aus dem glTF Modell umgewandelten `React-Three-Fiber` JSX Komponenten mit den notwendigen `States` und `EventHandler`. Der generierte Code nur für das `roof` Mesh und einen der 6 Räume (`room_1`) mit allen seinen `children` (Equipment und Bestuhlungsarten) wird im Code Ausschnitt 3 gezeigt. Eine mögliche Ergänzung der generierten JSX-Komponenten mit den notwendigen `States` und `EventHandler` nur für das `roof` Mesh, einer der 6 Räume (`room_1`), einer der drei Stuhlformation des Raumes und ohne das raumbezogene Equipment wird im Code Ausschnitt 4 gezeigt.

```jsx
/*
Auto-generated by: https://github.com/pmndrs/gltfjsx
*/

import React, { useRef } from 'react'
import { useGLTF } from '@react-three/drei'

export default function Model({ ...props }) {
  const group = useRef()
  const { nodes, materials } = useGLTF('/house-model.glb')
  return (
    <group ref={group} {...props} dispose={null}>
      <mesh
        geometry={nodes.roof.geometry}
        material={nodes.roof.material}
        position={[0.52, -0.51, 0.17]}
        scale={0.01}
      />
      <mesh
        geometry={nodes.room_2.geometry}
        material={nodes.room_2.material}
        position={[0.08, 0.08, 0.04]}
        scale={0.01}>
        <mesh
          geometry={nodes.chair_fiormation_seminar.geometry}
          material={nodes.chair_fiormation_seminar.material}
          position={[-702.13, 117.04, -328.61]}
          rotation={[-Math.PI / 2, 0, Math.PI / 2]}
          scale={[42.82, 42.82, 42.82]}
        />
        <mesh
          geometry={nodes.chair_formation_concert.geometry}
          material={nodes.chair_formation_concert.material}
          position={[-823.73, 90.8, -341.14]}
          rotation={[-Math.PI / 2, 0, Math.PI / 2]}
          scale={[30.17, 30.17, 30.17]}
        />
        <mesh
          geometry={nodes.equipment_podium.geometry}
          material={nodes.equipment_podium.material}
          position={[-435.88, 89.68, -163.01]}
          rotation={[-Math.PI / 2, 0.06, Math.PI / 2]}
          scale={[0.64, 0.64, 0.64]}
        />
        <mesh
          geometry={nodes.equipment_beamer.geometry}
          material={nodes.equipment_beamer.material}
          position={[-835.16, 206.9, -168.23]}
          rotation={[-Math.PI / 2, 0, Math.PI / 2]}
          scale={1.1}
        />
        <mesh
          geometry={nodes.equipment_stage001.geometry}
          material={nodes.equipment_stage001.material}
          position={[-723.4, 90.45, -121.74]}
          rotation={[0, -1.57, 0]}
          scale={[3.5, 3.5, 3.5]}
        />
        <mesh
          geometry={nodes.chair_formation_bankett.geometry}
          material={nodes.chair_formation_bankett.material}
          position={[-805.8, 117.04, -325.34]}
          rotation={[-Math.PI / 2, 0, Math.PI / 2]}
          scale={[42.82, 42.82, 42.82]}
        />
      </mesh>
    </group>
  )
}

useGLTF.preload('/house-model.glb')
```

Code Ausschnitt 3: Ausschnitt vom autogenerierten Code vom glTF Gebäude Modell

```jsx
import React, { useRef } from 'react';
import { useGLTF } from '@react-three/drei';

export default function Model({ ...props }) {
    const group = useRef();
    const { nodes, materials } = useGLTF('/house-model.glb');
    return (
        <group ref={group} {...props} dispose={null}>
            <mesh
                geometry={nodes.roof.geometry}
                position={[0.52, -0.51, 0.17]}
                scale={0.01}
                key={nodes.windows.name}
                name={nodes.windows.name}
                onPointerOver={(event) => {...}}
                onPointerOut={(event) => {...}}
                onPointerDown={(event) => {...}}
                onPointerMissed={(event) => {...}}
            >
                <meshStandardMaterial
                attach='material'
                color={nodes.windows.color}
                transparent
                visible={nodes.windows.isVisible}
                opacity={nodes.windows.meshOpacity}
                metalness={0.5}
                />
            <mesh/>
            <mesh
                geometry={nodes.room_2.geometry}
                material={nodes.room_2.material}
                position={[0.08, 0.08, 0.04]}
                scale={0.01}
                key={nodes.room_2.name}
                name={nodes.room_2.name}
                onPointerOver={(event) => {...}}
                onPointerOut={(event) => {...}}
                onPointerDown={(event) => {...}}
                onPointerMissed={(event) => {...}}
            >
                <meshStandardMaterial
                attach='material'
                color={nodes.windows.color}
                transparent
                visible={nodes.windows.isVisible}
                opacity={nodes.windows.meshOpacity}
                metalness={0.5}
                />
            <mesh/>
            <mesh
                geometry={nodes.chair_formation_seminar.geometry}
                material={nodes.chair_formation_seminar.material}
                position={[-702.13, 117.04, -328.61]}
                rotation={[-Math.PI / 2, 0, Math.PI / 2]}
                scale={[42.82, 42.82, 42.82]}
                key={nodes.chair_formation_seminar.name}
                name={nodes.chair_formation_seminar.name}
            >
                <meshStandardMaterial
                attach='material'
                color={nodes.chair_formation_seminar.color}
                transparent
                visible={nodes.chair_formation_seminar.isVisible}
                opacity={nodes.chair_formation_seminar.meshOpacity}
                metalness={0.5}
                />
            <mesh/>
        </group>
    );
}

useGLTF.preload('/house-model.glb');
```

Code Ausschnitt 4: Ausschnitt vom autogenerierten Code vom glTF Gebäude Modell mit eigenen, ergänzten States und EventHandler

Es wird deutlich, dass auf Grund der hohen Menge an redundanten Code, welcher automatisch generiert wird, dieser Ansatz für alle 32 Meshes im Modell schlecht skalierbar und wartbar wäre. Zusätzlich wäre es schwer, die Übersicht über die Meshes und deren einzelne States zu bewahren und die einzelnen States der 32 Meshes zu verwalten, sobald diese gerendert wurden. Um dieses Problem zu lösen, wurden die Daten vom glTF Modell in eine eigene JSON Datenstruktur runtergebrochen und so vereinfacht, dass diese nur die relevanten Daten und Informationen enthalten. Abbildung 5 zeigt die originalen Daten, wie sie vom glTF Modell geladen werden. Abbildung 6 zeigt dagegen die vereinfachte, eigene JSON Datenstruktur. Es ist zu sehen, dass alle Informationen bis auf die `nodes` verworfen wurden, da sich alle notwendigen Informationen im `nodes` Objekt befinden.

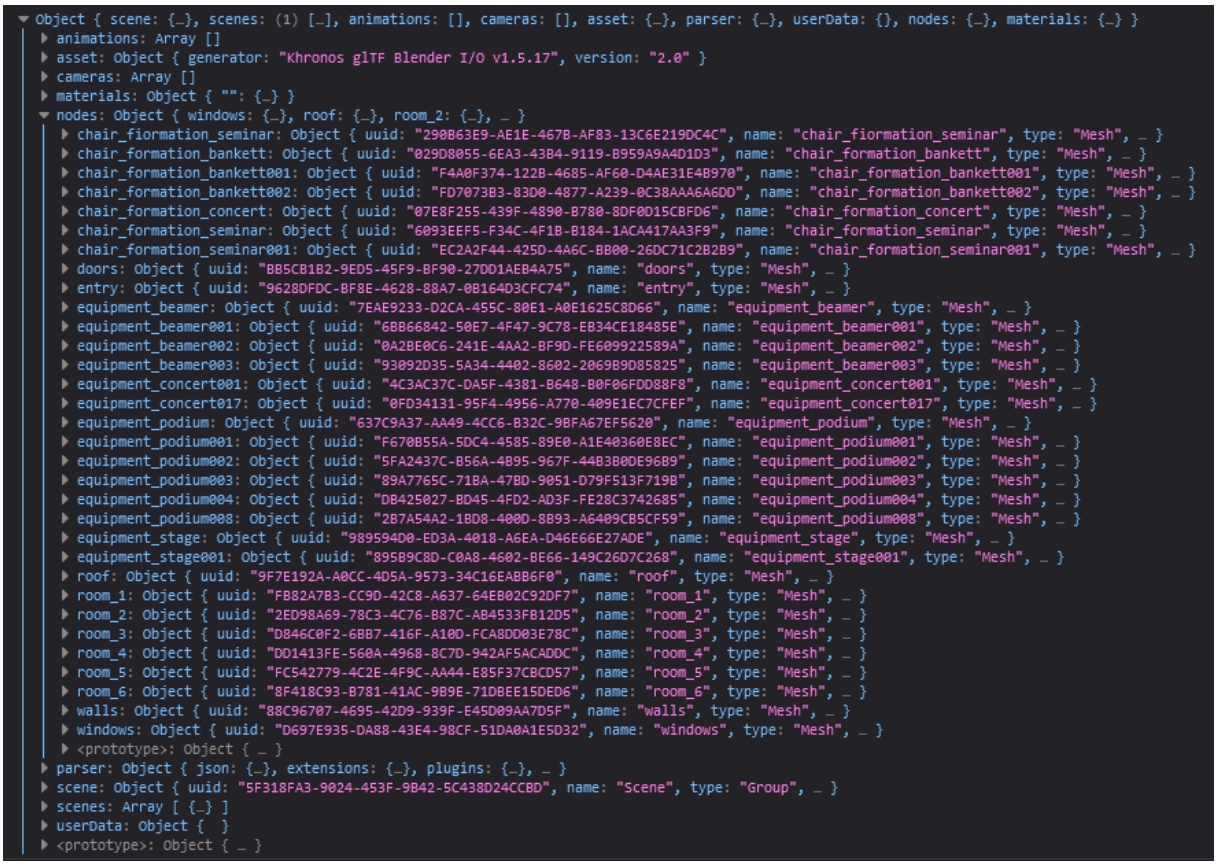


Abbildung 5: Originaldatenmodell vom glTF Gebäude Modell mit aufgeklapptem „nodes“ Objekt

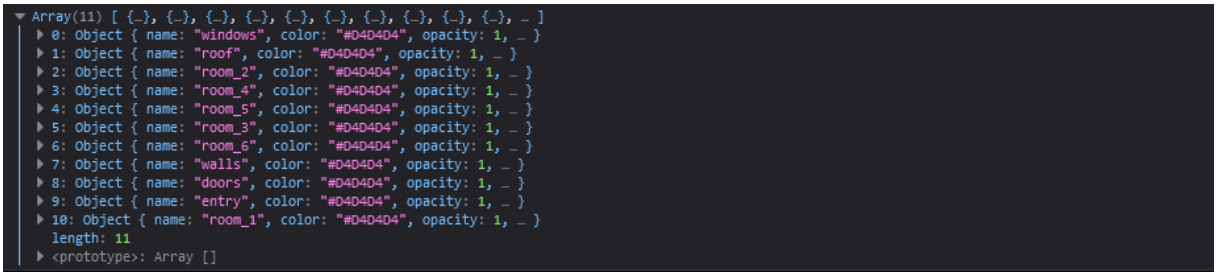


Abbildung 6: Vereinfachtes Datenmodell vom glTF Gebäude Modell

6.2.1 Mesh Abhängigkeiten

Im originalen glTF Modell befinden sich alle Mesh Objekte in `nodes` auf einer Ebene. Die Informationen über deren Abhängigkeit befindet sich im Mesh selbst innerhalb der `parent` und `children` Property. Die `parent` Property gibt an, welches Objekt der `parent` vom jeweiligen Mesh Objekt ist. Die `children` Property ist ein `Array` aus mehreren Objekten, welche alle die `children` von dem jeweiligen Mesh sind. Bei beiden Properties wird das gesamte Mesh mit allen zugehörigen Informationen gespeichert (Abbildung 7).

Abbildung 7: Originaldatenmodell vom glTF Gebäude Modell mit aufgeklapptem „room_1“ Objekt und dessen „parent“ und „children“ Property

Diese Informationen über die Parent-Child Abhängigkeiten der Meshes können auch in der Hierarchie der Elemente in Blender beobachtet werden (Abbildung 8). Entsprechend ist ersichtlich, dass die Meshes `chair_formation_concert`, `equipment_podium` und `equipment_stage` Child Elemente von `room_1` sind. Ausserdem ist erkennbar, dass `room_1` als Parent Element die „Scene“ selber hat, da nicht dem `room_1` übergeordnet ist. Das Gleiche gilt auch für die anderen Raum Meshes sowie Gebäude Meshes (`doors`, `entry`, `roof`, `windows`, `walls`).

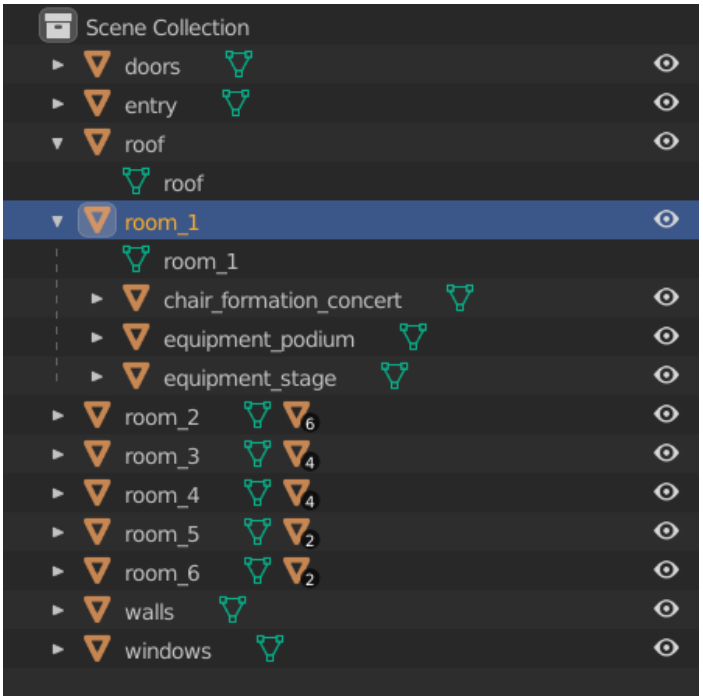


Abbildung 8: Elementen Hierarchie mit aufgeklapptem „room_1“ Objekt in Blender

Die eigene Datenstruktur lehnt an die Blender Element Hierarchie an und speichert alle Meshes, welche die „Scene“ als `parent` haben direkt ab (Abbildung 9) und speichert deren `children` innerhalb der jeweiligen Meshes in einem `children Array` genauso wie bei der originalen glTF Datenstruktur (Abbildung 7). Somit kommt jedes Mesh nur einmal in der eigenen, vereinfachten Datenstruktur vor, während die Meshes, welche die „Scene“ als `parent` besitzen, ihre Abhängigkeiten zu deren `children` nach wie vor kennen und in sich gespeichert haben.

Abbildung 9: Vereinfachtes Datenmodell vom glTF Gebäude Modell mit aufgeklapptem „room_1“ Objekt und dessen aufgeklappter „children“ Property

6.2.2 Mesh Informationen

Die ursprüngliche Menge an Mesh Informationen im originalen glTF Modell, wie sie in Abbildung 7 oder im Code Ausschnitt 5 gezeigt wird, wurde für das Rendering und für die Interaktion auf die im Code Ausschnitt 6 gezeigten Informationen reduziert. Diese Informationen werden in einer Liste (`meshList: MeshObjectType[]`) gespeichert, welche global als ein State über den `useMeshStore.ts` Store zugänglich ist, sodass auch ausserhalb des Modelles, Interaktionen mit dem Mesh getriggert werden können. Alle Meshes, unabhängig ob Child Mesh oder nicht, haben dieses Interface.

```
export class Object3D {
    id: number;
    uuid: string;
    name: string;
    type: string;
    parent: Object3D | null;
    children: Object3D[];
    up: Vector3;
    readonly position: Vector3;
    readonly rotation: Euler;
    readonly quaternion: Quaternion;
    readonly scale: Vector3;
    readonly modelViewMatrix: Matrix4;
    readonly normalMatrix: Matrix3;
    matrix: Matrix4;
    matrixWorld: Matrix4;
    matrixAutoUpdate: boolean;
    matrixWorldNeedsUpdate: boolean;
    layers: Layers;
    visible: boolean;
    castShadow: boolean;
    receiveShadow: boolean;
    frustumCulled: boolean;
    renderOrder: number;
    animations: AnimationClip[];
    userData: { [key: string]: any };
    customDepthMaterial: Material;
    customDistanceMaterial: Material;
    readonly isObject3D: true;

    // Methods, Functions and EventHandlers ...
}
```

Code Ausschnitt 5: Object3D Klasse der einzelnen Mesh Objekte in dem originalen Datenmodell des GLTF-Files

```
export type MeshObjectType = {
    name: string;
    geometry: THREE.BufferGeometry;
    position?: THREE.Vector3;
    rotation?: THREE.Euler;
    scale?: THREE.Vector3;
    material: THREE.Material;
    color: string;
    opacity: number;
    isVisible: boolean;
    userData?: Record<string, string>;
    children?: MeshObjectType[];
};
```

Code Ausschnitt 6: MeshObjectType Interface der reduzierten, für die Interaktion und das Rendering relevanten Informationen des eigenen, reduzierten Datenmodelles des GLTF-Files

Optionale Properties:

- **`position, rotation, scale:`** sind optional, da diese nur für die Transformation der `children` benötigt werden. Somit müssen diese Eigenschaften zusätzlich für die Children Meshes aus der originalen Datenstruktur ausgelesen, gespeichert und beim Rendering gesetzt werden.
- **`children`**: ist optional, da Children Meshes keine weiteren Children Meshes besitzen können. In der Applikation besitzen nur die Räume (`room_1` – `room_6`) Children Meshes (`chair_formation` und `equipment`).
- **`userData:`** ist optional, da nur interaktive Meshes, welche in der 3D-Computergrafik-Software als solche markiert werden müssen, die Property `userData` mit einem `customName` besitzen. `userData` wird im Kapitel 6.3 genauer erklärt. Die Properties `color`, `opacity` und `isVisible` der Parent und Children Meshes werden durch User Inputs verändert und entsprechend im Rendering Loop wieder erneut mit dem Modell gerendert.

6.2.3 Übertragung der Mesh Informationen

Zur Übertragung der reduzierten Informationen in die eigene Datenstruktur wurde die `convertGLTFTOMeshList()` Methode geschrieben, welche direkt bei der Initialisierung des Modelles einmalig aufgerufen wird. Vereinfacht beschrieben durchläuft diese Methode jeden Node und erstellt für jeden Node eine Liste mit dessen Children Nodes falls vorhanden. Danach wird überprüft, ob der aktuelle Node selber ein Child Node ist, in dem der `mesh.parent` dieses Nodes abgefragt wird. Der `mesh.parent` von nicht Children Nodes ist die „`Scene`“. Der `mesh.parent` der Child Nodes ist entsprechend einer der Räume. Dieser Check wird gemacht, um die Nodes nicht mehrmals in die Datenstruktur einzufügen, wie in der originalen glTF Datenstruktur. Ist der aktuelle Node kein Child Node, so können alle seine Informationen und die Informationen seiner Children Nodes (falls vorhanden) in die eigene Datenstruktur übertragen werden (};
Code Ausschnitt 7).

```
const convertGLTFToMeshList = (nodes: { [name: string]: THREE.Mesh }) => {
    const initialMeshList: MeshObjectType[] = [];
    delete nodes.Scene;
    delete nodes.Camera;
    delete nodes.Light;

    Object.values(nodes).forEach((mesh) => {
        const children: MeshObjectType[] = [];
        if (mesh.children.length !== 0) {
            Object.values(mesh.children).forEach((child: any) => {
                children.push({
                    // transfering all relevant children nodes informations
                });
            });
        }

        if (mesh.parent?.name === 'Scene') {
            initialMeshList.push({
                // transfering all relevant parent node informations
                children: children,
            });
        }
    });

    return initialMeshList;
};
```

Code Ausschnitt 7: Vereinfachte convertGLTFToMeshList Funktion zur Konvertierung der originalen GLTF Datenstruktur in die vereinfachte Datenstruktur (meshList)

Anschliessend wird ähnlich wie bei der Übertragung der Informationen die erstellte eigene Mesh Liste durchlaufen und für jedes Mesh überprüft, ob es `children` besitzt. Besitzt es welche, werden alle `children` durchlaufen und gerendert. Danach wird das Parent Mesh selbst gerendert. Alle Meshes werden entsprechend der `React-Three-Fiber` Empfehlung als `React-Three-Fiber` JSX-Komponente gerendert.

6.3 Naming Conventions und Verbindungspunkte zwischen dem glTF 3D Modell und der React Applikation

Aus den Kapiteln 6.1 und 6.2 wird deutlich, dass es zwei Datenmodelle gibt. Ein statisches mit allen für den User relevanten Informationen zu den Räumen und ein dynamischen, in einem State hinterlegtes Datenmodell, mit allen relevanten Informationen für das 3D Modell. Diese beiden Modelle benötigen einen Verbindungspunkt, sodass herausgefunden werden kann, um welchen Raum es sich handelt, wenn der Nutzer einen Raum im 3D Modell anklickt. Genauso muss im 3D Modell vom Gebäude der richtige Raum angezeigt werden, wenn der Nutzer eim Accordion Element auswählt. Um diesen Verbindungspunkt zu schaffen, wurde eine Naming Convention eingeführt. Wie im Kapitel 6.1 bereits erwähnt, betrifft diese Naming Convention die vier Properties `model.meshName`, `info.chairFormations`, `info.equipment` und `info.fittingSideRooms`. Diese vier Properties werden als Referenz auf deren 3D Meshes in dem glTF File des 3D Gebäudes verwendet.

6.3.1 Der Name des 3D Objektes als Referenz

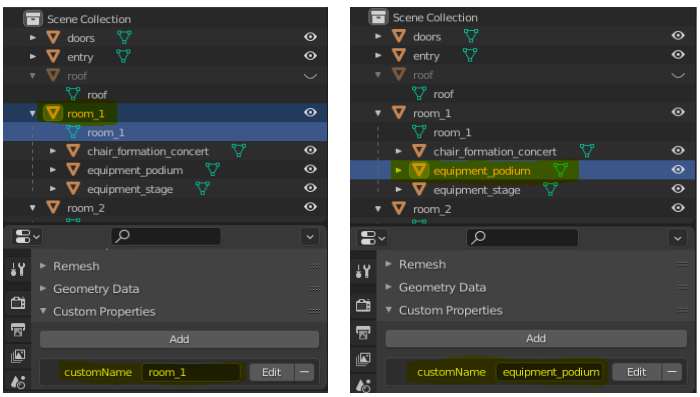
Der erste Ansatz war, für eine Übereinstimmung zwischen den oben erwähnten vier Properties und den in Blender definierten Mesh Namen zu sorgen, sodass die Namen im statischen Datenmodell (`roomData`) mit den Namen der 3D Meshes im 3D Modell in dem dynamischen Datenmodell (`meshList`) übereinstimmen.

Das würde bedeuten:

- Ist der Konzertsaal das 3D Objekt `room_1`, so heisst dessen `model.meshName` auch `room_1`.
- Hat der Konzertsaal als Nebenraum den Clubraum, welcher im glTF Modell vom 3D Objekt `room_3` dargestellt wird, so heisst die `model.meshName` Property vom Clubraum `room_3` und in der `info.fittingSideRooms` Property vom Konzertsaal wird dann entsprechend auf `room_3` referenziert.
- Besitzt der Konzertsaal eine Bühne, welche im 3D Modell als `equipment_stage` benannt worden ist, so ist die Bühne vom Konzertsaal in seiner `info.equipment` Property auch als `equipment_stage` benannt und kann als Referenz zu den entsprechenden 3D Objekten im 3D Model verwendet werden.
- Dieselbe Logik gilt auch für die Stuhlformationen. Die Konzert Bestuhlung muss sowohl im statischen Datenmodell bei `info.chairFormations` als auch im dynamischen, glTF Datenmodell bei dem 3D Mesh der Konzert Bestuhlung identisch benannt sein (`chair_formation_concert`).

6.3.2 Die Custom Property „customName“ des 3D Objektes als Referenz

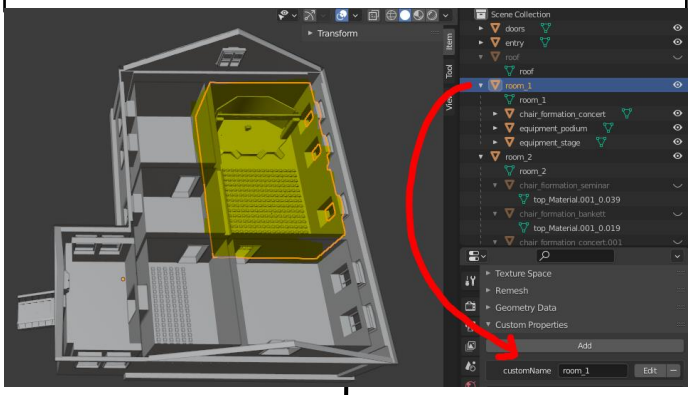
Obwohl dieser Ansatz funktionierte, war er nicht optimal für zukünftige Entwicklungen am 3D Modell. Alle Namen der relevanten 3D Meshes mussten der Naming Convention folgen, damit die React Applikation funktioniert. Jedoch haben 3D Entwickler meistens eigene Naming Conventions und benötigen diese, um sich bei der 3D Entwicklung im 3D Modell besser zu orientieren. Die vorgegebene Naming Convention würde somit die Namensgebung der Objekte stark einschränken. Um das zu lösen, wurde der zweite Ansatz entwickelt, welcher die Custom Properties der einzelnen 3D Objekte nutzt. Für jedes 3D Objekt lassen sich Custom Properties definieren. Dabei muss der Name der Property und der Wert festgelegt werden. Die Idee ist, die Information, welche vorher im Namen des 3D Objektes gespeichert wurde, in einer Custom Property namens `customName` zu speichern, wie in der Abbildung unten zu sehen ist.



Diese Custom Property wird dann im glTF Datenmodell als `userData: {customName: „room_1“}` übertragen und kann genauso wie die `name` Property ausgelesen und ins eigene Datenmodell übernommen werden.
Der gesamte Prozess wird in der Abbildung unten dargestellt.

IN BLENDER

1. Definition des `customName` als Customer Property für alle interaktiven 3D Meshes



IM GELADENEN GLTF DATENMODELL

2. Objekte im glTF Model tragen die in Blender definierte Custom Property `customName` als `userData:{customName: “room_1“}`

```
room_1: Object { uuid: "DBA8DE0C-5A17-42F3-9DB3-DE0990C41122", name: "room_1", type: "Mesh", … }
  userData: Object { customName: "room_1", name: "room_1" }
    customName: "room_1"
    name: "room_1"
    <prototype>: Object { … }
```

IM EIGENEN DATENMODELL

3. Objekte im eigenen Datenmodell übernehmen die `customName` Property als deren `name` Property

```
10: Object { name: "room_1", color: "#D4D4D4", opacity: 1, … }
```

IN DER REACT APPLIKATION

4. `INTERACTABLE_MESH_NAMES` müssen identische Namen besitzen wie die `customName` Properties deren korrespondierenden 3D Meshes.

```
export enum INTERACTABLE_MESH_NAMES {
    concertRoom = 'room_1',
    luzernerRoom = 'room_2',
    clubroom = 'room_3',
    businessMediaRoom = 'room_4',
    auditorium = 'room_5',
    entryFoyer = 'room_6',
    roof = 'roof',
}
```

5. `INTERACTABLE_MESH_NAMES` werden den 3D Objekten zugewiesen, welche das entsprechende 3D Mesh im glTF Model referenzieren sollen.

```
export const roomList: RoomFetchedDataType[] = [
    {
        model: {
            meshName: INTERACTABLE_MESH_NAMES.concertRoom,
            camPos: new THREE.Vector3(...),
            camTarget: new THREE.Vector3(...),
        },
        info: {
            title: 'Konzertsaal',
            fittingSideRooms: [
                INTERACTABLE_MESH_NAMES.clubroom,
                INTERACTABLE_MESH_NAMES.businessMediaRoom
            ],
            ...
        },
    },
    ...
];
```

6.3.3 Idee hinter den Verbindungspunkten

Dadurch das die Namen der Räume, des Equipments und der Bestuhlungen identisch sind mit den in Blender definierten Custom Properties (`customName`), entstehen Verbindungspunkte zwischen den Rauminformationen und den Raumdaten im statischen Datenmodell und deren 3D Objekte im dynamischen Datenmodell. Durch diese Verbindungspunkte kann beispielsweise sichergestellt werden, dass bei dem Öffnen des Accordion Elementes vom „Konzertsaal“, das 3D Objekt vom „Konzertsaal“ hervorgehoben wird. Gleichzeitig ist es möglich, im 3D Modell das 3D Objekt von „Konzertsaal“ auszuwählen, sodass sich das Accordion Element vom „Konzertsaal“ öffnet. Das funktioniert nur, weil sich beide durch den identischen Namen „`room_1`“ identifizieren lassen können und dadurch miteinander verbunden sind.

6.3.4 Custom Properties als Merkmal zur Unterscheidung zwischen interaktiven und nicht interaktiven 3D Objekten

Die Verwendung der Custom Properties hat noch einen weiteren Vorteil. Dadurch, dass nur die 3D Objekte, welche interaktiv sind, die `customName` Property besitzen, kann dieses Merkmal in der React Applikation genutzt werden zur Unterscheidung zwischen interaktiven und nicht-interaktiven 3D Objekten genutzt werden. Dadurch besitzen auch einige 3D Objekte keinen `customName`, wie die fest definierte Konzert Bestuhlung („`chair_formation_concert`“) im Konzertsaal. Diese soll im Gegensatz zu den anderen Bestuhlungen mit `customName` in ihrer Sichtbarkeit nicht togglebar sein. Auch besitzen Parent 3D Objekte vom Gebäude, wie die Fenster („`windows`“), die Wand („`walls`“) und der Eingang („`entry`“), keinen `customName`, da diese nicht klickbar, hoverbar oder in ihrer Sichtbarkeit togglebar sein sollen.

6.3.5 Prefixes für Equipment und Stuhlformationen

Zusätzlich zu der oben beschriebenen Naming Convention, wurden noch Prefixes für die auswählbaren und konfigurierbaren Equipment- („`equipment`“) und Bestuhlungsmöglichkeiten („`chair_formation`“) ergänzt. Diese werden in den `CHAIR_FORMATION` und `EQUIPMENT Enums` im Code festgehalten, wie unten im Code Ausschnitt 8 zu sehen ist. Diese müssen sowohl im dynamischen Datenmodell für die Properties `info.chairFormations` und `info.equipment` sowie in der Custom Property `customName` der jeweiligen 3D Objekte beachtet werden.

```
export enum CHAIR_FORMATION {
    concert = 'chair_formation_concert',
    seminar = 'chair_formation_seminar',
    bankett = 'chair_formation_bankett',
}

export enum EQUIPMENT {
    stage = 'equipment_stage',
    podium = 'equipment_podium',
    beamer = 'equipment_beamer',
}
```

Code Ausschnitt 8: CHAIR_FORMATION und EQUIPMENT Enums

Die Prefixes ermöglichen eine deutlich einfachere und effizientere Differenzierung zwischen dem Raumequipment und den Stuhlformationen in der React Applikation. Diese Differenzierung wird gebraucht bei dem Toggeln der Sichtbarkeit der einzelnen Equipment- und Stuhlformationsmöglichkeiten. Mehr dazu im nächsten Kapitel.

6.4 Ein- und ausblenden von Meshes innerhalb des Modelles

Das Ein- und Ausblenden von 3D Objekten innerhalb des Modelles baut auf die vorher beschriebenen Verbindungspunkte und auf dem dynamischen, im State gespeicherten Datenmodell vom 3D Modell auf. Dadurch, dass das 3D Modell jedes Mal neu gerendert wird, wenn der State des dynamischen Datenmodelles verändert wird, genügt es, den `isVisible` Wert eines Parent oder Child 3D Objektes zu verändern, um das 3D Objekt ein- und auszublenden. Hierfür wurden drei Methoden geschrieben:

- **`setMeshParentVisibility(meshName: INTERACTABLE_MESH_NAMES, visibility: boolean):`**
  Verändert die Sichtbarkeit von einem 3D Objekt auf der ersten Ebene des dynamischen `meshList` States (`room_1 – room_6, walls, windows, roof, entry`), welches als `name` Property den übergebenen `meshName` hat, indem dessen `isVisible` Property mit dem übergebenen `visible` Wert überschrieben wird. Diese Methode wird verwendet, um das Dach des 3D Modelles auszublenden, sodass der Nutzer die einzelnen Räume besser sehen kann.

- **`toggleMeshChildVisibility(toggledRoomName: INTERACTABLE_MESH_NAMES, toggledMeshName: CHAIR_FORMATION | EQUIPMENT, category?: ROOM_ADDITIONS_CATEGORY):`**
  Toggelt die Sichtbarkeit von dem `CHAIR_FORMATION` oder `EQUIPMENT` 3D Child Objekt, welches als `name` Property den übergebenen `toggledMeshNameden` hat und welches in den `children Array` von dem `INTERACTABLE_MESH_NAMES` Parent Objekt liegt, das als `name` Property den übergebenen `toggledRoomName` hat. Zusätzlich wird bei allen anderen Child Objekten im `children Array`, welche der gleichen `category` angehören, die `isVisible` Property auf `false` gesetzt. Um herauszufinden, welche Child Objekte dieser gleichen `category` angehören, werden die Prefixe „`equipment`“ und „`chair_formation`“ mit der übergebenen `category` abgeglichen. Diese Methode wird in den `MeshVisibilityButton.tsx` verwendet, um jeweils eine Konfigurationsmöglichkeit (Stuhlformation oder Equipment) auszuwählen, während die zuvor ausgewählten Konfigurationen unsichtbar gemacht werden.

- **`resetMeshVisibility():`**
  Durchläuft alle Parent Objekte in der dynamischen `meshList` und setzt deren `isVisibility` Property auf `true`, während gleichzeitig die `isVisible` Property deren `children`, welche einen `customName` besitzen auf `false` gesetzt wird. Diese Methode wird nach der Absendung der Konfiguration aufgerufen, um das Dach wieder einzublenden und alle toggelbaren Konfigurationen anfangs unsichtbar zu machen.

6.5 Interaktionen mit dem Modell

6.5.1 Hover und Klickinteraktionen

Interaktive 3D Objekte, welche auf der ersten Ebene des dynamischen `meshList` States sind und einen `customName` besitzen („room_1“, „room_2“, „room_3“, „room_4“, „room_5“ „room_6“, „roof“), erhalten beim Rendering die `EventHandler` `onPointerOver`, `onPointerOut` und `onPointerDown`. Diese drei `EventHandler` ermöglichen die Hover- und Klickinteraktionen mit dem 3D Modell.

- **`onPointerOver`:** Setzt den Namen aktuell gehoverete 3D Mesh (`event.object.name`) in den `hoveredMesh` State, welcher sowohl für den `Cursor.tsx` als auch für die `getMeshColor` und `getMeshOpacity` Methoden innerhalb der `Model.tsx` Klasse gebraucht wird. `Cursor.tsx` nutzt den Namen des gehovereten Meshes, um den Raumnamen (`info.title`) aus den statischen Rauminformationen herauszufinden und diesen beim Hoveren über das jeweilige Mesh als Cursor anzuzeigen. Die beiden Methoden `getMeshColor` und `getMeshOpacity` nutzen den Namen des gehovereten Meshes um die `opacity` und `color` Werte vom `meshStandartMaterial` des jeweiligen Meshes bei einer Änderung des `hoveredMesh` States neu zu berechnen und das Mesh mit dem aktualisierten Material erneut zu rendern.

- **`onPointerOut`**: Setzt den `hoveredMesh` State auf `null`.
- **`onPointerDown`**: Wird ein Mesh angeklickt, so werden mehrere Checks durchlaufen.
  1. Ist das Mesh das Dach Mesh, so wird eine Obersicht von allen Räumen gezeigt, bei der das Dach ausgeblendet wird.
  2. Befindet sich der Nutzer im ersten Schritt (Eingabe der Filterkriterien) oder im letzten Schritt (Überprüfung der Eingaben & Buchung) des Wizards und klickt ein Mesh an, welches nicht das Dach Mesh ist, so wird dieses optisch hervorgehoben und mittels einer Kamerafahrt in eine raumspezifische Nahansicht positioniert. Wichtig ist hierbei, dass das Mesh nicht als für die Buchung reservierter Raum (`activeMainRoom` oder `activeSideRoom`) vermerkt wird. Das liegt daran, dass der Nutzer im ersten und im letzten Schritt keine Räume reservieren soll, aber dennoch die Möglichkeit besitzen soll, durch das Anklicken der Räume, diese optisch hervorzuheben und aus der Nahansicht genauer betrachten zu können.
  3. Befindet sich der Nutzer nicht im ersten oder letzten Schritt des Wizards, sondern bereits bei der Haupt- oder Nebenraumauswahl und klickt einen Hauptraum an, der die zuvor definierten Filterkriterien (Art des Events, Anzahl Teilnehmer, Start- und Enddatum) alle erfüllt, so wird der Raum beim Anklicken nicht nur optisch hervorgehoben und mittels einer Kamerafahrt in eine raumspezifische Nahansicht positioniert, sondern auch als reservierter Hauptraum (`activeMainRoom`) vermerkt. Klickt der Nutzer einen Nebenraum an, so sind die zuvor definierten Filterkriterien, ob der Raum einer der `info.fittingSideRooms` des reservierten Hauptraumes (`activeMainRoom`) ist und ob der Nebenraum nicht bereits zu dem ausgewählten Start- und Enddatum gebucht worden ist. Erfüllt der angeklickte Nebenraum alle diese Kriterien, so wird dieser genauso wie beim Hauptraum optisch hervorgehoben und mittels einer Kamerafahrt in eine raumspezifische Nahansicht positioniert, sowie als reservierter Nebenraum (`activeSideRoom`) vermerkt.

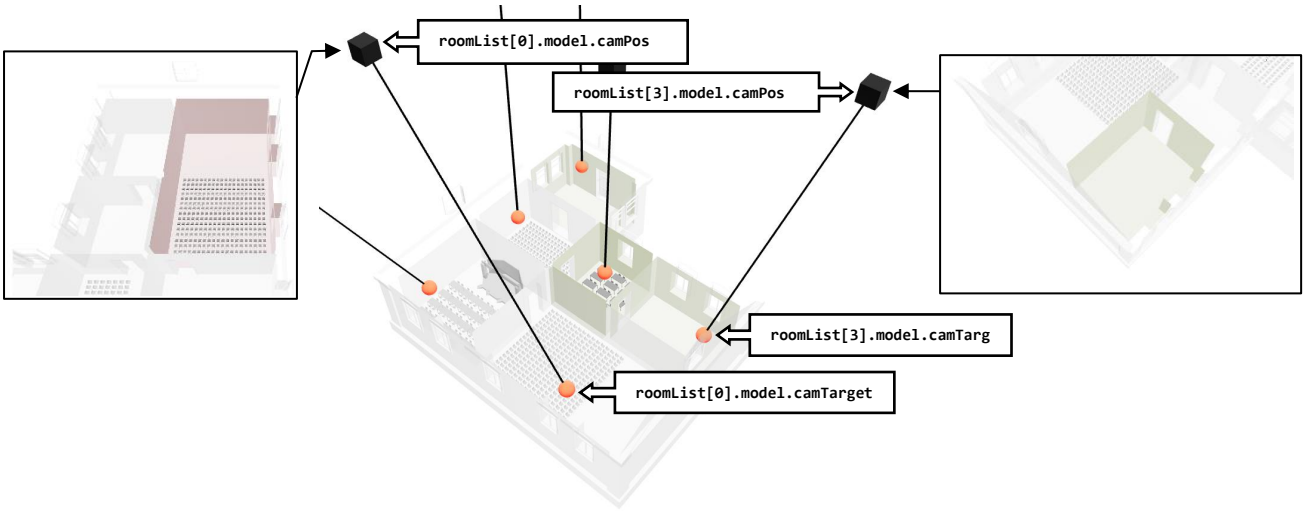
6.5.2 Rotation des Modelles

Die Rotation des Modelles wird mit Hilfe der `OrbitControls` von der `React-Three-Drei` Library ermöglicht. Diese wurden entsprechend angepasst, um die Kameraführung durch den Nutzer möglichst einfach zu gestalten, sodass unvorteilhafte Perspektiven vermieden werden. Dazu wurde das Kameraschwenken mit der rechten Maustaste sowie das Kamerazoomen mit dem Mausrad deaktiviert (`enablePanning={false}`, `enableZoom={false}`). Dadurch kann der Nutzer das 3D Objekt nichtmehr aus dem Sichtfeld verlieren, da nur noch die Kamera um das 3D Objekt rotiert werden kann. Auch die vertikale Rotation wurde beschränkt (`maxPolarAngle={Math.PI / 2}`), sodass die Kamera nicht unterhalb des 3D Objektes rotiert werden kann, da es dem Nutzer keinen Nutzen bringt, das Gebäude oder die Räume von unten zu sehen. Zusätzlich transformiert / interpoliert sich die Kamera „smooth“ von der aktuellen Kameraposition, welche durch die Rotation des Nutzers entstanden ist, zu der ursprünglichen Position Kameraposition (`model.camPos`) und der Kamerablickrichtung (`model.camTarget`) des angeklickten Raumes bestimmt wird, sobald der Nutzer die Linke Maustaste loslässt / den Finger vom Touchscreen entfernt. Hierzu wird die `damp` Funktion der `THREE.MathUtils` verwendet.

6.6 Kameraführung nach Interaktionen

6.6.1 Idee

Wie bereits im Kapitel 6.1 erwähnt, besitzt jeder Raum eine raumspezifische Kamera Position (`model.camPos`) und einen raumspezifischen Kamerablickpunkt (`model.camTarget`). Das dient dazu, dass jeder Raum seine eigene, individuelle Nahansicht besitzt, welche angezeigt wird, sobald der Raum in der Raumliste ausgewählt oder im 3D Modell angeklickt wird. Diese Ansicht ermöglicht dem Nutzer, den Raum und dessen Raumdetails im 3D Modell besser zu erkennen. In der unteren Abbildung sind die `camPos` und `camTarget` anhand von `Cube` und `Sphere` Meshes eingeblendet. Der `Cube` ist die Kamera und blickt zu dem `Sphere` Mesh, welches das `Target` ist.



Im `useCameraStore.ts` befinden sich zwei globale States, welche die aktuell ausgewählte `camPos` und den aktuell ausgewählten `camTarget` speichern. Diese werden jedes Mal, wenn ein neuer Raum ausgewählt wird, mit den raumspezifischen, neuen `camPos` und `camTarget` Werten aktualisiert. Diese aktualisierten Werte können in der `CameraControls.tsx` Klasse, welche auch die `PerspectiveCamera` sowie die `OrbitControls` initialisiert, genutzt werden, um die `PerspectiveCamera` entsprechend zu positionieren und auszurichten.

6.6.2 Problem

Der erste Lösungsansatz war, die Position der Kamera auf die aktualisierte, vom Store ausgelesenen `camPos` Position zu setzen und die Kamera mittels der `lookAt()` Funktion auf die aktualisierte, vom Store ausgelesene `camTarget` Position eines bestimmten Raumes blicken zu lassen. Dieser Ansatz mit der `lookAt()` Funktion wird häufig in anderen 3D Echtzeit Entwicklungsplattformen wie beispielsweise Unity verwendet. Das Problem hierbei sind die `OrbitControls` von `ThreeJS`. Zwar wird bei diesem Ansatz der Fokuspunkt der Kamera verändert, aber nicht der Fokuspunkt der `OrbitControls`. Somit rotiert die Kamera weiterhin um den Nullpunkt im Koordinatensystem (0, 0, 0), auch wenn bereits ein Raum ausgewählt worden ist.
Andere vorgeschlagene Ansätze, wie den Rotation- oder den Quaternion-Wert der Kamera manuell zu setzen, führten zum gleichen Problem. Parallel wurde auch versucht den Rotation- oder den Quaternion-Wert der `OrbitControls` anzupassen oder mit den Methoden `setAzimuthalAngle()` oder `setPolarAngle()` zu setzen. Auch dieser Ansatz hatte kein Erfolg.

6.6.3 Lösung

Die Lösung ist die Verwendung der `target` Property der `OrbitControls`, auf welche mittels einer `useRef` Hook zugegriffen werden kann. Die `target` Property ist der Fokuspunkt, um welchen die `OrbitControls` und die `PerspectiveCamera` kreisen. Somit kann dieser vom Nullpunkt auf den `model.camTarget` Wert des gewünschten Raumes gesetzt werden, sodass die `OrbitControls / Camera` um den Raum kreisen. Interessanterweise ist der `target` Punkt der `OrbitControls` gleichzeitig auch der Fokuspunkt, auf welchen sich die `Camera` ausrichtet. Daher genügt es, auf die `position` der `Camera` mittels einer `useRef` Hook zu zugreifen und diese auf die für den Raum speziell definierte `camPos` zu setzen und als `target` Punkt der `OrbitControls` den `model.camTarget` des Raumes zu definieren.

6.6.4 Animation

Damit sich die neuen Kamera Werte `camPos` und `camTarget` smooth von den vorherigen zu den neuen Werten anpassen und eine Kamerafahrt entstehen kann, muss die Änderung in der `useFrame` Hook von `React-Three-Fiber` ausgeführt werden. Diese erlaubt das Rendern von 3D Komponenten basierend auf jedem Frame Update. Somit können die `PerspectiveCamera` sowie die `OrbitControls` ständig im Rendering Loop geupdated werden. Zusätzlich kann aus der `useFrame` Hook das Zeitdelta ausgelesen werden. Mit diesem und der `THREE.MathUtils.damp()` Funktion, wird die Animation der Kamera ermöglicht. Dazu benötigt die `THREE.MathUtils.damp()` Funktion den aktuellen Wert, den Ziel Wert, ein Geschwindigkeits-Lambda und das Zeit Delta in Sekunden als Parameter. Mit diesen Werten kann die `THREE.MathUtils.damp()` Funktion die Kamera Position und Blickrichtung smooth interpolieren.

6.7 Interaktionen mit dem Modell durch den Wizard

6.7.1 Zustand als State Management Solution

Damit der Nutzer mit dem Modell durch den Wizard interagieren kann und vice versa, müssen bestimmte Informationen, wie zum Beispiel der aktuell reservierte Haupt- und Nebenraum zwischen den Wizard Klassen und der Modell Klasse geteilt werden. Hierfür wird eine State Management Solution benötigt. Im ersten Ansatz wurde `Redux` verwendet. Doch da der React `context` nicht ohne Weiteres zwischen zwei Renderern verwendet werden kann, ist es nicht möglich, einen `Provider` der bereits im `ReactDOM` erstellt wurde, auch in der `React-Three-Fiber <Canvas>` Komponente zu konsumieren. Diese Information kann in der `React-Three-Fiber` Dokumentation unter https://docs.pmnd.rs/react-three-fiber/advanced/gotchas „Consuming context from a foreign provider“ nachgelesen werden:

“At the moment React context can not be readily used between two renderers, this is due to a problem within React. If ReactDOM opens up a provider, you will not be able to consume it within `<Canvas>`. If managing state (like Redux) is your problem, then Zustand is likely the best solution […].”

Da Lösungen wie `Redux` oder die `Context API` auf den `Provider` angewiesen sind, um deren `store` für andere geschachtelte Komponenten zugänglich zu machen und `Zustand` im Gegensatz dazu eine providerlose Lösung ist, welche auf vereinfachten Flux Prinzipien aufbaut und von `React-Three-Fiber` als die wahrscheinlich beste Lösung empfohlen wird, wird `Zustand` als State Management Lösung verwendet.

6.7.2 Logik hinter den selectedMeshes und filtredMeshes

Die untere Abbildung 10 zeigt das 3D Modell bei der Hauptraumauswahl. Der `ThreeJsDataDebugger` ist aktiviert und zeigt die aktuell ausgewählten `filteredMeshes` und `selectedMeshes`. Im Wizard als auch im Modell ist erkennbar, dass der Luzerner Saal (`room_2`) als aktiver Raum selektiert worden ist und somit in das `selectedMeshes Array` hinzugefügt wurde. Die beiden Räume Luzerner Saal und Konzertsaal sind nach der Filtrierung als auswählbare Räume übriggeblieben und werden daher ins `filteredMeshes Array` hinzugefügt wurden. Entsprechend sehen die Daten auch in dem `useCameraStore.ts` aus, welcher die `filteredMeshes` und `selectedMeshes` als globale zugängliche States abspeichert.

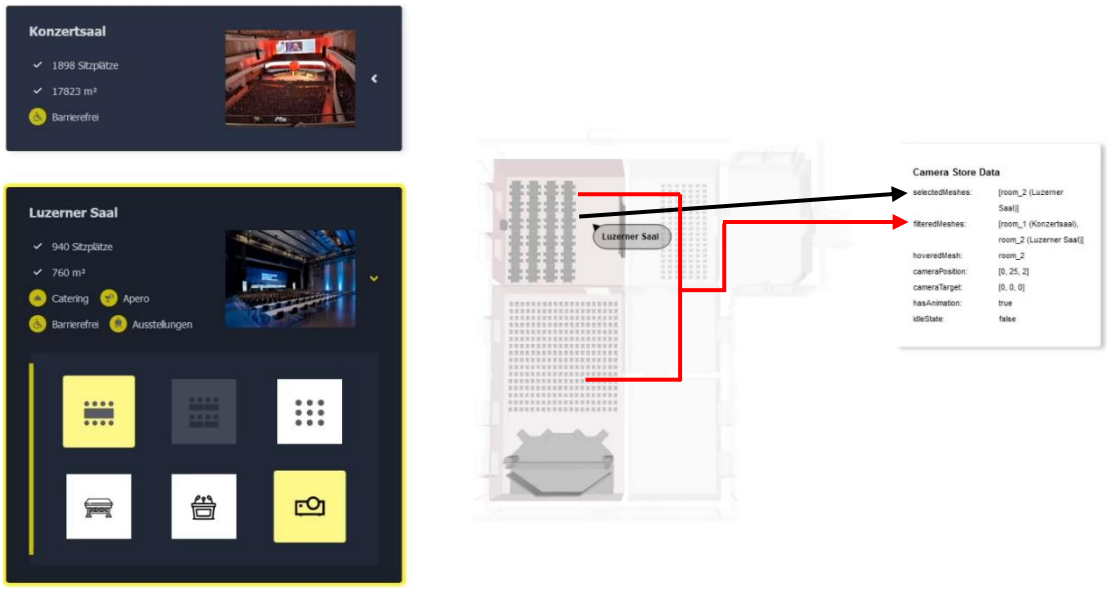


Abbildung 10: Logik von selectedMeshes und filteredMeshes Array anhand von einem Beispiel

`filteredMeshes`: wird verwendet, um zu erkennen, welche Räume die Filterkriterien erfüllen und im 3D Modell entsprechend markiert und in der Raumliste gerendert werden sollen. Nur Räume aus dem `filteredMeshes Array` werden als Accordion Element angezeigt und können im 3D Modell ausgewählt werden.

- Im ersten Schritt des Wizards (`RoomFilteringWizard.tsx`) werden die Filterkriterien für die Hauptraumauswahl gesammelt.
- Im zweiten Schritt (`RoomMainSelectionWizard.tsx`) wird mit Hilfe dieser, die gesamte Menge an Haupträumen filtriert. Die resultierenden Mesh Namen der Räume werden im `filteredMeshes` State gespeichert.
  Mit Hilfe des `filteredMeshes Arrays` werden aus dem statischen Datenmodell (`roomList`) nur die passenden Räume in der Raumliste gerendert und im 3D Modell auswählbar gemacht und durch die Veränderung deren `color` und `opacity` Werte markiert.
- Im dritten Schritt (`RoomSideSelectionWizard.tsx`) wird überprüft, welche Nebenräume zu dem ausgewählten Hauptraum passen und ob diese auch die Filterkriterien erfüllen. Es wird erneut gefiltert und die Ergebnisse werden in der Raumliste angezeigt und im 3D Modell markiert und auswählbar gemacht.
- Im vierten und letzten Schritt (`RoomSummaryWizard.tsx`) wird der reservierte Haupt- und Nebenraum in das `filteredMeshes Array` hinzugefügt, damit diese auch im 3D Modell markiert werden und auswählbar sind.
- Nach der Buchung wird das `filteredMeshes` Array geleert, damit im ersten Schritt keine Räume mehr hervorgehoben sind. Jedoch sind im ersten Schritt trotzdem alle Räume auswählbar.

`selectedMeshes`: wird verwendet, um zu erkennen, welcher Raum der aktuelle, ausgewählte Raum ist.

- Wird aus der Menge der passenden, wählbaren Räume des `filteredMeshes Arrays`, ein Raum durch das Anklicken im 3D Modell oder das Öffnen eines Accordion Elementes ausgewählt, so wird dieser in das `selectedMeshes Array` hinzugefügt und im 3D Modell farblich und von der Transparenz von den anderen Räumen des `filteredMeshes Arrays` hervorgehoben.
- Dadurch soll der Nutzer erkennen können, welche Räume auswählbar sind und welcher Raum bereits ausgewählt ist.
- Der Farb- und Transparenzwert der Räume im `selectedMeshes Array` ist der initiale Wert. Daher sind im zweiten, dritten und letzten Schritt alle Räume, welche `im filteredMeshes Array` sind, auch im `selectedMeshes Array`, solange kein Raum ausgewählt wurde.

7 Kompatibilität

7.1 Browser Kompatibilität

Im Folgenden wird der Browser Support für React, ThreeJS und WebGL aufgelistet. Da ThreeJS auf allen Browsern laufen sollte, die WebGL supporten, ist der Support von WebGL entscheidend.

React Browser Support:
"React supports all popular browsers, including Internet Explorer 9 and above, although some polyfills are required for older browsers such as IE 9 and IE 10."

ThreeJS Browser Support:
"Three.js runs in all browsers supported by WebGL."
Quelle: https://en.wikipedia.org/wiki/Three.js

WebGL Browser Support:
Die Browser, welche WebGL supporten werden in der unteren Abbildung 11 dargestellt und konnten auf der Seite https://caniuse.com/webgl ermittelt werden. Die Seite wurde das letzte Mal am 29.01.2022 geupdated.

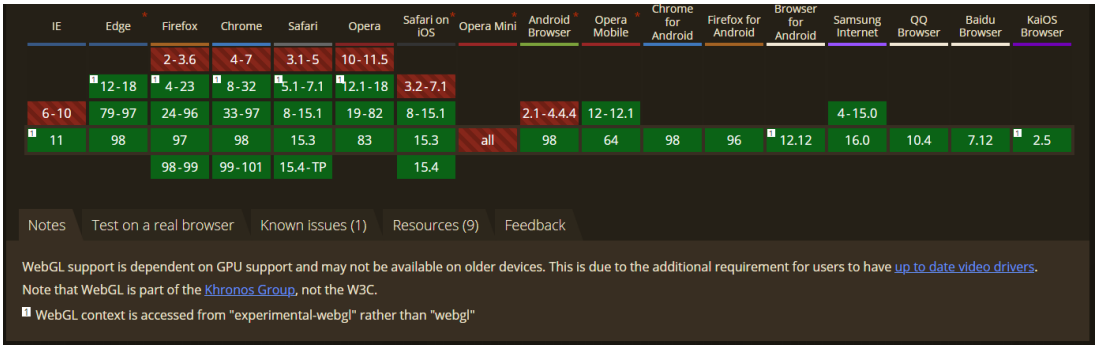


Abbildung 11: WebGL Browser Support

7.1.1 Tests an echten Geräten

DELL PRECISION | M4700

- Modell: Precision M4700 (Erscheinungsjahr: 2012)
- Prozessor: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz 2.70 GHz
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
- Betriebssystem: Windows 10 Pro (Version 21H1)
- Getesteter Browser: Firefox (97.0), Chrome (98.0.4758.82), Microsoft EdgeVersion (98.0.1108.43)
  Ergebnis: Funktioniert fehlerfrei

13‘‘ MacBook Pro

- Modell: MacBook Pro (Erscheinungsjahr: 2006)
- Prozessor: Dual-Core Intel Core i5 2.6 GHz
- Betriebssystem: macOS 10.15.7 (19H2)
- Getesteter Browser: Safari (14.0), Chrome (98.0.4758.80)
  Ergebnis:
  In Safari scheinen bei manchen Klassen wie den Accordion Items simple CSS-Eigenschaften wie `width: 100%;` nicht richtig zu funktionieren. Das muss ein Safari Problem sein, da es bei demselben Gerät auf Chrome nicht auftritt. Der Grund hierfür ist noch unklar, aber das Problem wurde temporär für alle Viewport ab `993px` mit einer festen Pixel-Breite für die Accordion behoben. Bei den Viewports von `481px` bis `992px` muss noch eine Lösung gefunden werden.
  Bis auf das Problem mit der `width` CSS-Eigenschaften in Safari, funktioniert die Applikation auf Safari und Chrome fehlerfrei.

7.2 Mobile Kompatibilität

Auf den Mobilegeräten, die die Applikation nicht richtig rendern / öffnet konnten, wurde zusätzliche versucht, ein einfaches Standartbeispiel der https://threejs.org Seite zu öffnen. Dieses zeigt einige simple 3D Objekte, welche mit Hilfe der `OrbitControls` geschwenkt werden können. Falls dieses Beispiel auch nicht funktioniert, wird das Problem höchstwahrscheinlich am WebGL Renderer oder der Hardware des Gerätes liegen.
Das ThreeJS Standartbeispiel ist hier zu finden: https://threejs.org/examples/?q=orbit#misc_controls_orbit

7.2.1 Tests an echten Geräten

Mobistel Cynus T1

- Modell: Cynus T1
- Android-Version. 4.1.1
- Getesteter Browser: Chrome (Version 42.0)
- **Ergebnis: Funktioniert nicht. Webseite wird nicht gerendert. Standartbeispiel von ThreeJS funktioniert auch nicht.**

Sony Xperia

- Modell: ST21i
- Android-Version. 4.0.4
- Chipset: Qualcomm MSM7225AA Snapdragon S1
- CPU: 800 MHz Cortex-A5
- GPU: Qualcomm Adreno 200
- Getesteter Browser: Android Browser (4.0.4)
- **Ergebnis: Funktioniert nicht. Webseite wird nicht gerendert. Standartbeispiel von ThreeJS funktioniert auch nicht.**

Samsung Galaxy S4

- Modell: GT-i9515
- Android-Version: 5.0.1
- Chipset: Qualcomm APQ8064T Snapdragon 600
- CPU: 1.9GHz Quad-Core Krait 300 (1 Prozessor)
- GPU: Qualcomm Adreno 320
- Getesteter Browser: Chrome
- **Ergebnis: Funktioniert fehlerfrei (jedoch langsam).**

HUAWEI Mobile P10 Lite

- Modell: WAS-LX1A
- Android-Version: 8.0.0
- Getesteter Browser: Chrome (Version 98.0)
- **Ergebnis: Funktioniert fehlerfrei.**

RealMe 6

- Modell: RMX2001
- Android-Version: 11.0.0
- Getesteter Browser: Chrome (Version 98.0)
- **Ergebnis: Funktioniert fehlerfrei**.

Apple iPod Touch

- Modell: MC544FD/A
- iOS-Version: 6.1.6
- Getesteter Browser: Safari 2.0
- **Ergebnis: Funktioniert nicht. Webseite wird nicht gerendert. Standartbeispiel von ThreeJS funktioniert auch nicht.**

Apple iPhone 8 Plus

- Modell: MQ8N2ZD/A
- iOS-Version: 12.4.1
- Getesteter Browser: Safari (Version 12)
- **Ergebnis**:
  - ***Kein ResizeObserver Support***
    Zunächst funktionierte das Laden der Seite nicht. Über `localhost:3000` wurde die Fehlermeldung "`Error: This browser does not support ResizeObserver out of the box.`" angezeigt. Diese konnte behoben werden, durch den Import und das Setzen vom **Resize-Observer** von der `@juggle/resize-observer` Library in der `<Canvas> react-three-fiber` Komponente.
    Issue und die Lösung: https://github.com/pmndrs/react-three-fiber/issues/248
  - ***Deaktivierte Pointer-Events***
    Danach konnte die Seite erfolgreich geladen werden, jedoch funktionierten die **Pointer-Events** mit dem Modell nicht. Dafür mussten in den Einstellungen die Pointer Events aktiviert werden:
    `Einstellungen > Safari > Erweitert > Experimental Features > Pointer Events (enable)`
  - ***Probleme beim ThreeJS Standartbeispiel***
    Das Standartbeispiel von ThreeJS wird gerendert aber jegliche `OrbitControls` Interaktionen, wie in der Szene zoomen, das Modell rotieren oder die Kamera schwenken sind nicht möglich.
  - Auswahl und Konfiguration, sowie die Steuerung des Modelles durch den Wizard funktionieren dagegen fehlerfrei.

Apple iPhone 7

- Modell: MN922ZD/A
- iOS-Version: 14.0.1
- Getesteter Browser: Safari (Version 14)
- **Ergebnis: Funktioniert fehlerfrei.**

iPhone X

- Modell: NQAC2ZD/A
- iOS-Version: 14.8.1
- Getesteter Browser: Safari (Version 14), Chrome (Version 98.0.4)
- **Ergebnis**: **Funktioniert fehlerfrei.**

iPhone 11

- Modell: MWM22ZD/A
- iOS-Version: 15.2.1
- Getesteter Browser: Safari (Version 15)
- **Ergebnis**: **Funktioniert fehlerfrei.**

7.3 Tablet Kompatibilität

7.3.1 Tests an echten Geräten

Galaxy Tab S2

- Modell: SM-T813
- Android-Version. 7.0
- Getesteter Browser: Chrome (Version 97.0.4)
- **Ergebnis: Funktioniert fehlerfrei.**

iPad Air

- Modell: MD794GP/A
- iOS-Version. 12.5.4
- Getesteter Browser: Safari (Version 12),
- **Ergebnis:**
  - **Deaktivierte Pointer-Events:**
    Zu Beginn funktionierten die Pointer-Events mit dem Modell nicht (genauso wie beim iPhone 8 Plus). Um das zu beheben, müssen in den Einstellungen die Pointer Events aktiviert werden:
    `Einstellungen > Safari > Erweitert > Experimental Features > Pointer Events (enable)`
  - **Danach funktioniert alles fehlerfrei.**

iPad Pro (12,9 Zoll, 3. Generation)

- Modell: MTHV2TY/A
- iOS-Version. 15.2.1
- Getesteter Browser: Safari (Version 15), Chrome, Microsoft Edge
- **Ergebnis: Funktioniert fehlerfrei.**

7.4 Fazit

7.4.1 Mobile

Leider konnten nicht alle Android und iOS Versionen getestet werden. Jedoch wurde bei Android eine klare Grenze ermittelt, ab welcher der ThreeJS Inhalt nichtmehr gerendert wird und die App nicht funktioniert. Diese Grenze lag zwischen den beiden Android 4 Geräten Mobistel Cynus T1 (Android-Version. 4.1.1) und Sony Xperia (Android-Version. 4.0.4), bei welchen die Applikation noch auf dem Android und Chrome Browser fehlschlug, und dem Android 5 Gerät Samsung Galaxy S4 (Android-Version: 5.0.1), bei dem die App zwar langsam lief, aber funktionierte. Bei iOS konnte der Apple iPod Touch (iOS-Version: 6.1.6) die App auf Safari 2.0 nicht rendern, während das Apple iPhone 8 Plus (iOS-Version: 12.4.1), nach dem Hinzufügen eines Resize-Oberserver zu der `React-Three-Fiber` Canvas Komponente im Code und dem Aktivieren von Pointer-Events in den Safari Einstellungen, die App erfolgreich auf Safari 12 gerendert hat. Jedoch zeigten sich auch hier erste Schwierigkeiten mit den `OrbitControls` des Standart-Beispiels aus ThreeJS. Für alle neueren, getesteten Geräte mit Android 5 und iOS 12 aufwärts, funktionierte die Applikation problemlos.

7.4.2 Tablet

Beim Kompatibilitätstest der Tablets funktionierten alle drei Geräte problemlos. Jedoch mussten beim iPad Air (iOS-Version 12.5.4) genauso wie beim Apple iPhone 8 Plus, die Pointer Events in den Safari Einstellungen aktiviert werden. Es lässt sich vermuten, dass dieses Problem in späteren iOS-Versionen behoben wurde, da die Pointer-Events bei den anderen Apple Geräten mit höheren iOS-Version nicht mehr aktiviert werden mussten.

7.4.3 Browser

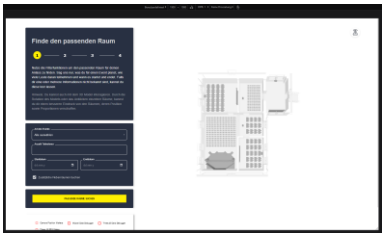
Getestet wurden zwei Laptops; Dell Precision M4700 (2012) und 13‘‘ MacBook Pro (2006). Beide hatten kein Problem das 3D Modell zu rendern und die Interaktionen mit dem Modell und den `OrbitControls` korrekt auszuführen. Jedoch hatte Safari beim MacBook Pro Probleme mit einige CSS-Eigenschaften, während Chrome auf dem MacBook diese nicht hatte. Zusammenfassend für Desktop, Mobile und Tablet, müssen für ältere Geräte und für den Safari Browser noch einige Anpassungen gemacht werden, damit die Geräte, welche von der Hardware stark genug sind, die Applikation fehlerfrei ausführen können.

8 Responsiveness

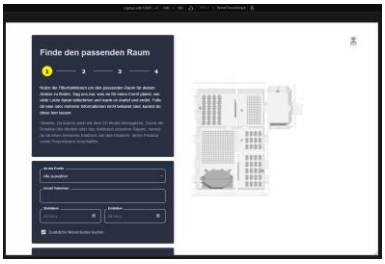
Die Herausforderung bei der Responsiveness war, das Modell für jeden Viewport und für jedes Gerät so zu positionieren, dass es gut sichtbar ist, aber nicht zu sehr die Liste mit den Accordion Elementen / den auswählbaren Räumen verdeckt. Die Liste sollte im Vordergrund sein und die Visualisierung des 3D Modelles sollte weiterhin nur als Unterstützung dienen. Daher wurde das 3D Modell ab einer Bildschirmbreite von `1240 Pixel` und weniger (`md` bis `lg`) als ein überliegendes Karten Element mit einer Höhe von `40 vh` oberhalb der Liste positioniert. Die Höhe der Karte wird dabei mit sinkender Bildschirmbreite runterskaliert.
Bei einer Bildschirmbreite von `768 Pixel` und weniger (`xs` bis `sm`) beträgt die Höhe der Karte `30 vh`.
Bei einer Bildschirmbreite von `481 Pixel` und weniger beträgt die Höhe der Karte `25 vh`.
Im Folgenden werden Screenshots der einzelnen Ansichten für Desktops, Tablets und Mobilegeräte von einzelnen Geräten aus dem „`Bildschirmgrössen testen`“ Werkzeug von Firefox gezeigt.

8.1 Desktop

Large Desktops (1080p Full HD Television): 1900 x 1080



Medium Desktops (Laptop with HiDPI screen): 1440 x 900



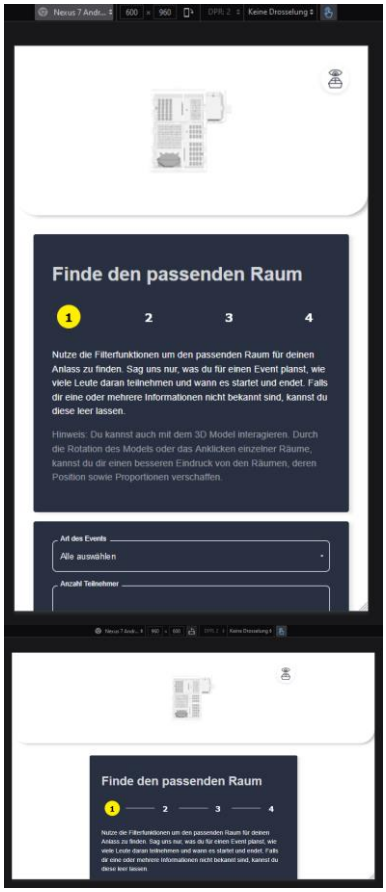
Smaller Desktop (Laptop with MDPI screen): 1280 x 800

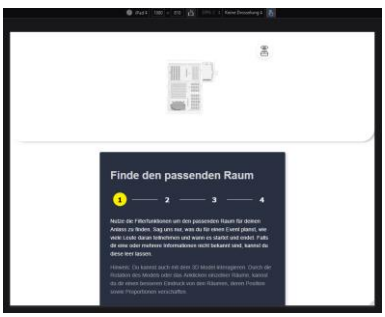
8.2 Tablet horizontal & vertikal

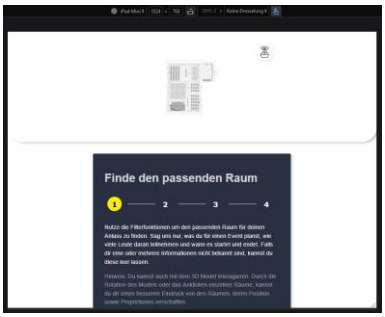
Small Apple Tablet (iPad Mini): 768 x 1024 / 1024 x 768

Medium Apple Tablet (iPad): 810 x 1080 / 1080 x 810

Large Tablet (Nexus 7): 600 x 960 / 960 x 600



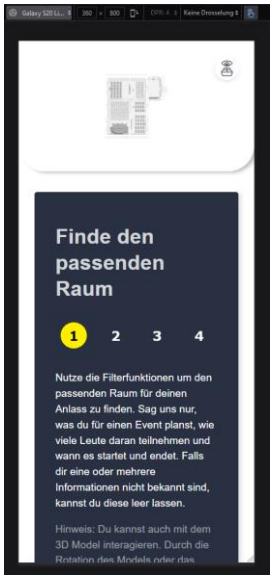




8.3 Mobile

Small Mobile (Galaxy S5): 360 x 640

Medium Android Mobile (Galaxy S20): 360 x 800



Medium Apple Mobile (iPhone 11 Pro): 375 x 812

Big Mobile (Galaxy S20 Ultra): 412 x 915

9 Tools und Packages

Im Folgenden werden alle verwendeten Packages im React Projekt aufgelistet.
Diese sind auch in der `package.json` Datei im Projekt zu finden.
Aufgelistet werden: der Name des Packages, dessen Lizenz, dessen Beschreibung, sowie die verwendete Version im Projekt. In manchen Fällen wird auch die Stelle, an der das Package verwendet wurde und dessen Nutzen dort erläutert.
Die Lizenzen und Beschreibungen wurden alle von der Seite `https://www.npmjs.com` verwendet.
Bis aus typescript und @juggle/resize-observer haben alle Packages eine MIT Lizenz.

Package: **typescript**
Version: **4.4.4**
Licence: **Apache-2.0**
Description: TypeScript is a language for application-scale JavaScript. TypeScript adds optional types to JavaScript that support tools for large-scale JavaScript applications for any browser, for any host, on any OS. TypeScript compiles to readable, standards-based JavaScript.

Package: **@emotion/react**
Version: **11.7.1**
Licence: **MIT**
Description: Simple styling in React.

Package: **@emotion/react**
Version: **11.6.0**
Licence: **MIT**
Description: The styled API for @emotion/react.

Package: **@mui/lab**
Version: **5.0.0-alpha.60**
Licence: **MIT**
Description: This package hosts the incubator components that are not yet ready to move to core.
Usage: DatePicker, LocalizationProvider, AdapterDateFns in RoomFilteringWizard.tsx.

Package: **@mui/material**
Version: **5.2.4**
Licence: **MIT**
Description: MUI is a simple and customizable component library to build faster, beautiful, and more accessible React applications.
Usage: ThemeProvider in App.js, createTheme in theme.tsx, Stepper, Step, StepButton in RoomSelection.tsx, FormControlLabel, Checkbox in DebugControlPanel.tsx, FormControlLabel, Checkbox, TextField, MenuItem in RoomFilteringWizard.tsx, TextField in RoomSummaryWizard, Tooltip in ModelCanvas.tsx & MeshVisibilityButton.tsx

Package: **@react-three/drei**
Version: **7.13.1**
Licence: **MIT**
Description: A growing collection of useful helpers and fully functional, ready-made abstractions for react-three-fiber.
Usage: OrbitControlsProps, Stats in ModelCanvas.tsx, Line in CameraPositionMarkers.tsx, Center, useGLTF in Model.tsx

Package: **@react-three/fiber**
Version: **7.0.12**
Licence: **MIT**
Description: react-three-fiber is a React renderer for threejs.
Usage: Canvas, PerspectiveCameraProps in ModelCanvas.tsx, GroupProps in Model.tsx

Package: **Three**
Version: **0.133.1**
Licence: **MIT**
Description: The aim of the project is to create an easy to use, lightweight, cross-browser, general purpose 3D library. The current builds only include a WebGL renderer but WebGPU (experimental), SVG and CSS3D renderers are also available in the examples.
Usage: * as THREE in ModelCanvas.tsx, * as THREE, GLTF in Model.tsx & roomData.ts & useCameraStore.ts

Package: **@juggle/resize-observer**
Version: **3.3.1**
Licence: **Apache-2.0**
Description: A minimal library which polyfills the ResizeObserver API. It immediately detects when an element resizes and provides accurate sizing information back to the handler.
Usage: resize-observer in three-js-fiber <Canvas> component in ModelCanvas.tsx

Package: **zustand**
Version: **3.6.5**
Licence: **MIT**
Description: A small, fast and scalable bearbones state-management solution using simplified flux principles.
Usage: create method in useCameraStore.ts, useDebugStore.ts, useMeshStore.ts, useWizardStore.ts

Package: **classnames**
Version: **2.3.1**
Licence: **MIT**
Description: A simple JavaScript utility for conditionally joining classNames together.
Usage: cn in AccordionItem.tsx, cn in MeshVisibilityButton.tsx

Package: **date-fns**
Version: **2.27.0**
Licence: **MIT**
Description: date-fns provides the most comprehensive, yet simple and consistent toolset for manipulating JavaScript dates in a browser & Node.js.
Usage: deLocale in RoomFilteringWizard.tsx (necessary for MUI DatePicker)

Package: **sass**
Version: **1.44.0**
Licence: **MIT**
Description: A pure JavaScript implementation of Sass.

Packages die mit dem initialen Installieren von `create-react-app` automatisch dabei sind.

Package: **react**
Version: **17.0.2**
Licence: **MIT**
Description: React is a JavaScript library for creating user interfaces.
Usage: Used in multiple places inside the project for: Suspense, useRef, useState, useEffect, useMemo

Package: **react-dom**
Version: **17.0.2**
Licence: **MIT**
Description: This package serves as the entry point to the DOM and server renderers for React. It is intended to be paired with the generic React package, which is shipped as `react` to npm.
Usage: ReactDOM in index.js

Package: **react-scripts**
Version: **5.0.0**
Licence: **MIT**
Description: This package includes scripts and configuration used by Create React App.

10 Credits für 3D Modelle

HOUSE MODEL
* title: Village house low-poly 3d model
* source: https://www.cgtrader.com/free-3d-models/exterior/house/village-house-497361f5-86cb-4e35-9ec5-f49a88ea8112
* author: Chiliaz (https://www.cgtrader.com/chiliaz)
* license type: Royalty Free License

CHAIR MODEL
* title: Low Poly Chair
* source: https://sketchfab.com/3d-models/low-poly-chair-7344e73dbb5b456e84f15b0dacd0f907
* author: Jeremy E. Grayson (https://sketchfab.com/JeremyGrayson)
* license type: CC-BY-4.0

TABLE MODEL
* title: Office Table
* source: https://sketchfab.com/3d-models/office-table-2e298de5176540f8bdf5633842604806
* author: Konstantin Koretskyi (https://sketchfab.com/koklesh)
* license type: CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)

PODIUM MODEL
* title: Podium
* source: https://sketchfab.com/3d-models/podium-e228f4566ef54a268a16a538a1cd1f74
* author: awa (https://sketchfab.com/awa)
* license type: CC-BY-4.0

PROJECTOR MODEL
* title: Projector Set
* source: https://sketchfab.com/3d-models/projector-set-18428ceaffee47f5b9e90219adbfbefd
* author: comphonia (https://sketchfab.com/comphonia)
* license type: CC-BY-4.0

CONCERT STAGE MODEL
* title: Concert Stage
* source: https://sketchfab.com/3d-models/concert-stage-8074957c9ad24d878d2c0b91b41339b7
* author: wessel.huizenga (https://sketchfab.com/Wessel.Huizenga)
* license type: CC-BY-NC-4.0

11 Ausblick und Ideen zur Verbesserung

11.1 Einleitendes Tutorial

Bei steigender Komplexität der Applikation, wäre es denkbar, dass ein initiales Popup Modal eingeblendet wird. Dieses könnte in Form einer kurzen Video- oder GIF-Sequenz die einzelnen Wizard Schritte und deren Funktionen sowie Möglichkeiten demonstrieren. Dadurch könnte der Nutzer in kurzer Zeit den Workflow des Wizards und dessen Interaktion mit dem Modell verstehen und hätte weniger Berührungsängste. Dieses Modal sollte nur bei der aller ersten Buchung erscheinen und jederzeit wegklickbar sein. Das hätte auch den Vorteil, dass die Beschreibungen der Wizard Schritte entfernt werden können und es somit mehr Platz für die Raum-Listenelemente gibt. Bei den Usability Tests wurde diese Idee vorgeschlagen und bekam sehr viel Zuspruch. Zusätzlich wurde beobachtet, dass die getesteten Nutzer sehr selten und ungerne die Beschreibungen lesen, da sie sich häufig selbst als visuelle Menschen beschreiben. Dadurch wurden einige Funktionen nicht ausprobiert und andere benötigten eine kurze Zeit, bis Sie verstanden wurden.

11.2 Backend Anbindung

In dem Prototyp wird das Backend durch ein File mit allen relevanten Daten gemockt (`roomData.ts`). Würde jedoch zukünftig eine echte Backend Anbindung implementiert werden, so müsste zunächst die Frage beantwortet werden, ob alle Daten auf der Clientseite sichtbar sein sollen, oder ob beispielsweise die Filtrierung der Räume auf der Serverseite passieren soll, sodass nur die filtrierten Räume zurückgegeben werden. So könnte auf der Clientseite beispielsweise nicht ausgelesen werden, zu welchem Datum, welcher Raum ausgebucht ist. Eine andere Herausforderung wäre, dass hochladen / updaten der Rauminformationen mit den neuen Buchungsdaten der Buchungen und das Übertragen der Buchungsinformationen an den Server.

11.3 Positionierung des Equipments im Raum

Neben der Auswahl von dem Raumkonfigurationen, kann es auch interessant sein, diese direkt im Raum zu positionieren und auszurichten. Bei manchen Use-Cases würde es mehr Sinn machen als bei anderen. So wäre die Positionierung und Ausrichtung einer Bühne weniger realistisch als die einer Leinwand mit Beamer, eines Podestes oder einer Stuhlformation. Hierzu existieren bereits die `DragControls` für simple Drag'n'Drop Interaktionen von `ThreeJS` oder `TransformControls` für fortgeschrittenere Skalier- Rotations- und Transformationsinteraktionen von `React-Three-Drei`.
Sandbox für `DragControls` in einer `React-Three-Fiber` Applikation:
https://codesandbox.io/s/react-three-fiber-orbitcontrols-kf5wl?from-embed=&file=/src/index.js:160-172
Sandbox für `TransformControls` in einer `React-Three-Fiber` Applikation:
https://codesandbox.io/s/r3f-drei-transformcontrols-hc8gm?from-embed

11.4 Auswahl mehrerer Nebenräume

Die Auswahl mehrerer Nebenräume wurde absichtlich aus dem Scope des Prototyps rausgenommen, da diese eine deutlich höhere Komplexität in den Wizard einführen würde. Jedoch müsste an der Logik des 3D Modelles nicht viel angepasst werden, dass diese Funktion möglich wäre. Die globale Property `activeSideRoom` müsste zu einem `Array` gemacht werden und die Accordion Elemente, sowie das Modell bräuchten eine Flag zur Erkennung, ob nur ein Raum oder mehrere Räume ausgewählt werden können. Ansonsten bliebe die Logik der Markierung der Räume mit den `selectedMeshes` und den `filteredMeshes` dieselbe. Es könnte sogar eine neue Kameraposition und Blickrichtung definiert werden, welche speziell die ausgewählten Nebenräume am besten anzeigt. Dazu könnte der Mittelpunkt aller ausgewählten `camPos` und `camTarget` Punkte berechnet oder einfach die Übersicht der Räume angezeigt werden.

11.5 Performance Verbesserungen

Beim Rendering von 3D Modellen im Web ist Performance ein sehr kritisches Thema. In den Prototypen wurde speziell darauf geachtet, möglichst nur Low Poly 3D Modelle mit sehr wenigen Triangles und Vertexes zu verwenden und auf komplexe Texturen oder Materials zu verzichten. Stattdessen wurden eigene, sehr simple StandardMaterials direkt im Code erstellt und den jeweiligen Meshes zugewiesen. Diese haben den Vorteil, dass sie deutlich einfacher, dynamisch durch einen eigenen State kontrolliert und verändert werden können, wie im Kapitel 6.5 Interaktionen mit dem Modell beschrieben wurde.

Bevor die Performance verbessert wurde, stieg die Ladezeit auf bis zu 15 Sekunden. Gründe hierfür waren:

- ein 3D Stuhl Mesh mit einer sehr hohen Anzahl an Vertices und Triangles (11.400), welche zusätzlich mehrfach in der Szene dupliziert wurden. Auch wenn das Mesh sehr simpel aussah, entstanden durch Rundung und Kurven eine deutlich höhere Anzahl an Triangles und Vertices als erwartet. Durch erneute Recherche und Überprüfung aller 3D Objekte und ihrer Performance Lastigkeit, konnten einige Rendering intensive Objekte ausgetauscht werden. Darunter wurde auch das Stuhl Mesh durch ein deutlich simpleres Mesh mit nur 472 Triangles und 246 Vertices ersetzt. Auch wurden ungenutzte Materials und Texturen bereinigt, damit nur das geladen wird, was geladen werden muss.

- ein zusätzliches `<lineSegment>` Mesh, welches in jedem 3D Objekt zusätzlich erstellt worden ist und eine Reihe von Linien zwischen Paaren von Vertices zeichnet und dadurch dem Modell eine Outline verleiht. Dieses Mesh erwies sich als sehr Performance lastig, da es die gesamte Geometrie eines Objektes nachfahren muss und diese mittels eines eigenen `<lineBasicMaterial>` einfärbt. Daher müssen doppelt so viele Meshes und Geometrien gerendert werden (Code Ausschnitt 9).

```
<lineSegments>
    <edgesGeometry attach='geometry' args={[meshObject.geometry]} />
    <lineBasicMaterial color='black' attach='material' transparent />
</lineSegments>
```

Code Ausschnitt 9: <lineSegments> Code in Model.tsx

Um dieses Problem zu lösen, wurde das `<lineSegments>` Mesh durch ein simples, weniger Performance lastigeres Material `<meshStandardMaterial>` ersetzt. Es konnte nicht dieselbe Optik erreicht werden, doch durch komplexe Beleuchtungen der Szenerie, konnte auch bei dem `<meshStandardMaterial>` der Kontrast so angepasst werden, dass das einfachere Material auch funktioniert. Einen Vergleich vom `<lineSegments>` Mesh und dem `<meshStandardMaterial>` ist in der Abbildung 12 zu sehen.

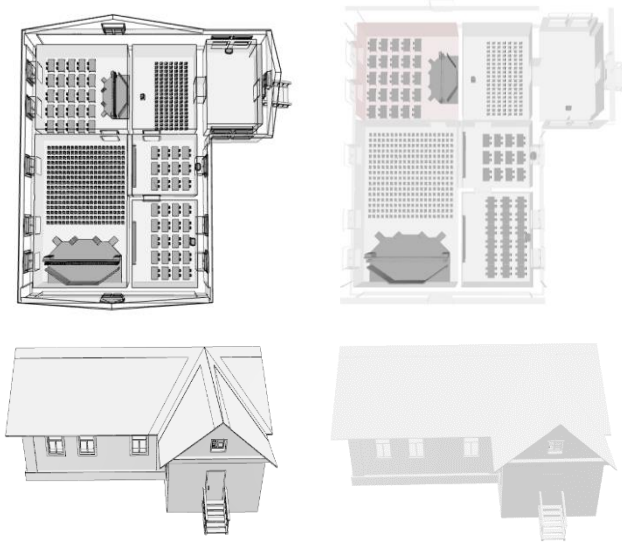


Abbildung 12: Vergleich mit <lineSegments> (links) und ohne <lineSegments> (rechts)

Es ist jedoch denkbar, dass mit weiteren, professionellen Optimierungen des 3D Modelles, auch das Rendering intensive `<lineSegments>` Mesh verwendet werden könnte. Dennoch muss auch bedacht werden, dass bei steigender Komplexität des Modelles (mehr Räume, mehr Konfigurationsmöglichkeiten), der zusätzliche Performance Aufwand des `<lineSegments>` Mesh im praktischen Kontext nicht mehr realistisch ist.

11.6 Umgang mit längeren Renderzeiten

Problem:
Im Normalfall, wird ein `Spinner` oder `Loader` gezeigt, wenn etwas lange laden muss. Das wurde in der Applikation zum Beispiel bei dem Laden der einzelnen Raumelemente in der Raumliste implementiert. Es wurde versucht, die gleiche Logik für das Rendering des Modelles zu implementieren, jedoch wurde festgestellt, dass die gesamte Applikation einfriert, solange das Modell gerendert wird. Das Laden des Modelles aus dem GLTF File und das Konvertieren der geladenen Daten in die eigene Datenstruktur nimmt kaum Zeit in Anspruch. Aber das Rendern des gesamten Modelles ist sehr zeitintensiv und beansprucht alle Ressourcen. Entsprechend können keine Animationen von einem `Spinner` oder `Loader` gezeigt werden und auch keine Änderungen im UI visualisiert werden. Auch wenn die `EventHandler` auf Interaktionen reagieren, werden diese verzögert, nach dem Rendering im UI visualisiert.

Versuchte Lösungen:
Um dieses Problem zu lösen, wurden zwei Ansätze ausprobiert. `Async Functions`, welche den Rendering Prozess aufrufen und `Lazy-Loading`. Beide lösten das Problem nicht. Auch die `Web Workers` Klasse von React, welche Aufgaben im Background Thread erledigt, ohne dabei das User Interface zu beeinflussen, ist keine Option, da in diesem keine Rendering Prozesse ausgeführt werden können.

Einfachere Lösung (Statisches HTML):
Eine einfachere Lösung, wäre das Anzeigen von statischem HTML, was dem Nutzer erklärt, dass das Modell noch laden muss. Zusätzlich können auch alle `EventHandler` deaktiviert werden, solange das Modell lädt. Jedoch muss dafür der Zeitpunkt, an welchem das Modell fertig geladen hat, in der React Applikation kommuniziert werden. Hierfür bietet React keine direkte Lösung. Es lässt sich tracken, wann eine Komponente das erste Mal initialisiert wurde (`componentDidMount` oder `useEffect(()=>{},[])`) oder wenn diese geupdatet wurde (`componentDidUpdate` oder `useEffect(()=>{},[trackingValue])`), aber nicht, wann diese das erste Mal vollständig fertig gerendert wurde. So wird `useEffect(()=>{},[])` schon aufgerufen, bevor das Rendering des Modelles fertig ist. Hierzu bietet `ThreeJS` die `onAfterRender` Funktion an, welche ein Callback zurückgibt, direkt nach dem ein 3D Objekt gerendert wurde. Diese Funktion wurde noch nicht ausprobiert, könnte aber hilfreich sein.
Quelle: https://threejs.org/docs/#api/en/core/Object3D.onAfterRender