FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Webová komponenta pro interaktivní vizualizaci 3D modelů
Student:	Bc. Pavel Antoš
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

V rámci vznikající aplikace pro management 3D modelů vznikla potřeba pro realizaci universální webové komponenty, pro interaktivní vizualizaci 3D modelů.

1. Analyzujte vybrané aplikace pro vizualizaci 3D modelu v prostředí webu, zaměřte se na jejich funkcionalitu a uživatelskou přívětivost.
2. Analyzujte funkční a nefunkční požadavky zadavatele na vizualizační komponentu 3D modelů.
3. Pomocí metod softwarového inženýrství navrhněte webovou komponentu splňující požadované funkčnosti.
4. Implementujte prototyp webové komponenty pod knihovnou React.
5. Prototyp otestujte vhodnými uživatelskými a akceptačními testy a testování vyhodnoťte.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 6. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Webová komponenta pro interaktivní vizualizaci 3D modelů

Bc. Pavel Antoš

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Chludil

4. května 2023

Poděkování

Rád bych poděkoval vedoucímu své práce Ing. Jiřímu Chludilovi za ochotu a cenné rady.

Dále bych chtěl poděkovat své rodině za podporu při studiu FIT ČVUT.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2023 . . . . . .

Odkaz na tuto práci

Antoš, Pavel. *Webová komponenta pro interaktivní vizualizaci 3D modelů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato diplomová práce je zaměřena na tvorbu universálních webových komponent pro zobrazení 3D modelů. Výsledkem je knihovna, která obsahuje komponentu pro zobrazení jednoho 3D modelu, komponentu pro zobrazení dvou modelů v jedné 3D scéně a komponentu pro zobrazení dvou modelů ve dvou synchronizovaných scénách.

Součástí této práce je analýza funkčních a nefunkčních požadavků, průzkum již existujících aplikací využívajících zobrazení 3D modelů a analýzu technologií vybraných pro implementaci. Dále byl vytvořen návrh knihovny zaměřený na vztahy mezi moduly a způsob porovnávání dvou 3D modelů. V rámci implementace byl popsán způsob tvorby klíčových modulů, způsob sestavení, nasazení, instalace a integrace výsledné knihovny.

Po dokončení implementace byla knihovna otestována. Byla vytvořena testovací aplikace, která byla podrobena uživatelským testům. Knihovna samotná byla podrobena akceptačním testům.

Klíčová slova webová komponenta, vizualizace 3D modelů, React, Three.js, TypeScript

Abstract

This thesis focuses on the creation of universal web components for visualizing 3D models. The result is a library that includes a component for visualizing one 3D model, a component for visualizing two models in one 3D scene and a component for visualizing two models in two synchronized scenes.

This work includes an analysis of functional and non-functional requirements, a survey of existing applications using 3D model visualization and an analysis of the technologies selected for implementation. In addition, a library design was developed focusing on the relationships between modules and how to compare two 3D models. The implementation describes the process of creating the key modules, how to build, deploy, install and integrate the resulting library.

After the implementation was completed, the library was tested. A test application was created and subjected to user testing. The library itself was subjected to acceptance tests.

Keywords web component, 3D model visualization, React, Three.js, TypeScript

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Webové aplikace v současnosti stále nabírají na popularitě. Jejich použití zasahuje i do oblastí, kde v minulosti byly výsadně použity instalované aplikace u klienta. Jednou z těchto oblastí je i práce s 3D modely.

Několik projektů v rámci FIT ČVUT se zabývalo manipulací s 3D modely v prostředí webových aplikací. Příkladem je bakalářská práce Pavla Antoše *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*[1] a projekt Ing. Jiřího Chludila a Ing. Petr Pauše Ph.D. *Lite průvodce*[2]. Zmíněná bakalářská práce využívá pro práci s 3D modely alternativu k *Three.js* a projekt *Lite průvodce* využívá framework *A-Frame*. Oběma projektům byly na základě zvolených technologií způsobeny problémy.

Three.js je knihovna pro zobrazení 3D obsahu a při práci s ní je třeba mít znalosti v oboru práce s 3D scénou. Relativně velké množství základních funkcionalit je třeba teprve implementovat. Příkladem může být absence jakékoli základní 3D scény, kdy je vždy vytvořena prázdná 3D scéna bez podlahy, bez pozadí, bez světel a bez kamery.

A-Frame je framework, který je postavený nad *Three.js*. Tento framework se snaží naopak co nejvíce zjednodušit práci s 3D scénou. To si ale vybírá svou daň ve formě slabé upravitelnosti. Jinými slovy, při nestandardním způsobu použití, lze jen těžko řešení upravit dle našich představ.

Pro *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*[1] to znamená, že samotná práce s 3D scénou byla velmi pracná. Práce na zbytku aplikace byla pak velmi časově vypjatá. Pro *Lite průvodce* to znamená omezené možnosti práce s 3D scénou a prakticky nulové možnosti rozšíření.

Výsledkem této diplomové práce by měla být knihovna, která nebude vyžadovat žádné znalosti 3D scény pro její použití. Tato knihovna bude obsahovat webové komponenty, které bude pro vývojáře (primárně FIT ČVUT) snadné integrovat do svých webových aplikací. Sice nebude poskytovat tolik možností jako *A-Frame*, ale bude ji možno rozšířit tak, že bude obsahovat

funkce „šité na míru“. Při nestandardních požadavcích bude možné knihovnu rozšířit, ať už formou vytvoření nových komponent, tak i formou rozšíření těch existujících.

Kapitola 1

Cíl práce

Cílem této diplomové práce je vytvoření universální webové komponenty pro interaktivní vizualizaci 3D modelů. Výstupem bude knihovna, která bude obsahovat již zmíněnou webovou komponentu a bude dostupná pro použití uvnitř *React* webových aplikací.

Analýza
Nejdříve je nutné provést analýzu již existujících řešení v prostoru webu. V této analýze je vhodné se zaměřit na funkcionalitu a uživatelskou přívětivost.

Dále je nutné provést analýzu funkčních a nefunkčních požadavků na webovou komponentu. Primárním zdrojem informací o požadavcích bude přímo zadavatel.

Pro potřeby návrhu je také vhodné provést analýzu vybraných technologií. Součástí budou technologie pro vývoj webových komponent a pro zobrazení 3D modelů.

Návrh
Po dokončení analytické části je nutné vytvořit návrh webové komponenty. Tento návrh bude respektovat metody softwarového inženýrství a bude proveden tak, aby výsledná komponenta splňovala zjištěné požadavky.

Implementace
Následně je nutné implementovat prototyp webové komponenty. Tento prototyp bude postaven na základech, které budou vymezeny v rámci návrhu. Z důvodu potřeby integrace s již existujícími aplikacemi je nutné použít knihovnu *React*.

Testování
Výsledný prototyp je pak vhodné otestovat. Testování bude provedeno v podobě uživatelských a akceptačních testů. Výsledky testování budou v rámci této práce vyhodnoceny.

Kapitola 2

Analýza

V rámci této kapitoly byly zjištěny požadavky na výslednou webovou komponentu a byla provedena analýza již existujících webových aplikací, které řeší podobnou problematiku. Dále byla také provedena analýza vybraných technologií pro implementaci webové komponenty.

2.1 Požadavky na webovou komponentu

Požadavky na webovou komponentu byly získány formou rozhovoru od zadavatele diplomové práce. Jsou rozděleny na funkční a nefunkční požadavky.

2.1.1 Nefunkční požadavky

N1: Kompatibilita s React webovými aplikacemi

Kompatibilita s *React* webovými aplikacemi je důležitá jak pro budoucí projekty, tak i ty, které již probíhají. Většina webových projektů, které vznikly v nedávné době, za spolupráce s Ing. Jiřím Chludilem, využívá knihovnu *React*. Pro budoucí projekty, za spolupráce s ním, je *React* primární volbou knihovny pro webovou aplikaci.

N2: Komunikace s již existujícím API

Primárním zdrojem 3D modelů je již existující API. Původní implementace API má kořeny v projektu *Věnná města českých královen*. Výsledná knihovna by měla být schopna korektně zobrazit 3D modely z tohoto zdroje.

N3: Uživatelská přívětivost

Aby výsledná knihovna byla použitelná v dalších projektech, je důležité, aby byla uživatelsky přívětivá. V kontextu této aplikace se jedná hlavně o ovládání kamery a o manipulaci s 3D scénou.

2.1.2 Funkční požadavky

F1: Podpora formátů FBX, GLB a OBJ

Tento požadavek navazuje na nefunkční požadavek na kominukaci s API. V rámci API jsou dostupné modely v několika formátech (*FBX*, *GLB* a *OBJ*). Obecně je většina existujících 3D modelů dostupná v těchto formátech.

F2: Pokročilé ovládání kamery

Je vhodné použít pokročilé ovládání kamery. Tento požadavek navazuje na nefunkční požadavek *N3: Uživatelská přívětivost*. Ne vždy je vhodné použít základní ovládání kamery ve stylu *first-person* 3D her (klávesy WASD). Pro zkoumání jednoho konkrétního 3D modelu se spíše nabízí použití takové kamery, která „obíhá“ kolem modelu.

F3: Variabilní prostředí uvnitř 3D scény

Tímto požadavkem je hlavně myšlena možnost nastavit jakou barvu bude mít 3D scéna (pozadí a zem). Možnost nastavení této barvy lze využít pro barevné sladění 3D scény a zbytku webové stránky.

F4: Podpora externích ovládacích prvků

Vývojář, využívající výslednou knihovnu ve své webové aplikaci, bude schopen si vytvořit své vlastní ovládací prvky pro 3D scénu. Pod tím si lze představit přepínače, které budou vytvořené vývojářem webové aplikace, které budou odpovídat jejímu stylu.

F5: Zobrazení dvou modelů do jedné 3D scény

Vyžadovanou funkcionalitou je možnost porovnání 3D modelů. Jedním z možných způsobů je zobrazení dvou modelů do jedné 3D scény.

F6: Synchronizované zobrazení 2 modelů do odlišných 3D scén

Dalším způsobem porovnání dvou 3D modelů je synchronizované zobrazení do dvou různých 3D scén. Obě scény by měli být synchronizovány alespoň na úrovni pohybu kamer.

F7: Příručka pro integraci

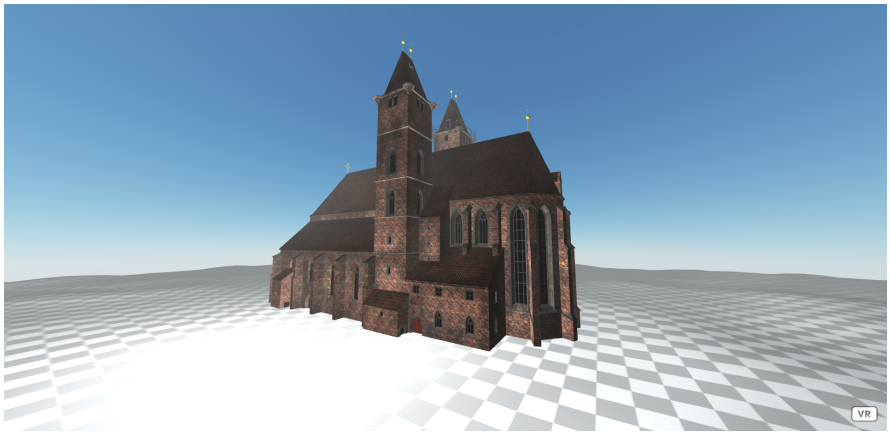
Je důležité aby vývojáři rozuměli, jak tuto knihovnou integrovat do vlastních webových aplikací. Proto je vhodné vytvořit příručku pro integraci, ve které bude vysvětleno, jak „vložit“ obsah knihovny (komponentu) do webové aplikace.

2.2 Již existující aplikace

Za účelem analýzy byly vybrány aplikace Lite průvodce a Webová aplikace pro schvalovací proces 3D modelů. Jedná se o webové aplikace, které implementují zobrazení 3D modelu na webové stránce.

2.2.1 Lite průvodce (Věnná města českých královen)

Webová aplikace Věnná města českých královen je určena pro PC, mobilní telefony a tablety. Demonstrační prototyp webové aplikace je součástí výstupů pro následující výstupy naplánované na rok 2022 (*Cesta časem po Hradci Králové bez 3D brýlí pro mobilní zařízení a web*). [3]



Obrázek 2.1: Lite průvodce

Prototyp webové aplikace využívá rozšířenou a virtuální realitu pro zobrazení 3D modelu historické budovy v reálné scéně (smíšená realita) nebo na jednoduchém podkladu (virtuální realita). Tento prototyp demonstruje načtení

modelu z objektové databáze, vložení do reálné nebo virtuální scény a jeho trackování na fixní poloze v reálném nebo virtuálním světě. [3]

Tento prototyp podporuje dva režimy kamery (letecký pohled a pěší pohled). U verze pro PC je pohyb implementován standardně pro *first-person* 3D hry. Jak pro pohyb „na zemi“ tak pro pohyb „ve vzduchu“ jsou používány klávesy WASD. V obou případech je pohyb zafixován na konkrétní výšku. Tím pádem je uživatel omezen na volbu dvou výšek, které nemůže dynamicky měnit.

U verze pro mobilní zařízení je základní ovládání omezeno jen na otáčení kamery, buď pomocí gyroskopu nebo pomocí potažení ve směru otočení kamery. V této verzi je možné využít dva dodatečné módy zobrazení a tím je AR a VR. Ovládání v AR režimu je přirozené (uživatel obchází model promítnutý do reálného prostoru). Ovládání ve VR režimu funguje totožně jako u standardního režimu.

Dle mého názoru nejlepším způsobem využití této aplikace je buď v AR nebo ve VR s ovladačem (otáčení kamery pomocí gyroskopu a pohyb kamery pomocí ovladače). Standardní režim pro PC je použitelný. Standardní režim pro mobilní zařízení je prakticky nepoužitelný kvůli chybějící možnosti pohybu. Oběma standardním režimům by dle mého názoru prospělo použití orbitální kamery.

Je nutné dodat, že „zážitek“ z používání mobilního zařízení v této aplikaci se může lišit v závislosti na použitém zařízení. Dle autorů, může být problematické použití ať už starého / výkonostně slabšího zařízení, tak i příliš nového / výkoného zařízení. Analýza byla provedena za použití *Xiaomi Mi 8 Lite 4GB/64GB* z roku 2018.

Volba technologií

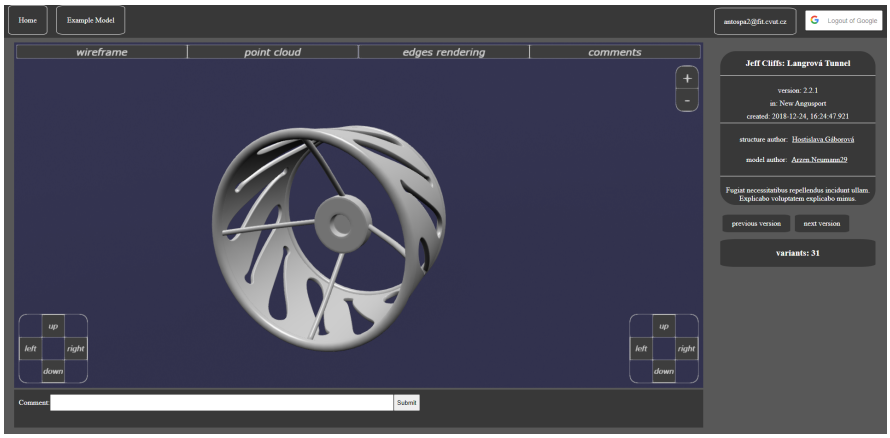
Prototyp aplikace je napsán v jazyce *HTML5* a *JavaScript* a využívá schopnosti moderních prohlížečů zobrazovat 3D modely a interagovat s nimi. Zobrazování 3D modelů a práce s virtuální a rozšířenou realitou je prováděno pomocí knihovny *A-Frame*. Pro verzování byl použit systém Git. Jako nosný formát pro 3D modely je použit formát *glTF 2.0*. [3]

Knihovna *A-Frame* je opensource knihovna od společnosti Mozilla založená na knihovně *Three.js*, která zpřístupňuje rozšířenou a virtuální realitu ve webovém prohlížeči pomocí standardu *WebXR*. Knihovna je vysokoúrovňová, tedy programátor nemusí řešit přímé zpracování obrazu z kamery nebo implementaci využití headsetu pro VR. V knihovně stačí sestavit scénu a tu následně zobrazit. [3]

Nevýhodou je, že programátor nemá přímý přístup ke kameře a nemůže tedy doprogramovávat zpřesňování lokace uživatele pomocí dalších dat. Zároveň je pro smíšenou realitu obtížné tvořit uživatelské rozhraní. Původně uvažovaný formát 3D modelů *FBX* není aktuálně podporován knihovnou *A-Frame*. [3]

2.2.2 Webová aplikace pro schvalovací proces 3D modelů

Jedná se o prototyp webové aplikace pro zjednodušení schvalovacího procesu, navrženého v rámci projektu Věnná města českých královen. Jejími uživateli jsou grafici, historici a modeláři. Tato aplikace komunikuje s již vytvořeným privátním API. Toto API je zdrojem 3D modelů (název, popisek, tvar, textura, ...) a připomínek vztažených k němu. [1]



Obrázek 2.2: Webová aplikace pro schvalovací proces 3D modelů[1]

Hlavní funkcionalitou této aplikace je načtení 3D modelu z privátního API a jeho zobrazení. Je nabízena i možnost přepínání verzí 3D modelu. Jelikož se jedná o aplikaci podporující schvalovací proces 3D modelů, tak důležitou součástí je možnost vytvořit připomínku, která je „připnuta“ do konkrétní oblasti 3D modelu. Doplňující funkcionalitou je možnost přepínaní zobrazení 3D modelu do módů (*wireframe*, *point cloud*, *edge rendering*).

Co se týče uživatelské přívětivosti, tak během uživatelského testování bylo zjištěno hned několik nedostatků. Hlavními nedostatky byly nedostatečné odlišení textu a odkazů a místy špatně čitelný text (nevhodně zvolené styly písma). Nedostatky byly zaznamenány i v případě ovládání 3D scény. Zvolený styl tlačítek pro otáčení a posouvání kamery se ukázal být neintuitivní. Menší výtka byla adresována na možnosti pohybu kamery a chybějící možnost „převrácení“.

Volba technologií

Prototyp webové aplikace byl napsán v *TypeScript*u za použití knihovny *React*. Pro vizualizaci 3D modelů byla vybrána technologie *Babylon.js*. Výběr byl proveden na základě několika kritérií (podpora webové platformy, kvalita dokumentace, možnosti skriptování, podpůrné nástroje). V porovnání s ostatními technologiemi byl *Babylon.js* ve většině kritérií ohodnocen jakožto průměrný. Zdaleka nejlepší technologií se ukázal být v kvalitě dokumentace.

Volba *Babylon.js* měla jak pozitivní, tak i negativní dopady. Pozitivním dopadem je relativně jednoduchý vývoj 3D scény. Naprostá většina kódu přípravy scény byla odvozena z dokumentace *Babylon.js*. Negativním dopadem je podpora formátů pro 3D modely. Oficiálně jsou podporovány pouze formáty *.gltf/.glb*, *.obj* a *.stl*. Tento omezený výběr se ukázal být nedostatečným po vložení reálných modelů budov do *private API*.

2.3 React

React je open source *JavaScript*ová knihovna pro tvorbu uživatelského rozhraní a je distribuována pod licencí MIT. *React* je založený na konceptu komponentů (*React component*). [4]

2.3.1 React Components

Komponenty fungují tak, že přijímají *properties* a vrací odpovídající *React Element*. Všechny komponenty se musejí chovat ve vztahu k *properties* jako „*pure function*“. Stejné vstupy implikují stejné výstupy a funkce nemá vedlejší efekty (např. modifikace nějaké ne-lokální proměnné). [5]

Uživatelská rozhraní aplikací jsou samozřejmě dynamická a v čase se mění. *State* umožňuje *React component*ám měnit svůj výstup v průběhu času v reakci na akce uživatele, *network responses* a cokoli jiného, aniž by došlo k porušení pravidla *pure function*. [5] *State* může být připodobněn k lokální proměnné s tím rozdílem, že není vázán na konkrétní volání funkce, instanci třídy nebo místo v kódu, ale je lokální ke konkrétní *React component*ě. [6] Po změně stavu je automaticky komponenta aktualizována.

2.3.1.1 Function a Class Component

Vnitřní podoba *React component*y je závislá na volbě mezi *function component* a *class component*. Vnější podoba je vždy stejná, takže je možné používat/skládat komponenty stejným způsobem, nezávisle na typu komponent.

Class Component
Dle mého názoru snáze uchopitelná varianta je *class component*. Podoba komponenty odpovídá ES6 třídám. ES6 třídy jsou šablonou pro vytváření objektů a jsou postaveny na prototypech, ale mají také syntaxi a sémantiku, která je typická pro třídy.[7]

Constructor slouží k předávání *properties*. Hlavní metodou je ale *render*, která je volána při sestavení komponenty a při každém *update*. Důležitými metodami jsou také *componentDidMount* a *componentWillUnmount*, které jsou volány v odpovídající fázi životního cyklu komponenty. *State* je reprezentován jako instanční proměnná (*this.state*), která může být změněna zavoláním

this.setState. Pro zachycení změny, ať už *state* nebo *properties*, je používána metoda *componentDidUpdate*, která je volána s parametry: předchozí *state* a předchozí *properties*. [8]

```
import React, { Component } from "react";

class ClassComponent extends React.Component{
    constructor(){
        super();
        this.state={
            count :0
        };
        this.increase=this.increase.bind(this);
    }

    increase(){
        this.setState({count : this.state.count +1});
    }

    render(){
        return (
            <div style={{margin:'50px'}}>
                <h1>Welcome to Geeks for Geeks </h1>
                <h3>Counter App using Class Component : </h3>
                <h2> {this.state.count}</h2>
                <button onClick={this.increase}> Add</button>
            </div>
        )
    }
}
```

Tabulka 2.1: Ukázka class componenty[9]

Function Component

Function component nemá tak jasné rozdělení funkcionalit, jako *class component*. Jak už z názvu vyplývá, tak celá komponenta je reprezentována jednou funkcí.

V nejjednodušší formě je tělo funkce kombinací *constructor*u a *render* metody. Pokud je ale nutné pracovat s životním cyklem komponenty, stavem nebo je nutné sledovat konkrétní změny *properties*, tak řešením je použití *hooks*. *Hooks* jsou funkce, které umožňují „připojit se“ (hook into) k stavu (*state hook*) a k životnímu cyklu (*effect hook*) funkčních komponent. [10] Obdobně, jako u *class componenty*, je při použití *state hook*u dostupná proměnná, která reprezentuje *state* a funkce pro jeho změnu. *Effect hook* přidává možnost vytvářet vedlejší efekty z *function componenty*. Slouží ke stejnému účelu jako componentDidMount, componentDidUpdate a componentWillUnmount v *class componentě*, ale je sjednocen do jednotného rozhraní API. [10]

```
import React, { useState } from "react";

const FunctionalComponent=()=>{
    const [count, setCount] = useState(0);

    const increase = () => {
        setCount(count+1);
    }

    return (
        <div style={{margin:'50px'}}>
            <h1>Welcome to Geeks for Geeks </h1>
            <h3>Counter App using Functional Component : </h3>
            <h2>{count}</h2>
            <button onClick={increase}>Add</button>
        </div>
    )
}
```

Tabulka 2.2: Ukázka function componenty[9]

2.3.2 ReactDOM

React využívá programovací koncept *VDOM*, který udržuje reprezentaci UI v paměti a provádí synchronizaci s „reálným“ *DOM* pomocí knihovny jako je *ReactDOM*. Tomuto procesu se říká *reconciliation*. [11]

Nejmenším stavebním prvkem *React* je *element*. Oproti *DOM Element* je *React Element* pouze jednoduchý objekt, takže je nenáročné ho vytvořit. *React Element* je *immutable* a jakmile je element vytvořen, nelze měnit jeho potomka nebo jeho atributy. To znamená, že pokud je potřeba změnit nějakou část *React Element*, tak je nutné vytvořit nový *React Element*. *ReactDOM* pak porovná tyto dva elementy a provede update pouze na změněné části. [12]

2.4 React Cosmos

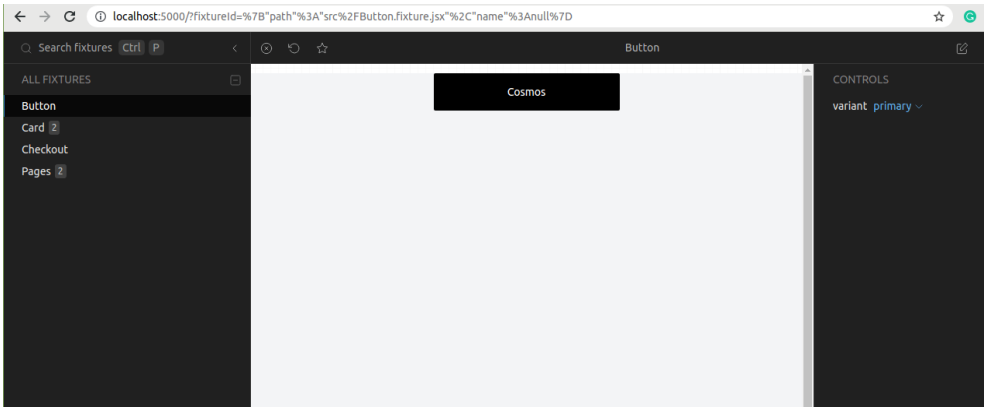
React Cosmos je knihovna pro *React*, která poskytuje izolované prostředí pro znovuvyužití, testování a vývoj komponent uživatelského rozhraní.[13] Jeden z hlavních cílů autora tohoto nástroje je podpora *visual TDD*.

Tato knihovna funguje nad uživatelem definovanými komponenty (*fixtures*), které mohou být použity k simulaci konkrétního *use case*. U každé *fixture* lze interaktivně měnit *properties* a díky *hot-reload*[1] lze sledovat její změny během vývoje.

```
export default (
    <div>
        <div className="w-50 mx-auto">
                <Button>Cosmos</Button>
        </div>
    </div>
);
```

Tabulka 2.3: Ukázka fixture[13]

[1] promítnutí změn ve zdrojovém kódu bez nutnosti restartu



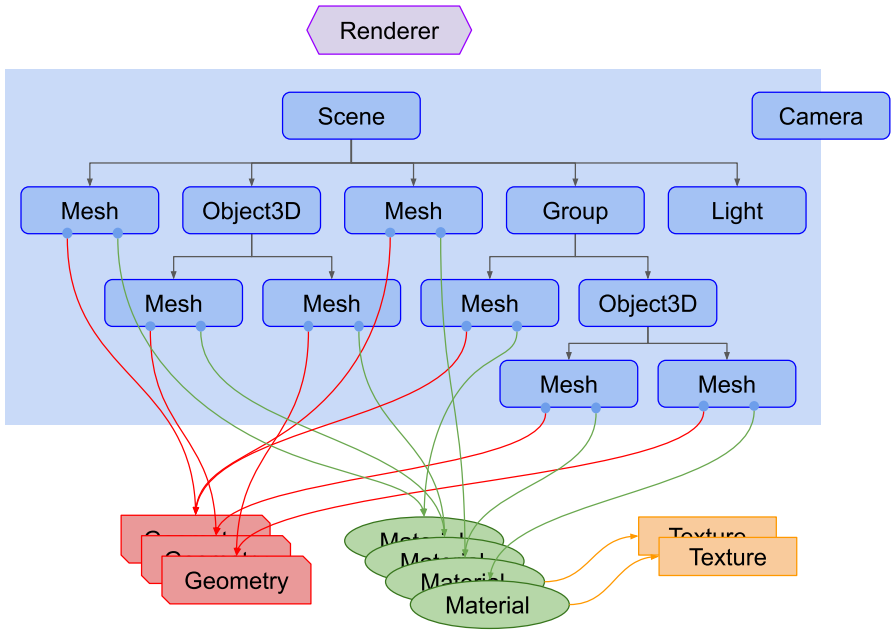
Obrázek 2.3: Vzhled React Cosmos[13]

2.5 Three.js

Three.js je 3D knihovna, která se pokouší co nejvíce usnadnit zobrazení 3D obsahu na webové stránce. [14]

Three.js je často zaměňován s *WebGL*, protože většinou, ale ne vždy, používá k vykreslování 3D *WebGL*. *WebGL* je low-level systém, který vykresluje pouze body, čáry a trojúhelníky. K tomu, aby se s *WebGL* dalo dělat cokoli užitečného, je obvykle zapotřebí poměrně hodně kódu, a proto přichází na řadu *Three.js*. Zajišťuje věci jako scény, světla, stíny, materiály, textury ...[14]

2.5.1 Základní objekty



Obrázek 2.4: Ukázka struktury aplikace v Three.js [14]

2.5.1.1 Renderer

Renderer vykresluje část 3D scény, která se nachází uvnitř zorného pole kamery, jako 2D snímek na *HTML canvas*. [14] Ve výchozím nastavení používá *WebGL*. *WebGL* umožňuje GPU akcelerované zpracování obrazu a efektů. [15]

2.5.1.2 Scene

Scenegraph (modrý obdélník na obrázku 2.4) je stromová struktura, která se skládá z různých objektů, jako je *Scene* objekt, několik *Mesh* objektů, *Light* objektů, *Group*, *Object3D* a *Camera* objektů. Umístění v této struktuře určuje, kde se objekty zobrazují a jak jsou orientovány (děti jsou umístěny a orientovány relativně ke svému rodiči). Kořenem tohoto stromu je *Scene*, která obsahuje vlastnosti jako je barva pozadí a mlha. [14] Scény umožňují nastavit, co se má pomocí *Three.js* vykreslovat a kde se to má nacházet v 3D souřadnicích. [15]

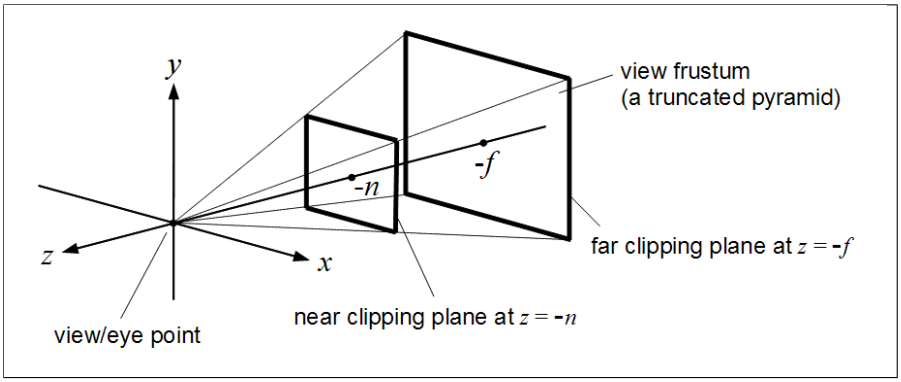
2.5.1.3 Object3D a Group

Object3D je základní třída pro řadu objektů v *Three.js* a poskytuje metody a vlastnosti pro manipulaci s objekty ve 3D prostoru. Nejběžnějšími příklady jsou *Meshes*, *Lights*, *Cameras* a *Groups*. [15]

Na *Group* je možné nahlížet jako na univerzálního rodiče v *scenegraph*u. *Group* je sama o sobě *Object3D* a tudíž s ní lze manipulovat v 3D prostoru.

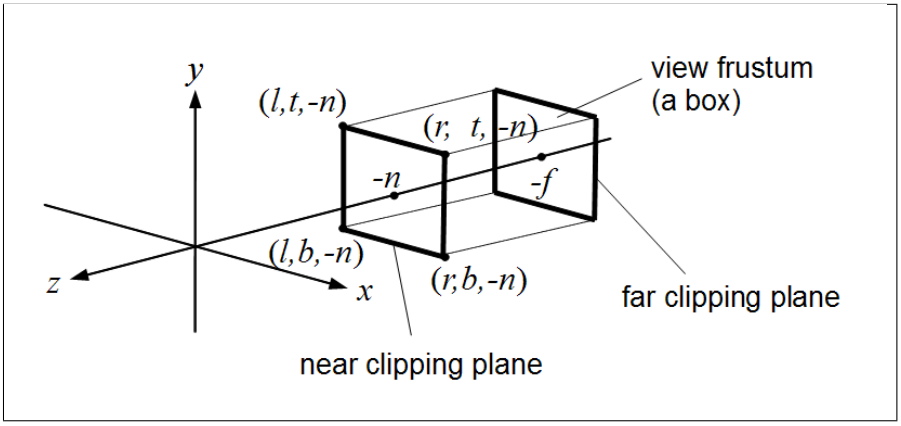
2.5.1.4 Camera

V systému *Three.js* existuje mnoho typů kamer. Například perspektivní nebo ortografická kamera. Perspektivní projekce je navržena tak, aby napodobovala způsob, jakým vidí lidské oko. Je to velmi častý režim projekce, používaný při vykreslování 3D scén. Ortografická projekce je jako kostka sama o sobě, kde perspektiva zůstává konstantní bez ohledu na vzdálenost od kamery. Vlastnosti kamery popisují Frustum[2], což jsou rozměry uvnitř scény, která bude vykreslena. [15]



Obrázek 2.5: Perspektivní projekce [15]

[2]Komolý jehlan – část jehlanu, která leží mezi dvěma rovnoběžnými rovinami procházející tímto jehlanem.[16]



Obrázek 2.6: Ortografická projekce [15]

Na obrázku 2.4 je *Camera* napůl uvnitř a napůl mimo *scenegraph*. Tím je znázorněno, že v *Three.js* nemusí být *Camera*, na rozdíl od ostatních objektů, ve *scenegraph*u, aby fungovala. Stejně jako ostatní objekty se bude *Camera* jako potomek jiného objektu pohybovat a orientovat vzhledem ke svému rodičovskému objektu. [14]

2.5.1.5 Mesh, Geometry a Material

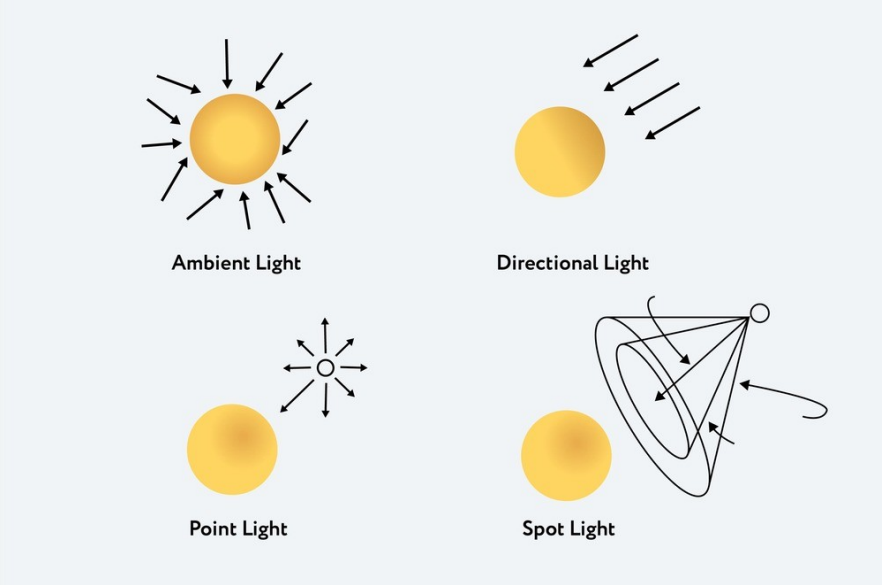
*Mesh*e reprezentují 3D objekty k vykreslení. Konkrétní tvar a texturu/textury jsou dodány pomocí *Geometry* a *Material*. Jak je vidět na obrázku 2.4, *Geometry* i *Material* může být použit pro vícero *Mesh*ů.

Geometry reprezentuje vertexová data libovolného geometrického útvaru. Kromě načítání geometrie ze souborů, poskytuje *Three.js* mnoho druhů zabudovaných geometrických útvarů. [14]

Material reprezentuje vlastnosti povrchu, které se používají při vykreslování *Mesh*e, včetně takových věcí jako je barva, deformace, lom světla, lesk ...*Material* může také odkazovat na jeden nebo více objektů *Texture*, které mohou být opět přepoužity.

2.5.1.6 Light

Three.js obsahuje několik různých variant osvětlení. Za zmínku, dle mého názoru, stojí *AmbientLight*, *HemisphereLight*, *DirectionalLight* a *SpotLight*. U každého *Light* je možné nastavit barvu světla a jeho intenzitu.



Obrázek 2.7: Ukázka světel v Three.js [17]

AmbientLight
Tím nejvíce přímočarým způsobem nasvícení je *AmbientLight*. Toto světlo nasvítí každý *Mesh* ve scéně rovnoměrně. U žádného *Mesh*e nezáleží na jeho umístění nebo natočení. Ať už je v zákrytu jiného *Mesh*e nebo ne, bude rovnoměrně osvětlen.

HemisphereLight
Často používaná náhrada pro *AmbientLight* je *HemisphereLight*. Podobně jako u *AmbientLight* osvětluje každý *Mesh* nezávisle na tom, kde je umístěný v prostoru. Rozdílem je, že každý *Mesh* osvětluje jen zespoda a seshora a může k tomu použít různé barvy. Těmto barvám se říká *ground color* a *sky color*.

DirectionalLight
Prvním světlem, které může vrhat stíny je *DirectionalLight*. Jak název napovídá, tak se jedná o světlo, které nasvicuje celou scénu z jednoho úhlu (směru). Nastavení konkrétního směru lze provést dvěma způsoby. Buď přímo natočit světlo pomocí atributu *rotation* nebo umístit toto světlo do konkrétního místa v prostoru a nastavit atribut *target* (specifický pro *DirectionalLight*).

SpotLight
SpotLight je velice podobné a v některých případech i zaměnitelné za *DirectionalLight*. Podobnost spočívá v tom, že opět dochází k nasvícení jen z jednoho konkrétního směru. Rozdílem je, že u *SpotLight* dochází k emisi světla

přímo z bodu v prostoru, kde je *SpotLight* umístěno. Důsledkem je, že v prostoru před světlem vznikne „kužel světla“, se kterým je možné manipulovat prostřednictvím objektu *SpotLight*. U tohoto kužele je možné nastavit výšku (*range*), úhel (*angle*) nebo i postupné slábnutí světla.

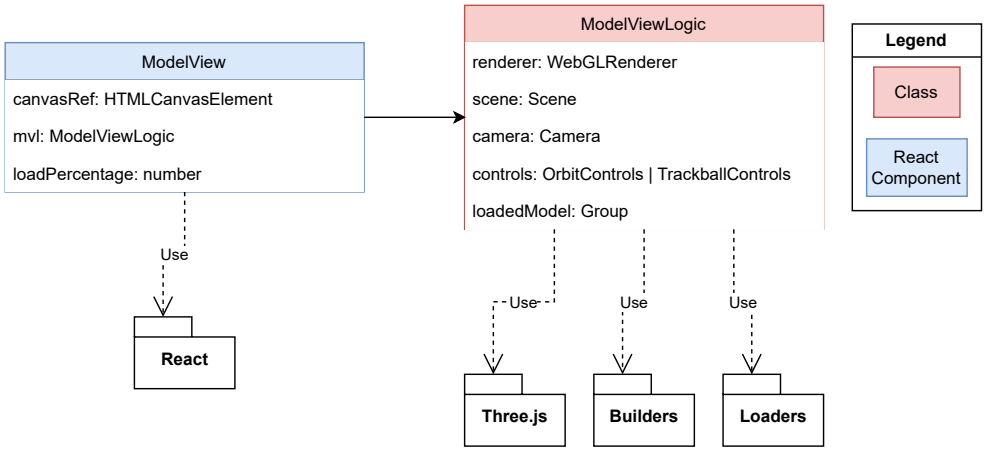
Kapitola 3

Návrh

Součástí této kapitoly je návrh webových komponent a *TypeScript* tříd. Dále je proveden návrh na podobu a fungování zobrazení dvou modelů do jedné scény a zobrazení dvou modelů do synchronizovaných scén.

3.1 React a Three.js

Základ vztahů *React*u a *Three.js* třímá v maximální izolaci. Účelem je jasné oddělení co je ještě manipulace s HTML a co už je manipulace s 3D scénou. Mezi původní důvody patřila i možnost výměny knihoven *React* a *Three.js* za jiné. U knihovny *Three.js* to je nereálné, protože se nelze vyhnout specifickým postupům při konstrukci scény, načítání modelů, manipulace s kamerou atp. U *React*u to reálné je, protože každý webový framework pracuje v nějaké míře s referencemi na HTML elementy a s vnitřním stavem.



Obrázek 3.1: Oddělení Three.js od React

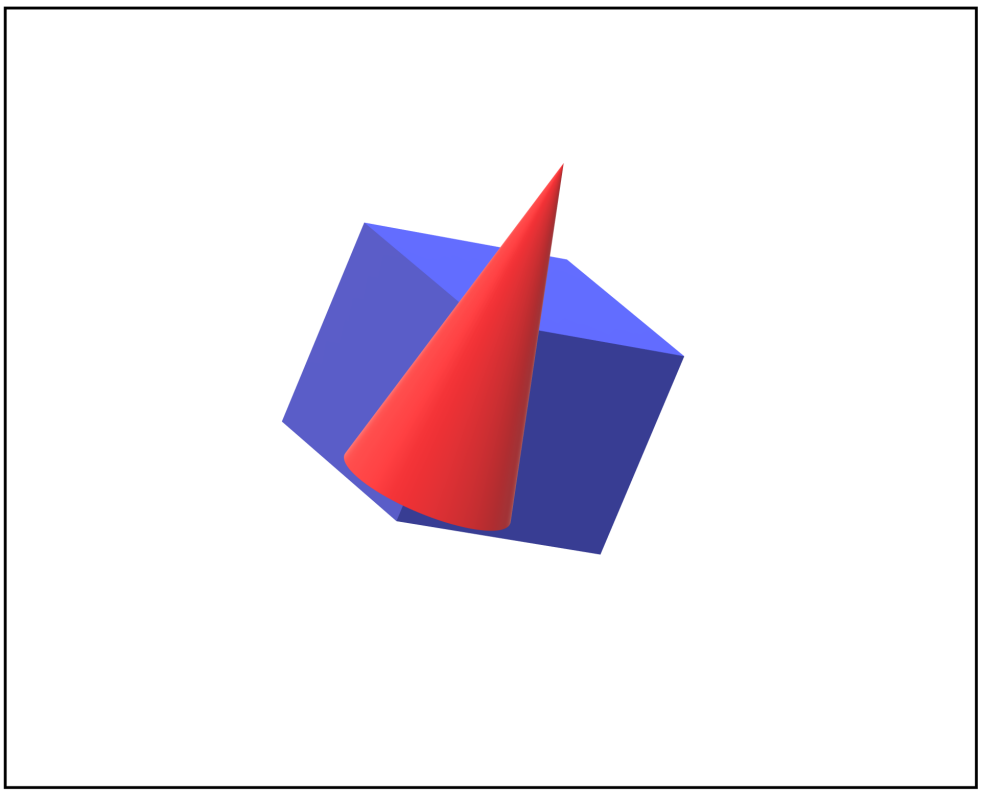
Vztahy mezi těmito dvěma knihovnami lze nejlépe demonstrovat na základní komponentě pro zobrazení 3D modelu (obrázek 3.1). Třída s logikou (*ModelViewLogic*) spravuje 3D scénu a má na starosti i pohyb kamery a načítání 3D modelu. *React* komponenta spravuje HTML část a předává *properties* třídě s logikou. Nejdůležitějším elementem HTML části je *canvas* pro vykreslení 3D scény.

Např. proces vytvoření 3D scény s načtením modelu je rozdělitelný do několika fází. První je vytvořen *React component* (*ModelView*), jehož úkolem je vytvořit *HTML canvas*. Jakmile je vytvořen *canvas*, tak komponenta vytvoří třídu s logikou (*ModelViewLogic*) a předá jí referenci na *canvas*. *ModelViewLogic* připraví celou 3D scénu (pozadí, podlaha, světla, ...) a začne vykreslovat do přijatého *canvasu*. Jakmile je scéna připravena, tak *ModelView* notifikuje *ModelViewLogic*, že je třeba načíst model. Samotné načítání už má na starosti odpovídající *loader*.

3.2 Porovnání dvou 3D modelů

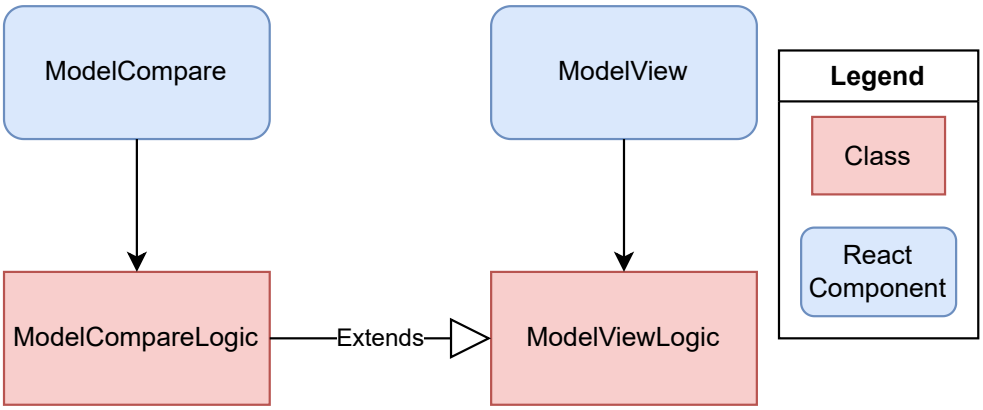
Pro vizuální porovnání dvou 3D modelů existují 2 varianty zobrazení. První variantou je zobrazení do jedné 3D scény, tudíž projekce do jednoho okna. Druhou variantou je vykreslení 3D modelů do vlastních 3D scén, tudíž projekce do dvou oken. Obě varianty jsou žádoucí.

3.2.1 Modely v jedné 3D scéně



Obrázek 3.2: Ukázka porovnání dvou modelů v jedné scéně

V tomto případě jsou dva modely, pro porovnání mezi sebou, načteny do jedné 3D scény, jsou umístěny do stejného bodu (0, 0, 0) se stejnou rotací (0, 0, 0). Pokud bude taková scéna bez dalších úprav vykreslena, bude porovnání těchto modelů prakticky nemožné. Z umístění a rotace modelů vyplývá, že s vysokou pravděpodobností dojde k prolínání modelů a v horším případě dojde i k problikávání textur. Řešením těchto problémů je odlišení aktivního a ne-aktivního modelu na úrovni vykreslování jeho *Mesh*e.



Obrázek 3.3: Odvození CompareView

Z technického hlediska je nutné, oproti základní komponentě pro zobrazení 3D modelů, změnit jak *React* komponentu, tak i třídu s logikou.

Třída s logikou je vytvořena rozšířením *ModelViewLogic*. Tato nová třída (*ModelCompareLogic*) reflektuje, že je nutné pracovat s dvěma modely, tudíž i s dvěma operacemi načítání. Jak už bylo zmíněno výše, musí se vypořádat s aktivními a ne-aktivními modely a jejich volbou. Zbytek chování může být převzatý z *ModelViewLogic*.

Z důvodu používání *function React component* není možné použít stejný princip rozšíření jako u třídy s logikou. Opět velká část kódu této komponenty odpovídá *ModelView*, až na předávání URL modelu. V tomto případě musí *React* komponenta předávat dvě URL a jako bonus musí předávat informaci o zvoleném aktivním modelu a o zvoleném odlišení aktivního a ne-aktivního modelu.

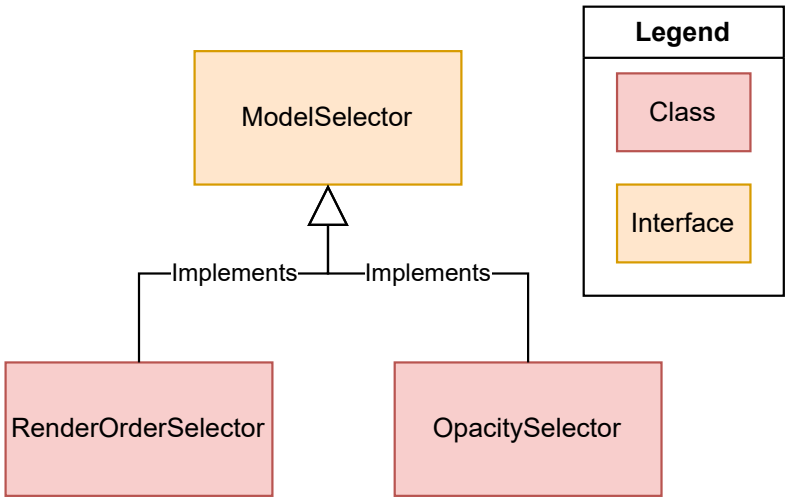
3.2.1.1 Odlišení aktivního a ne-aktivního modelu

Způsobů odlišení aktivního a ne-aktivního modelu je hned několik. Např. úprava *renderOrder*u, nastavení průhlednosti, použití *disabled* u *Object3D*...

Technika úpravy *renderOrder*u spočívá v tom, že *renderer*u je specifikováno, které objekty se mají vykreslit přednostně. Výsledný efekt je iluze, že některé objekty jsou v popředí, i když jsou v prostoru až za objekty, které nemají vyšší prioritu u *renderer*u (obrázek 3.2).

Nastavení průhlednosti je další relevantní způsob odlišení. Základem tohoto postupu je zvyšování průhlednosti ne-aktivních modelů. Zvyšování a zmenšování průhlednosti může být nebezpečné u již částečně průhledných objektů (např. okna), proto je nutné, si buď pamatovat původní průhlednost, nebo měnit průhlednost relativně.

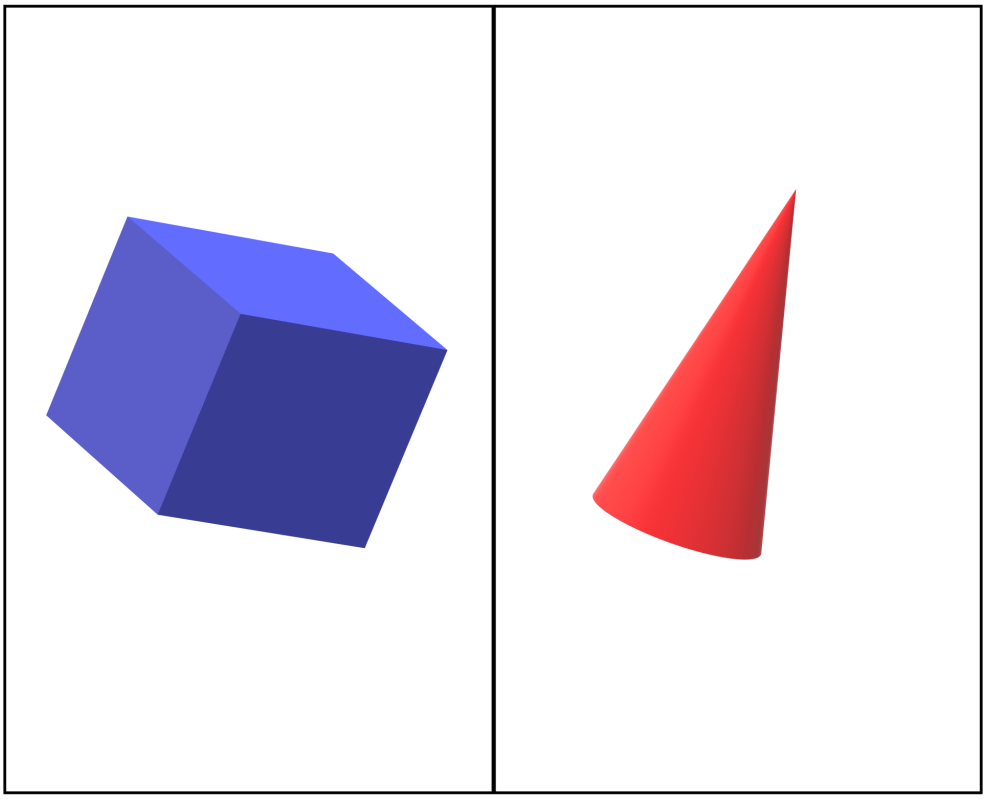
Použití *disabled* na *Object3D* v kontextu této práce nedává smysl. Základní komponenta pro zobrazení 3D modelu (*ModelView*) má podobné chování při změně vstupní URL. Rozdílem je, že *ModelView* starý model úplně smaže ze scény, to ale není problém, protože modely jsou cacheované.



Obrázek 3.4: Hiearchie Selectorů

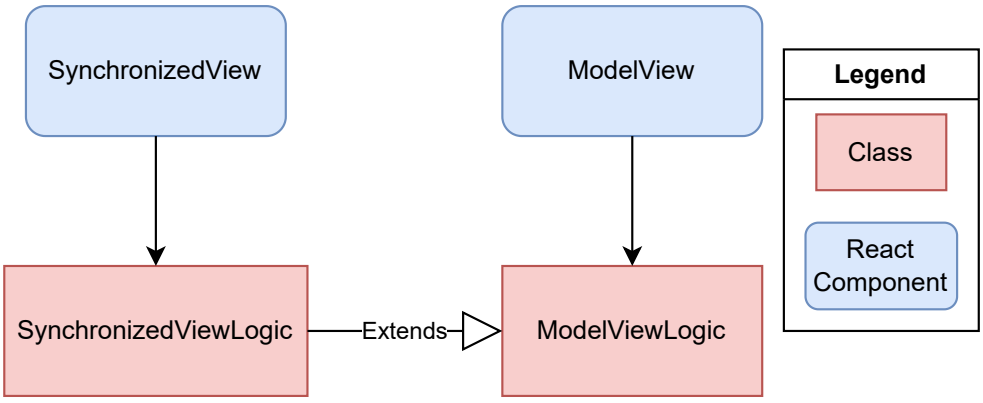
Tuto funkcionalitu je vhodné oddělit od třídy s logikou (*ModelCompareLogic*) a vytvořit samostatné třídy pro správu aktivních a ne-aktivních modelů. Konkrétně se jedná o třídy *RenderOrderSelector* (pro odlišení pomocí *renderOrder*) a *OpacitySelector* (pro odlišení pomocí průhlednosti). Obě tyto třídy implementují jednotné rozhraní *ModelSelector* tak, aby bylo jednoduché mezi nimi přepínat.

3.2.2 Modely ve vlastních 3D scénách



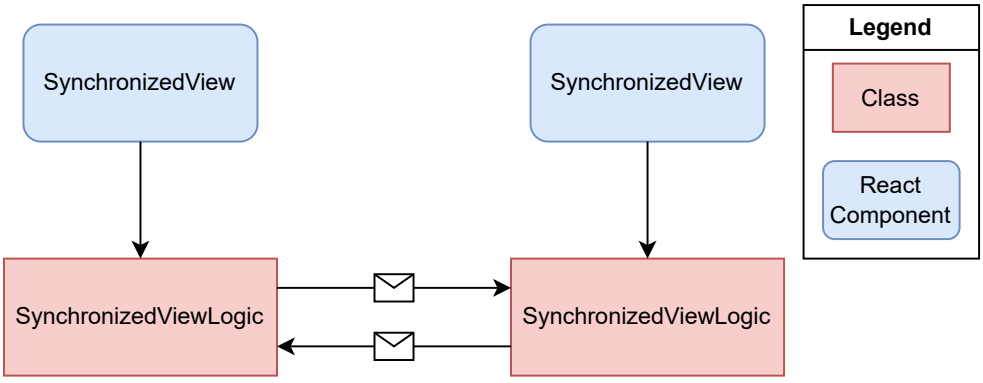
Obrázek 3.5: Ukázka porovnání dvou modelů v separátních scénách

V té nejtriviálnější formě je porovnání dvou modelů ve vlastních 3D scénách ekvivalentní použití dvou základních komponent *ModelView*. Za použití dvou *ModelView* lze jen těžko zkoumat odlišnosti v rozměrech a detailech. Pro minimalizaci těchto nedostatků je třeba zaručit, že pozice a rotace kamer v obou scénách jsou totožné a že existuje nějaká vizuální nápověda, kam ukazuje kurzor v obou scénách. Tudíž je nutná synchronizace předem specifikovaných parametrů mezi dvěma 3D scénami.



Obrázek 3.6: Odvození SynchronizedView

Logicky se jedná o rozšiřování základní komponenty pro zobrazení 3D modelu o funkcionalitu synchronizace, a je tedy vhodné reflektovat tento fakt ve zdrojovém kódu. Z důvodu používání *function React component* není možné rozšířit již existující *React* komponentu *ModelView*. Je ale možné rozšířit třídu s logikou *ModelViewLogic*.

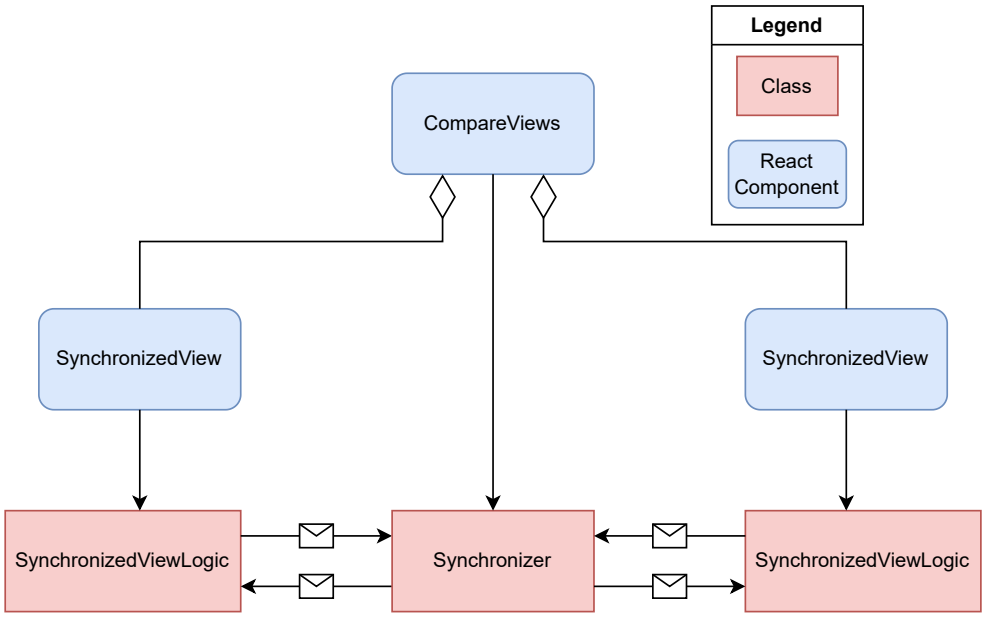


Obrázek 3.7: Nutnost komunikace pro synchronizaci

Aby bylo možné provést synchronizaci mezi dvěma scénami, je nutné nějakým způsobem umožnit instanci *SynchronizedViewLogic* komunikovat se svým protějškem a naopak. To znamená, že pro obousměrnou komunikaci si musí být první instance *SynchronizedViewLogic* vědoma existence druhé instance a naopak. Tuto problematiku ještě komplikuje fakt, že *SynchronizedViewLogic* je uložený jako *state* v *React* komponentě *SynchronizedView*. *SynchronizedView* se mohou dynamicky měnit tak, že jednou platný protějšek *SynchronizedViewLogic* může být nahrazený jinou instancí. Tento problém je řešitelný systémem *callback*ů,

který by reflektoval změny protějšků a který by byl spravován rodičovskou *React* komponentou.

3.2.2.1 Mediátor



Obrázek 3.8: Komunikace přes Synchronizer

Objektivně lepším řešením je použít pro komunikaci třídu (*Synchronizer*), která bude navržena podle návrhového vzoru. Vhodným návrhovým vzorem je mediátor, který je specificky určen pro zjednodušení komunikace mezi vícero objekty. V tomto konkrétním případě odbourá nutnost znalosti protějšku *SynchronizedViewLogic*.

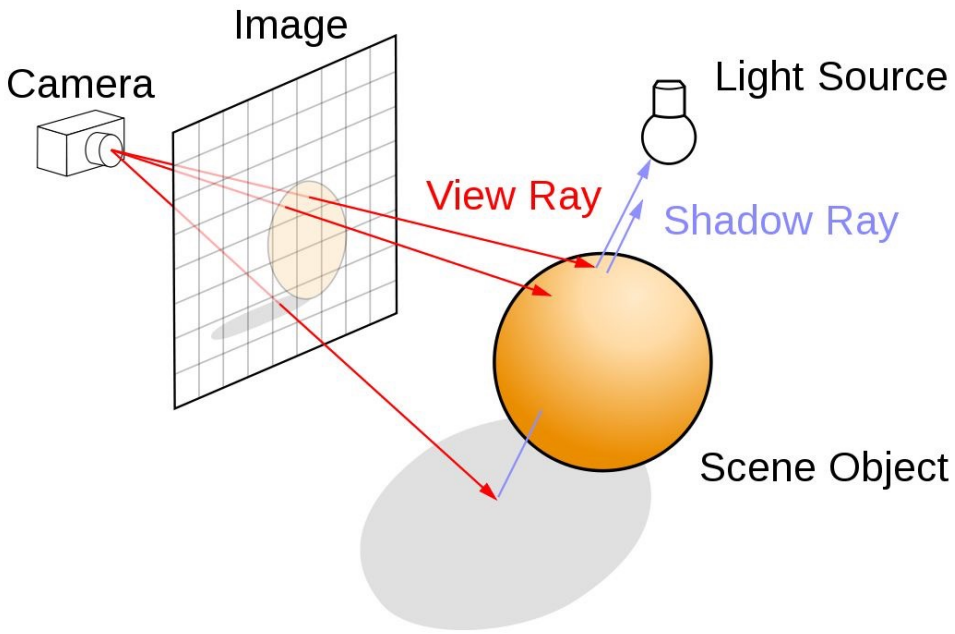
Použití mediátora řeší problémy, které vyplývají i z toho, že je použita *React* komponenta. Pokud je *Synchronizer* vytvořen v rámci rodičovské komponenty (*SynchronizedView*), tak může být poslán jako parametr do komponent potomků. V potomcích pak stačí pouze sledovat změny *Synchronizer*u a v závislosti na tom, dát podnět své třídě s logikou, aby se přihlásila nebo odhlásila k odběru zpráv. Tento princip řeší i problém se změnami potomků a jejich tříd s logikou, protože každá třída s logikou se přihlašuje a odhlašuje sama za sebe a komunikační protějšek v tu chvíli nemusí ani existovat.

Vedlejším efektem tohoto návrhu je možnost přidání libovolného počtu modelů k porovnání. *Synchronizer* nemá vnitřní limit pro počet komunikujících instancí *SynchronizedViewLogic*. Druhým vedlejším efektem je možnost lo-

gování, kde má uživatel kameru nebo kurzor, protože *Synchronizer* nemusí být omezen jen na instance *SynchronizedViewLogic*.

3.3 Kurzory

Hlavním účelem kurzorů je vizuální nápověda, kam, případně na co, uživatel ukazuje. V kontextu této práce je největší potenciál v zobrazování kurzorů u porovnávání modelů v různých 3D scénách. Jiné využití může být označování oblastí modelu tak, jako to bylo provedeno v bakalářské práci *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*[1].



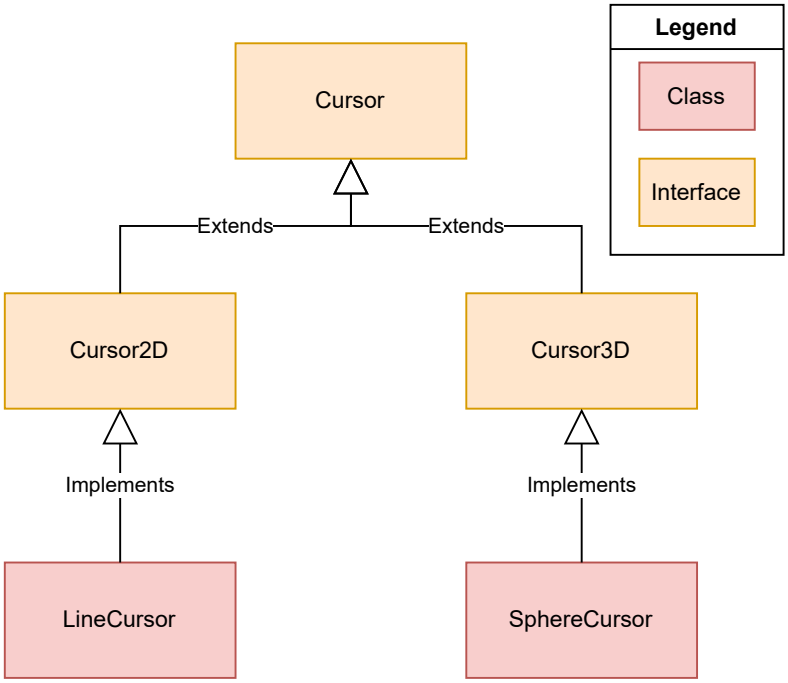
Obrázek 3.9: Ukázka ray casting na příkladu ray tracing[18]

Pro tuto práci byly vybrány kurzory promítané na 3D model, ne na úrovni UI. Vytvoření těchto kurzorů spoléhá na techniku *ray casting*. Principem *ray casting*u je sledování paprsků z oka/kamery po jednom pixelu a nalezení nejbližšího objektu, který blokuje dráhu tohoto paprsku.[19] Paprsek je vytvořen na základě pozice kurzoru myši (v případě PC). Pokud vytvořený paprsek koliduje s načteným modelem, tak v místě kolize je vytvořen kurzor.

3.3.1 Synchronizace kurzorů

Pro synchronizaci kurzorů je vhodné zavést dvě různé metody. První je synchronizace na úrovni obrazu nebo-li synchronizace pomocí 2D souřadnic. Dru-

hou metodou je synchronizace na úrovni 3D prostoru nebo-li synchronizace pomocí 3D souřadnic. Podle zvolené techniky jsou pak kurzory rozdělené na *Cursor2D* a *Cursor3D*.



Obrázek 3.10: Hiearchie kurzorů

Cursor2D využívá synchronizaci na úrovni obrazu. Pomocí *Synchronizer*u je přenášena informace o relativní pozici kurzoru myši ve vztahu k *HTML canvas*u. Pro scénu, která se má podvolit synchronizaci, to znamená, že místo pozice kurzoru myši jsou pro vytvoření paprsku použity přijaté souřadnice. Totožně jako u vytváření kurzoru je kurzor při synchronizaci umístěn do místa kolize s načteným objektem.

Cursor3D využívá synchronizaci na úrovni 3D prostoru. Pomocí *Synchronizer*u je přenášena informace o souřadnicích kurzoru uvnitř 3D scény. Pro scénu, která se má podvolit synchronizaci, to znamená, že pozice kurzoru je nastavena na hodnoty přijatých souřadnic. Při použití této metody u subjektu synchronizace není použit *ray casting*.

Z uživatelského hlediska je za použití *Cursor2D* zobrazen kurzor v obou scénách, jen pokud v obou scénách dojde ke kolizi paprsků s načtenými modely. Při použití *Cursor3D* je zobrazen kurzor v obou scénách, pokud ve scéně, na které je umístěn kurzor myši, dojde ke kolizi paprsku s načteným modelem.

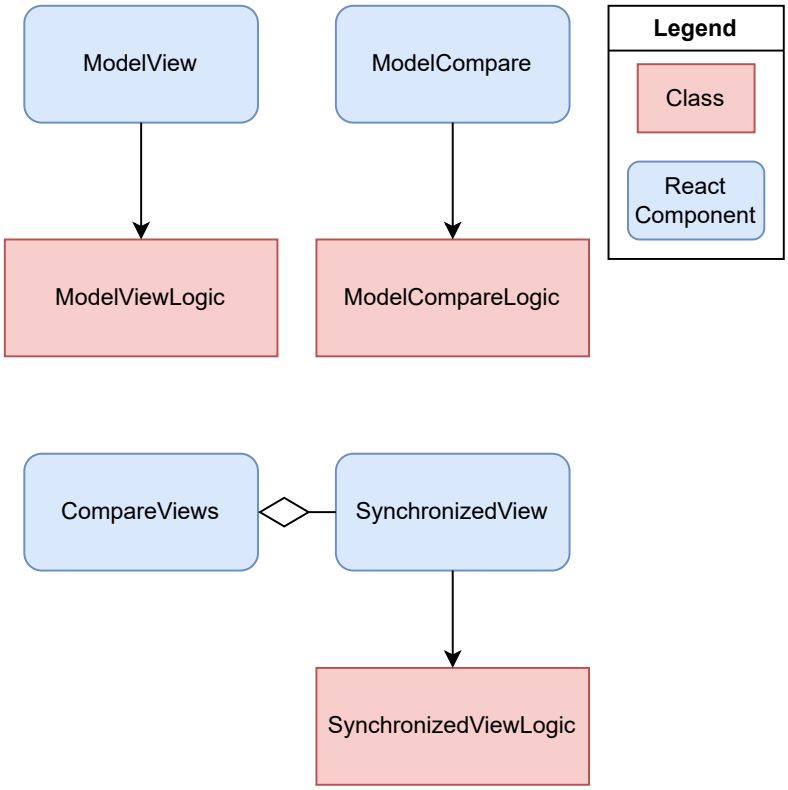
Kapitola 4

Implementace

Kapitola Implementace obsahuje popis *React* komponent, tříd s logikou a jejich vztahy. Součástí této kapitoly je i podkapitola o způsobu načítání 3D modelů. Pro porovnávání modelu jsou zde popsány implementace správy aktivních a ne-aktivních modelů a synchronizace 3D scén. Kapitolu uzavírají návody pro instalaci, integraci a rozšiřování knihovny.

4.1 React kompenenty a třídy s logikou

V rámci návrhu, byl vytyčen cíl co nejvíce izolovat manipulaci s *HTML* a manipulaci s 3D scénou. Pro dosažení tohoto cíle bylo navrženo rozdělení na *React* komponenty a třídy s logikou.



Obrázek 4.1: React kompenenty a třídy s logikou

4.1.1 React kompenenty

Volba mezi *Function Component* a *Class Component* byla provedena na základě osobních preferencích autora. Co se týče základního způsobu použití, jsou obě podoby komponentů, na úrovní funkcionalit, ekvivalentní. Pro tuto práci byla zvolena podoba *Function Component*. Hlavním důvodem této volby je možnost reagovat na změnu parametru(ů) individuálně (*effect hook*).

React komponenty lze rozdělit do dvou kategorií. První kategorií jsou komponenty, které ne-interagují s třídou s logikou. Druhou kategorií jsou naopak komponenty, které přímo interagují s třídou s logikou.

4.1.1.1 Ne-interagující s třídou s logikou

```
const CompareViews = (props: Props)=>{

    ...

    const[synchronizer] = useState<Synchronizer>(new SynchronizerImpl)

    return<div>
        {props.urls.map((url, index)=>
            <SynchronizedView key={index}
                              style={props.styles?.at(index)}
                              cursorOption={props.cursorOption}
                              cameraOption={props.cameraOption}
                              model={model}
                              requestHeaders={props.requestHeaders}
                              synchronizer={synchronizer}
            />
        )}
    </div>
}
```

Tabulka 4.1: Komponenta CompareViews

Jediným reprezentantem první kategorie komponentů je *CompareViews*. Jedná se o jednoduchou komponentu, jejímž účelem je vytvoření dvou *SynchronizedView*. Existence této komponenty je vynucena potřebou synchronizace „potomků". Nezbytnou částí této komponenty je tedy vytvoření *Synchronizer*u a jeho vložení do zprostředkovaných *properties*.

Zde je vhodné se pozastavit nad použitím *state hook*u (*useState*). Na první pohled může jeho použití působit nadbytečně, ale jeho použití je v tomto kontextu při nejmenším vhodné. Na základě znalostí nabytých z analytické části, lze tvrdit, že funkce reprezentující *React component* je volána opakovaně, při každé změně *properties*. Pokud by *synchronizer* byla jen obyčejná proměnná, tak při každém volání *CompareViews* by byl sestaven nový *Synchronizer*. Pro *potomky* by to znamenalo, že by se museli odhlásit od starého *Synchronizer*u a přihlásit k novému *Synchronizer*, což by mělo dopad na výkon aplikace.

4.1.1.2 Interagující s třídou s logikou

Pro každou komponentu tohoto typu platí, že spravuje *canvas* (*HTML element*) a instruuje třídu s logikou při změnách *properties*. Aby tak mohl činit, je nutné použít *reference*, *state* a *effect hook*.

Reference hook je použit pro uložení reference na *canvas* a následně pro předání do třídy s logikou. *State hook* je využíván hlavně pro uložení třídy s logikou, což řeší problém opakované inicializace. Tento problém byl již popsán v části *Ne-interagující s třídou s logikou* na příkladu s *Synchronizer*em. *Effect hook* slouží k definování chování při změně parametrů.

```
const ModelView = (props: Props) => {

    const canvasRef = useRef<HTMLCanvasElement>(null);

    const [mvl, setMvl] = useState<ModelViewLogic>()
    const [loadPercentage, setLP] = useState<number>(0)

    useEffect(()=>{
        mvl?.setEnvironment( props.environmentParams )
    }, [mvl, props.environmentParams])

    useEffect(()=>{

        ...

        const n_mvl = new ModelViewLogic(canvasRef.current, id)
        n_mvl.init()

        setMvl(n_mvl)
    },[canvasRef.current])

    useEffect(()=>{
        mvl?.setControls(props.controlsOption)
    },[mvl, props.controlsOption])

    useEffect(()=>{
        mvl?.setCamera(cameraOption, props.controlsOption)
    }, [mvl, props.cameraOption])

    useEffect(()=>{

        ...

        mvl.load(props.url, props.requestHeaders, (progress)=>{ ... })

        return ()=>{
            mvl.removeLoaded()
        }
    },[mvl, props.url, props.requestHeaders])

    return (
        <div style={style}>
            <ProgressBar animated={false}
                now={loadPercentage} label={`${loadPercentage}%`}
                style={...} />

            <canvas ref={canvasRef}/>
        </div>
    );
}
```

Tabulka 4.2: Komponenta ModelView

Příkladem komponenty interagující s třídou s logikou je *ModelView*. Jak už bylo zmíněno, jednou z hlavních povinností tohoto druhu komponent je řízení změn *properties*. Není tedy překvapením, že velkou částí kódu jsou volání *effect hook*ů. Konkrétně u této komponenty se jedná o 5 *effect hook*ů, které mají na starost inicializaci třídy s logikou, nastavení prostředí 3D scény, nastavení kamery, nastavení ovládání a načtení 3D modelu. Kromě samotné inicializace třídy s logikou, jsou *effect hook*y z velké části omezeny na volání metody na třídě s logikou.

Za zmínku stojí *effect hook*, který se stará o načtení 3D modelu. V tomto případě se totiž nejedná o standardní *effect hook*, ale o *effect hook with cleanup*. Místo funkce bez návratové hodnoty se zde používá funkce s návratovou

hodnotou *cleanup* funkce. *Cleanup* funkce, jak už vyplývá z názvu, je určena k „uklizení“ po předešlých změnách. Pro *effect hook*, starající se o načítání 3D modelů, to znamená, že pokaždé než dojde k načítání nového 3D modelu, dojde k odstranění starého 3D modelu ze scény.

4.1.2 Třídy s logikou

Tyto třídy spravují 3D scény a provádějí v nich změny na základě zpráv od *React* komponent. Pro zjednodušení kódu, využívají převážně parametrizované *buildery* (*RendererBuilder*, *ControlsBuilder*, *CameraBuilder*, ...). Výjimkou je *SceneBuilder*, který není parametrizovaný, jelikož počáteční podoba 3D scény je jednotná.

Třída *ModelViewLogic* je základní třídou pro práci s 3D scénou. Nejen že obsahuje základní sadu operací pro práci s 3D scénou, ale zároveň je rodičem všech tříd s logikou. Díky dědění jsou schopny *SynchronizedViewLogic* a *ModelCompareLogic* používat stejné operace a proměnné jako *ModelViewLogic*.

```
export default class ModelViewLogic {

    ...

    constructor(canvas: HTMLCanvasElement, id: number) {
        this.canvas = canvas
        this.id = id
    }

    private readonly animate = ()=>{
        requestAnimationFrame(this.animate)

        if(!this.scene || !this.camera)
            return;

        this.controls?.update();
        this.renderer?.render(this.scene, this.camera)
    }

    init(){
        this.renderer = RendererBuilder.build(this.canvas);
        this.scene = SceneBuilder.build();

        this.animate()
    }

    setEnvironment(envParams?: EnvironmentParams){ ... }

    setControls(option: ControlsOption){ ... }

    setCamera(cameraOption: CameraOption, controlsOption: ControlsOption)
    { ... }

    async loadModel(model: Model, requestHeaders: {[p: string]: string},
        onProgress?: (event: ProgressEvent<EventTarget>) => void
    ): Promise<THREE.Group> { ... }

    removeLoaded(){ ... }
}
```

Tabulka 4.3: ModelViewLogic

ModelViewLogic obsahuje metody pro inicializaci scény, *render loop* (*ani-*

mate), nastavení prostředí 3D scény (*setEnvironment*), nastavení ovládání (*setControls*), nastavení kamery (*setCamera*) a načtení/odebrání 3D modelu (*loadModel*/*removeLoaded*). Pro sestavení 3D scény a spuštění *render loopu*, je nutné zavolat konstruktor a následné inicializovat scénu (*init*). Následně se počítá s opakovaným nastavení prostředí 3D scény, nastavení ovládání, nastavení kamery a načtení/odebrání 3D modelu.

Ve stavu hned po inicializaci 3D scény, je scéna v situaci, kdy není nastaveno ovládání ani kamera. To znamená, že *render loop* nevykresluje nic do *canvas*u a běží „na prázdno“. Je nutné mít na paměti, že odpovídající *React* komponenta (*ModelView*) obsahuje *effect hook*y, které si v krátké době po inicializaci scény vyžádají nastavení jak kamery, tak i jejího ovládání. Důsledkem je, že *render loop* doopravdy běží na okamžik „na prázdno“, ale k nastavení kamery a ovládání dojde jen jednou.

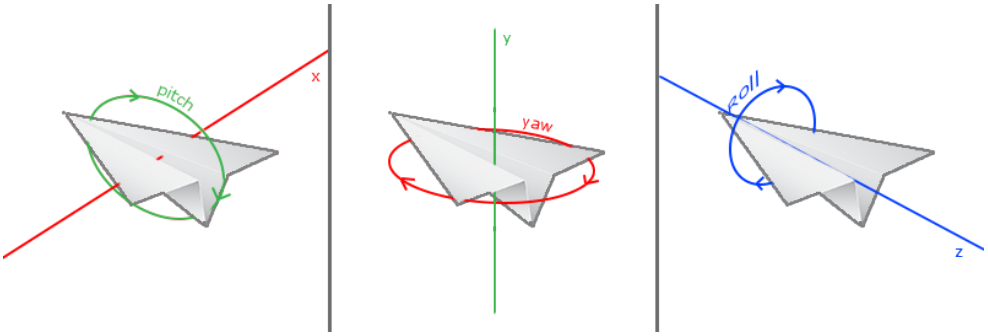
4.1.2.1 Nastavení kamery a ovládání

```
export default class ModelViewLogic {
    protected readonly canvas: HTMLCanvasElement;
    protected renderer?: THREE.WebGLRenderer;
    protected camera?: THREE.Camera;
    protected controls?: OrbitControls | TrackballControls;
    protected resizeObserver?: ResizeObserver;

    ...

    setControls(option: ControlsOption){
        if(!this.camera)
            return

        if(this.controls){
            const oldControls = this.controls
            this.controls = undefined

            oldControls.reset()
            oldControls?.dispose()
        }

        this.controls =
            ControlsBuilder.build(this.canvas, this.camera, option)
    }

    setCamera(cameraOption: CameraOption, controlsOption: ControlsOption){
        this.resizeObserver?.disconnect()

        this.camera=CameraBuilder.build(cameraOption)
        this.setControls(controlsOption)

        this.resizeObserver =
            getResizeObserver(this.canvas, this.camera, this.renderer);

        this.resizeObserver.observe(getParentElement(this.canvas))
    }
}
```

Tabulka 4.4: Nastavení kamery a ovládání

Nastavení ovládání

Původní podoba nastavení ovládání spočívala v obyčejném přepsání třídní proměnné. Objekt *controls* sice není součástí 3D scény, ale je těsně navázán na *canvas* pomoci *event listener*ů. Pokud nedojde k odstranění těchto *listener*ů (*dispose*), tak po změně ovládání dojde ke konfliktu mezi starým a novým ovládáním



Obrázek 4.2: Rotace kamery[20]

Dalším problémem při přechodu mezi různými ovládáními spočívá v rotaci kamery. Například *TrackballControls* umožňuje rotaci kamery kolem osy z (*roll*), zatímco *OrbitControls* takovou rotaci neumožňuje. Při otáčení kamery v režimu *TrackballControls* a po následném přepnutí do *OrbitControls* je tedy možné se dostat do polohy, ve které by kamera neměla být.

Jednoduchým řešením by při nastavení ovládání bylo nastavení takové rotace, která není v konfliktu s žádným ovládáním. To bohužel není možné, protože každý režim ovládání si rotaci uzamkne tak, aby kamera mohla sledovat svůj *target*. Dokonce není možné nastavit rotaci kamery v mezičase mezi odebráním starého ovládání a přidáním nového.

Pro řešení tohoto problému bylo nakonec použito metody *reset*. Tato metoda nastaví kameře původní pozici a rotaci. Důsledkem jsou ne zrovna plynulé přechody mezi ovládáními, ale na druhou stranu alespoň fungující přechody.

Nastavení kamery

Nastavení kamery je komplikováno dvěma navazujícími mechanismy. První komplikací je vazba ovládání na kameru a druhou je úprava velikosti *viewport*u při zvětšování/zmenšování okna.

První komplikaci lze vyřešit jednoduše. Při změně kamery znovunastavit ovládání. Zde se nabízí otázka, jestli při prvotním nastavení nedojde k nastavení ovládání dvakrát. Nedojde, pokud v odpovídající *React* komponentě budou *effect hook*y v pořadí: nastavit ovládání a pak nastavit kameru. V tomto

pořadí dojde nejdříve k zavolání *setControls*, které se ihned ukončí, protože kamera neexistuje. Následně dojde k zavolání *setCamera*, která následně nastaví jak kameru, tak ovládání.

Zvětšování/Zmenšování okna je navázáno na kameru, protože je nutné zachovat správný poměr stran kamery tak, aby nebyl při změně rozměrů *viewport*u deformován obraz. Tím pádem při změně kamery je nutné změnit i *ResizeObserver*. Konkrétněji je třeba při změně kamery odebrat *event listener*y starého *observer*u (*disconnect*), vytvořit nový *observer* a napojit ho na rodiče *canvas*u.

4.2 Načítání 3D modelů

Hlavním důvodem proč nebyl použit *Babylon.js* tak, jako v bakalářské práci *Webová aplikace pro schvalovací proces 3D modelů*, je podpora formátů 3D modelů. *Three.js* podporuje všechny vyžadované formáty (FBX, GLB i OBJ). Při implementaci načítání byl identifikován problém s načítáním vícero souborů najednou.

4.2.1 Loadery

Načítání souborů je zastřešeno třídou *Loader*. Důležitou funkcionalitou této třídy je možnost přidat *request header*, protože již existující API při komunikaci vyžaduje autorizační token ve formě hlavičky *Authorization*.

Pro potřeby této práce byly konkrétně použity *GLTFLoader*, *FBXLoader* a *OBJLoader*. Metody *loader*ů, které jsou používány v této práci nejsou zcela totožné, takže práce s nimi nemůže být universální.

```
async function modelLoad(loader: FBXLoader | OBJLoader, url: string,
    requestHeaders: {[p: string]: string}, scene: THREE.Scene,
    onProgress?: (event: ProgressEvent<EventTarget>) => void)
{
    loader.setRequestHeader(requestHeaders)

    const group = await loader.loadAsync(url, onProgress)

    group.traverse( processObject3D );

    group.name=groupName
    scene.add( group );
    return group;
}
```

Tabulka 4.5: Načítání FBX a OBJ modelů

```
async function loadGLTF(url: string,
    requestHeaders: {[p: string]: string}, scene: THREE.Scene,
    onProgress?: (event: ProgressEvent<EventTarget>) => void )
{
    const loader = new GLTFLoader();
    loader.setRequestHeader(requestHeaders)

    const gltf = await loader.loadAsync(url, onProgress)

    gltf.scene.traverse( processObject3D );

    gltf.scene.name=groupName
    scene.add( gltf.scene );
    return gltf.scene;
}
```

Tabulka 4.6: Načítání GLTF modelů

Loadery FBXLoader a *OBJLoader* mají totožné rozhraní, tudíž byly použity v rámci obecné funkce *modelLoad*. *GLTFLoader* se od nich liší v návratové hodnotě pro asynchronním načítání, tudíž byl použit uvnitř specifické funkce *loadGLTF*. Obě funkce se řídí stejným postupem: nastavit *request header*, načíst model, připravit všechny *Object3D* pro zobrazení, přidat vše do 3D scény.

4.2.2 Načítání několika souborů najednou

Během implementace bylo zjištěno, že při použití vícero *loader*ů v jeden moment pro jednu scénu, nemůže být vyřešeno pomocí jednoduchých callbacků. Pokud by byly použity callbacky, které si nebudou vědomy ostatních souběžných načítání, došlo by k „přetahování“ o jeden *loading bar*. Pro uživatele by to znamenalo, že by pouhým okem byli schopní zaznamenat „skákání“ procent na *loading bar*u nahoru a dolů. Řešením je vytvoření třídy *LoadingCallbacksHandler*, která pomocí metody *participate* obohatí callback.

```
export class LoadingCallbacksHandler {
    private participants: Participant[] = []

    ...

    participate( onProgress?: (event: ProgressEvent<EventTarget>)=>void,
        size?: number )
    {
        if(!onProgress)
            return undefined

        const participant: Participant = new Participant(size)
        this.participants.push(participant)

        return (event: ProgressEvent<EventTarget>)=>{
            participant.onProgress(event.loaded, event.total)
            const accumulated = this.participants.reduce( this.accumulate,
                new Participant() )

            onProgress(  new ProgressEvent(event.type, {
                loaded: accumulated.loaded,
                total: accumulated.total
            }) )
        }
    }
}
```

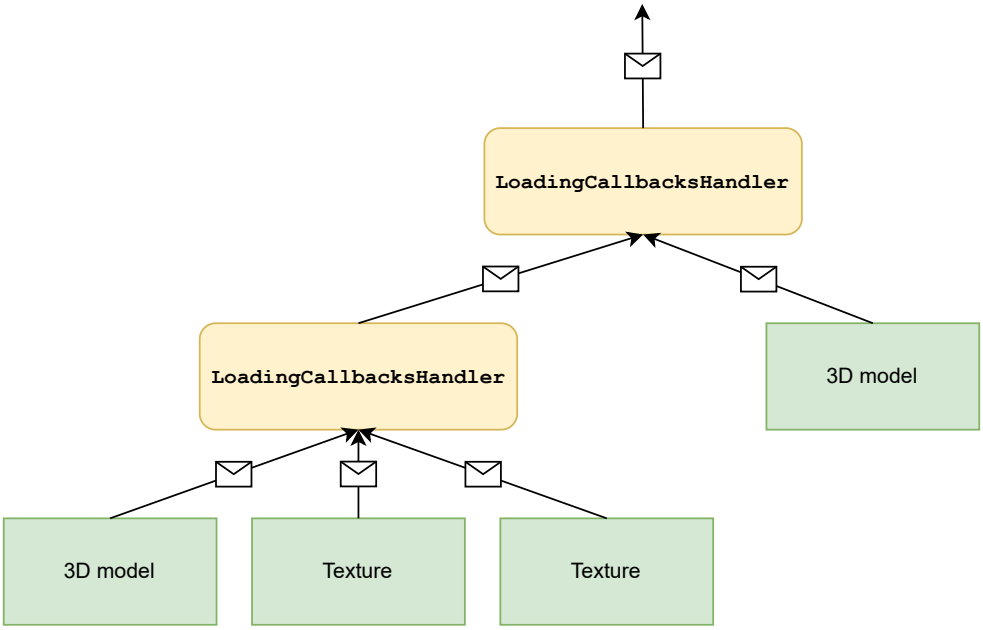
Tabulka 4.7: LoadingCallbacksHandler

Při každém volání, s výjimkou undefined callbacku, je vytvořen uvnitř *LoadingCallbacksHandler*u *Participant* a je vrácena nová funkce. Nová funkce je obohacený callback. Tato funkce používá svého *Participant*a k ukládání příchozích hodnot. Následně všechny *Participant*y uvnitř *LoadingCallbacksHandler*u využije k spočítání celkové velikosti dat k načítání a současného stavu načítání. Tyto informace nakonec vloží do původního callbacku.

```
const lch = new LoadingCallbacksHandler()

const model1 = this.loadModel(models[0], ..., lch.participate(onProgress))
const model2 = this.loadModel(models[1], ..., lch.participate(onProgress))

this.comparableModels[0] = {model: await model1, ...}
this.comparableModels[1] = {model: await model2, ...}
```

Tabulka 4.8: Příklad obohaceného callbacku

Díky tomu, že callback je zabalen do funkce stejného typu, tak nikdo mimo funkce, kde je callback *zabalen*, tuto informaci nemá a nemusí mít. To znamená, že tato funkcionalita může být vložena kamkoliv, bez nutnosti úprav v okolním kódu. Takže je i možné používat toto obohacení několikanásobně po sobě na jednom callbacku.



Obrázek 4.3: Další možnosti LoadingCallbacksHandleru

4.3 Správa aktivních a ne-aktivních modelů

Správa aktivních a ne-aktivních modelů probíhá během zobrazeni dvou 3D modelů do jedné scény. Pro manipulaci s těmito modely byly implementovány *Selector*y, které jsou schopny „aktivovat“ nebo „deaktivovat“ 3D model.

4.3.1 Selectory

*Selector*y (třídy implementující rozhraní *ModelSelector*) hrají klíčovou roli při odlišení aktivního a ne-aktivního modelu. *Selector*y vystavují metody *activate*, *deactivate* a *reset*. Jejich vstupem je datová struktura (*ComparableModel*), obsahující *Group* (reprezentuje načtený 3D model) a stav 3D modelu. Stavy 3D modelu jsou *default* (původní podoba modelu), *active* („zvýrazněný“ model) a *inactive* („upozaděný“ model). Na základě volané metody a stavu 3D modelu dojde k přechodu do konkrétního stavu.

```
export interface ComparableModel{
    model: THREE.Group,
    state: ComparableState
}

export default interface ModelSelector{
    activate(model: ComparableModel): void
    deactivate(model: ComparableModel): void
    reset(model: ComparableModel): void
}
```

Tabulka 4.9: Rozhraní ModelSelector

OpacitySelector

Jak už z názvu vyplývá, tento *Selector* je založený na průhlednosti 3D modelů. Konkrétněji manipuluje s parametrem *opacity* na materiálech přítomných na načteném 3D modelu. Jak už bylo zmíněno v návrhové části, aby byl zachován původní vzhled 3D modelů, musí se, v tomto kontextu, průhlednost násobit, respektive dělit. Z důvodu eliminace opakovaného násobení/dělení je nutné při přechodu mezi stavy 3D modelu znát jeho současný stav.

RenderOrderSelector

Tak, jak byl *RenderOrderSelector* naplánován v rámci návrhové části, byla by jeho implementace naprosto triviální. Bohužel navýšení *renderOrder* u každého *mesh*e nestačí. Pro dosažení kýženého efektu bylo nutné experimentovat s nastavením vykreslování materiálů na načteném 3D modelu.

Částečným řešením byla manipulace s *depth test*em (součást *3D-rendering pipeline*). Při promítání objektu na obrazovku se v *3D-rendering pipeline* porovnává hloubka (*z-value*) generovaného fragmentu v promítaném obraze s hodnotou již uloženou v *z-buffer* (*depth test*), a pokud je nová hodnota bližší, nahradí ji.[21] Při vypnutí zápisu ne-aktivního modelu do *z-buffer*u a při vyšší prioritě aktivního modelu, je dosaženo vykreslení aktivního modelu před ne-aktivní.

Při tomto nastavení je sice dosaženo efektu vytyčeného v návrhové části, ale dojde také ke zkreslení ne-aktivního modelu. Při vykreslování dochází ke konfliktu mezi vnitřní a vnější stranou ne-aktivního modelu. Tyto dvě strany se vykreslují přes sebe v závislosti na úhlu kamery. To lze eliminovat zakázáním vykreslování vnitřní strany modelu na všech materiálech přítomných v ne-aktivním modelu.

4.4 Synchronizace 3D scén

Synchronizace 3D scén probíhá během zobrazení dvou modelů do dvou odlišných 3D scén. Pro zprostředkování komunikace mezi 3D scénami byl implemen-

tován *Synchronizer*. Jednotlivé úkony v rámci synchronizace jsou vyjádřeny synchronizační funkcí.

4.4.1 Synchronizer

Synchronizer je klíčová třída pro synchronizaci 3D scén. Jedná se o mediátora/prostředníka v komunikaci primárně mezi instancemi *SynchronizedViewLogic*. Jak už bylo zmíněno v návrhu, *Synchronizer* řeší problematiku nutnosti znalosti adresáta synchronizačních zpráv.

```
interface Target {
    id: number
    synchronize: (msg: SynchronizedAttributes) => void
}

export default class Synchronizer{
    private targets: Target[] = []

    register( id: number, synchronize: (msg: SynchronizedAttributes ) => void ){
        this.targets.push({id, synchronize})
    }

    remove( id: number ){
        this.targets = this.targets.filter(t => t.id != id)
    }

    update( msg: SynchronizedAttributes, id: number? ){
        for(const t of this.targets){
            if(id !== t.id)
                t.synchronize(msg)
        }
    }
}
```

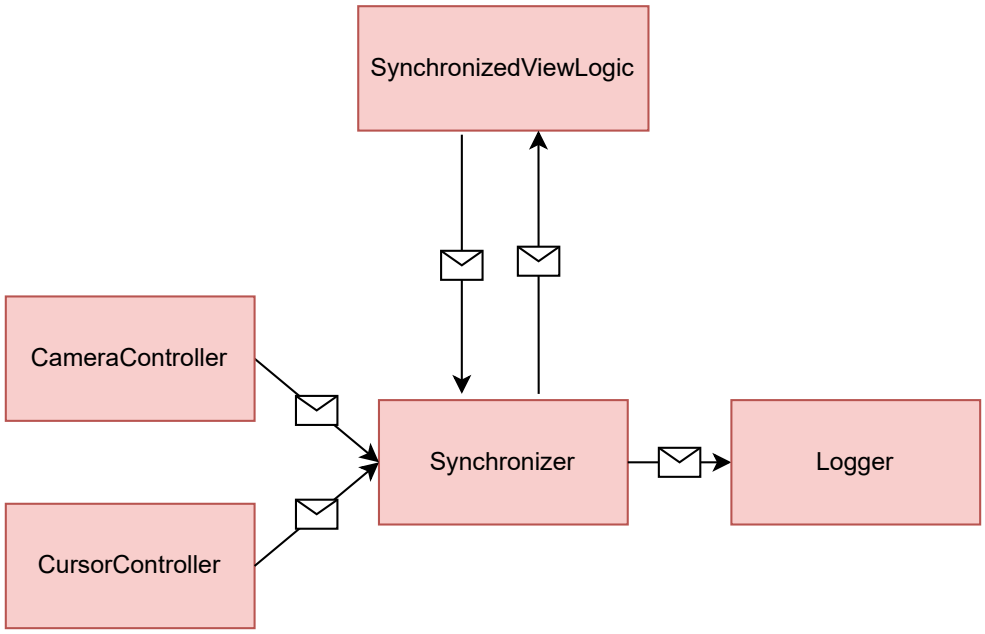
Tabulka 4.10: Implementace Synchronizeru

Samotná implementace této třídy je „na pár řádků“. Fungování třídy *Synchronizer* je založené na operacích na poli adresátů (*target*ů).

Register
Každý subjekt, který má zájem přijímat synchronizační zprávy pomocí *Synchronizer*u, se musí nejdřive „registrovat“ (*register*). Při registraci musí subjekt poskytnout svůj identifikátor (jakékoli číslo) a funkci k předání synchronizační zprávy. Zavoláním této metody dojde k uložení těchto údajů do pole *target*ů.

Update
Pokud chce kdokoli odeslat synchronizační zprávu všem „registrovaným“ v *Synchronizer*u, tak stačí pouze zavolat metodu *update*. Tato metoda vyžaduje synchronizační zprávu (*SynchronizedAttributes*) a případně identifikátor odesílatele. Uvnitř dojde u každého *target*u k zavolání jeho funkce, s výjimkou *target*u s *id* odesílatele.

Remove
Pro odhlášení odběru zpráv ze *Synchronizer*u slouži metoda *remove*. Na úrovni třídy dojde k odebrání všech *target*ů s dodaným *id* z pole.



Obrázek 4.4: Další možnosti Synchronizeru

V této implementaci *Synchronizer*u jsou kladeny minimální nároky na odesílatele a adresáta. Od adresátů je vyžadován pouze identifikátor a funkce. Od odesílatele je vyžadováno, aby zpráva respektovala *interface SynchronizedAttributes*. To znamená, že zprávy může odebírat a odesílat prakticky cokoli. V rámci návrhu bylo zmíněno, že je možné *Synchronizer* využit i při logování uživatelských akcí. Při znalosti požadavků na odesílatele synchronizačních zpráv je možné přidat např. ovládací prvek, který nastaví kameru nebo kurzor na konkrétní pozici.

4.4.2 Synchronizační funkce

Synchronizační funkce je komunikační prostředek, nacházející se uvnitř třídy *SynchronizedViewLogic*. Tato funkce je vložena do *Synchronizer*u v rámci registrace pro odebírání synchronizačních zpráv. Dále je používána *Synchronizer*em pro komunikaci ve směru ze *Synchronizer*u do *SynchronizedViewLogic*.

```
import * as SynchronizingTasks
    from "../synchronization/SynchronizingTasks";

export default class SynchronizedViewLogic extends ModelViewLogic{

    ...

    private readonly syncFun = (msg: SynchronizedAttributes)=>{

        SynchronizingTasks.setCameraPosition(msg, this.camera)
        SynchronizingTasks.setCameraTarget(msg, this.controls)
        SynchronizingTasks.setCameraZoom(msg, this.camera)
        SynchronizingTasks.setCursorPosition(msg, this.camera,
            this.cursor, this.loadedModel)
    }

    ...

}
```

Tabulka 4.11: Implementace synchronizační funkce

Pokaždé, když je na *Synchronizer*u zavolána metoda *update*, tak je synchronizační zpráva rozeslána všem *target*ům pomocí jejich synchronizačních funkcí. Synchronizační funkce pak na základě doručené zprávy nastaví vlastnosti objeků uvnitř 3D scény. Pro jednoduchost je synchronizační funkce (*syncFun*) rozdělena na několik volání funkcí, které spravují konkrétní část synchronizace. Tyto funkce byly pojmenovány *synchronizing tasks*.

```
export const setCameraPosition =
    ( attr: SynchronizedAttributes, camera?: THREE.PerspectiveCamera ) => {

        if(attr.cameraPosition)
            camera?.position.set(
                attr.cameraPosition.x,
                attr.cameraPosition.y,
                attr.cameraPosition.z
            )
    }
```

Tabulka 4.12: Ukázka synchronizačního tasku

Například *synchronizing task setCameraPosition* se stará pouze o pozici kamery. Vstupnímy parametry jsou synchronizační zpráva (*SynchronizedAttributes*) a reference na kameru. Nejdříve dojde ke kontrole, zda synchronizační zpráva obsahuje atributy pro nastavení pozice kamery. Pokud ano, tak dojde k nastavení pozice kamery pomocí 3D souřadnic. Pokud by náhodou neexistovala kamera uvnitř 3D scény, nedojde k zavolání *camera?.position.set*, ani nebude vyhozena výjimka (díky *null check*u).

4.5 Build a deployment

Pro build a deployment knihovny byly vybrány nástroje na fakultním *GitLab*u (https://gitlab.fit.cvut.cz). Konkrétně se jedná o *GitLab CI/CD* a *GitLab Package Registry*. *GitLab CI/CD* je používán pro automatizovaný build a deploy-

ment. *GitLab Package Registry* slouží k uložení a distribuci výsledných *package*ů.

Volba fakultního *GitLab*u je logická, jelikož se jedná o projekt v rámci FIT ČVUT. *GitLab CI/CD* je to nejjednodušší řešení pro automatizovaný build pro projekty uložené v *GitLab*u. Deployment do *GitLab Package Registry* ovšem není tím nejpřímočařejším řešením. Pro naprostou většinu knihoven je nejlepší volbou *npm registry*, které o sobě tvrdí, že je největším softwarovým registrem na světě. Nevýhodou *npm registry* je, že je primárně veřejný a že soukromý *package* je zpoplatněný. Jelikož tato knihovna vznikla primárně za účelem podpory projektů, jako jsou Věnná města českých královen, tak je vhodné použít registr, který je privátní a v ideálním případě na infrastruktuře FIT ČVUT.

4.5.1 CI/CD pipeline

CI/CD pipeline je rozdělena do *stage*ů a ty jsou dále děleny na jednotlivé *job*y. V tomto případě se jedná o *stage install*, *build*, *deploy* a od nich odvozené *job*y *install-job*, *build-job* a *deploy-job*.

Install

V rámci *install stage* dojde k zavolání příkazu *npm install*, což nainstaluje *dependencies* pro chod knihovny. Tyto *dependencies* jsou pak standardně uloženy v souboru *node_modules*. Aby mohly být *node_modules* použity v dalších *stage*ích, tak je vytvořen *artifact*.

Build

Uvnitř *build stage* je volán příkaz *npm run build*, což je v *package.json* definovaný script pro volání *rollup -c*. *Rollup* je bundler pro *JavaScript*ové moduly i s podporou *TypeScript*u. V tomto případě je *Rollup* nakonfigurován tak, aby do složky *dist* vytvořil soubor *index.js* a deklarační soubory (*.d.ts*) odpovídající zdrojovým souborům (*.tsx, .ts*) uvnitř složky *src*. Opět je vytovřen *artifact*, tentokrát pro složku *dist*.

Deploy

Na závěr proběhne *deploy stage*. Ta obsahuje vložení URL registru do konfiguračního souboru (*.npmrc*) a volání *npm publish*. To samo o sobě nestačí a je nutné předem specifikovat URL registru ještě v souboru *package.json*. Bohužel nebylo nalezeno proveditelné řešení uvnitř *CI/CD pipeline*, tudíž je toto URL specifikováno již ve verzi *package.json*, která je dostupná přímo z *GitLab* repozitáře. Dalším problémem je verzování, které zakazuje *deploy* verze knihovny, která již v registru existuje. Řešením je nechat verzování na

vývojáři a při konfliktu verzí „tiše“ selhat. Pří konfliktu verzí je výsledkem *deploy stage*, která selhala, ale *CI/CD pipeline*, která prošla s upozorněním.

Obrázek 4.5: CI/CD pipeline s konfliktem verzí

Pro celou *CI/CD pipeline* byl použit *node:14-buster image*. Vývoj knihovny byl proveden v *node:19.0.3*, ale u této verze nebylo možné provést *deploy*. *Node* po verzi 14 obsahuje chybu/změnu, která neumožňuje provést autorizaci pomocí tokenu.

4.6 Návod pro instalaci a integraci

Díky tomu že knihovna je deploynuta jako *npm package*, lze s ní manipulovat standardně pomocí nástroje *npm*. Jelikož je knihovna uložena v soukromém repozitáři, tak je nutné před samotnou instalací se nejdříve autorizovat. Po instalaci jsou pomocí importu dostupné jednotlivé *React* komponenty, pomocné *Enums* a *Interfaces* pro komunikaci s nimi.

4.6.1 Instalace

Jak bylo zmíněno v sekci *Build a deployment*, knihovna je uložena formou *npm package* v soukromém repozitáři ve fakultním *GitLab*u. V době implementace byl hlavním zamýšleným způsobem přístupu do tohoto repozitáře za použití nástroje *npm*.

K tomu, aby *npm* měl přístup do soukromého repozitáře, je nutné upravit konfigurační soubor *.npmrc* (v kořenovém adresáři projektu). Nejjednodušším způsobem je zadefinovat v *.npmrc* zdroj s autorizačním tokenem a *scope*, pro který je použit soukromý repozitář.

S takto modifikovaným *.npmrc* lze knihovnu jednoduše nainstalovat pomocí příkazu *npm i @antospa2/interactive-3d-viewer*. Po dokončení instalace je tato knihovna uložena standardně v *node_modules* a je vedena standardně v *package.json* mezi ostatními *dependencies*.

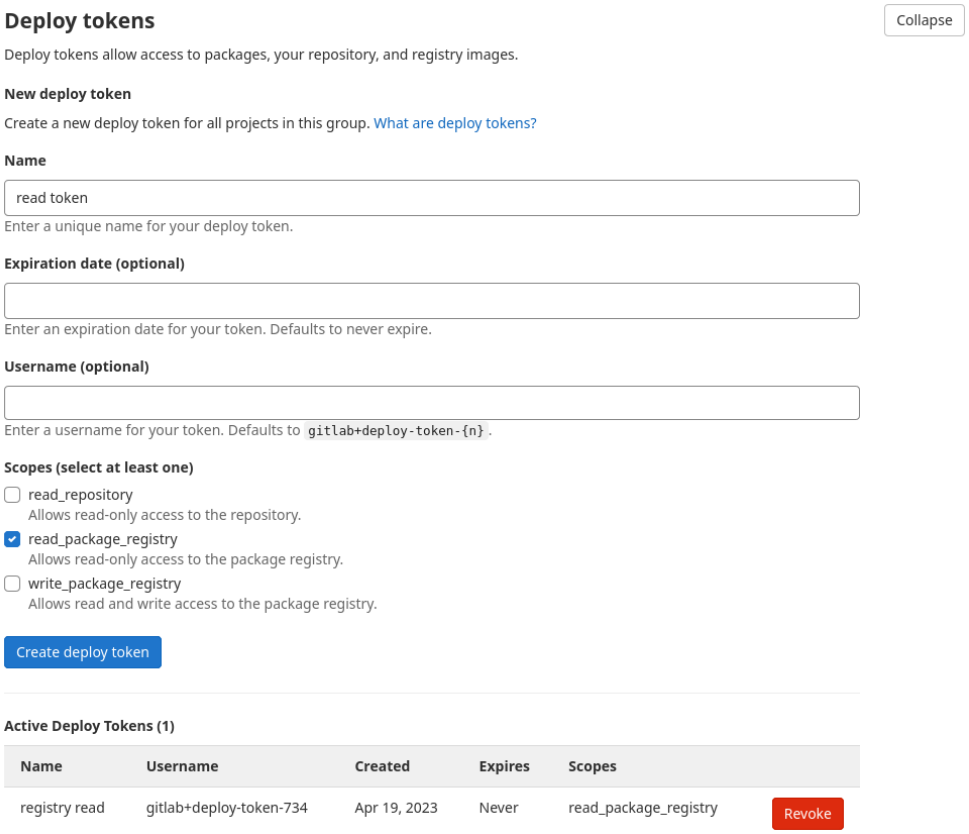
```
//gitlab.fit.cvut.cz/:_authToken=[deploy token]
@antospa2:registry=https://gitlab.fit.cvut.cz/api/v4/packages/npm/
```

Tabulka 4.13: Ukázka .npmrc

Získání přístupového tokenu

Přístupový token lze vygenerovat ve webovém rozhraní fakultního *GitLab*u. Token je možné vygenerovat v *Settings/Repository/Deploy tokens*. *Name* a *Username* v tomto případě nehrají u tokenu žádnou roli. Důležité je, aby token byl součástí *scope read_package registry*.

Deploy tokens mohou být spravovány pouze rolemi *Maintainer* a vyšší. V reálném životě to znamená, že osoba/team, mající zájem o integraci této knihovny, si musí vyžádat přístup od osoby s dostatečně velkým oprávněním. Když tato osoba/team získá *deploy token*, tak na ni/něj nejsou kladeny žádné další požadavky. To znamená, že osoba/team, mající zájem o integraci této knihovny, nemusí mít přímý přístup ke *GitLab* projektu, dokonce ani přístup do fakultního *GitLabu*.



Obrázek 4.6: Vytvoření deploy tokenu

Alternativní způsob přístupu

Alternativním způsobem přístupu ke knihovně je stažení *artifact*u z *build stage* z *CI/CD pipeline*, nebo stažením zabalené složky z *Package registry*. V obou případech by pak měly být stažené soubory vloženy do složky se zdrojovými kódy a ne do složky s ostatními knihovnami (*node_modules*). Pokud by byly vloženy do *node_modules*, tak dříve nebo později nástroj spravující *dependencies* soubory smaže.

U tohoto alternativního způsobu přístupu byly nalezeny problémy při *type check*u. První spuštění po vložení souborů s knihovnou vždy selže, ale ty následující projdou bez chyb.

4.6.2 Integrace

Pokud je knihovna nainstalována zamýšleným způsobem (pomocí *npm*), tak její soubory jsou uloženy v *node_modules*. Díky tomu lze ke knihovně přistupovat standardně pomocí klíčového slova import. Například získání *React* komponenty *ModelView* lze provést následujícím způsobem: *import* {*ModelView*} *from "antospa2/interactive-3d-viewer"*.

Vložení *React* komponent z knihovny do již existující komponenty probíhá standardně jako u jiných *React* komponent. Kam přesně do kódu umístit *React* komponentu z knihovny závisí na formě komponenty, do které chceme vkládat. U *Class* komponent musí dojít k vložení do návratové hodnoty metody *render* a u *Function* komponenty do návratové hodnoty funkce.

```
<ModelView style={ style }
           model={ model }
           cameraOption={ camera }
           controlsOption={ controls }
           requestHeaders={ requestHeaders }
           environmentParams={ environment }
/>
```

Tabulka 4.14: Integrace ModelView

```
<ModelCompare style={ style }
              models={ models }
              cameraOption={ camera }
              controlsOption={ controls }
              requestHeaders={ requestHeaders }
              activeModelIndex={ activeModel }
              selectorOption={ selector }
              environmentParams={ environment }
/>
```

Tabulka 4.15: Integrace ModelCompare

```
<CompareViews styles={ styles }
              models={ models }
              cameraOption={ camera }
              cursorOption={ {style: cursorStyle, event: cursorEvent} }
              requestHeaders={ requestHeaders }
              environmentParams={ environment }
/>
```

Tabulka 4.16: Integrace CompareViews

Každá z výsledných *React* komponent byla stavěna tak, aby dynamicky reagovala na změny parametrů. To znamená, že již existující komponenta, u které dojde ke změně vstupních parametrů, podstoupí jen nutné minimum změn.

Vstupní parametry

Každý vstupní parametr, kromě *requestHeaders*, mají jasně definovanou podobu. Jejich podoba je definována pomocí *Interfaces* nebo *Enums*, které je možné importovat společně s *React* komponenty. *RequestHeaders* jsou definovány vágně, aby jejich prostřednictvím bylo možné vložit cokoliv do hlavičky *HTTP request*ů. Jejich definice je převzána z *Three.js loader*ů.

Pro většinu vstupních parametrů platí, že jejich změny jsou sledovány na základě hodnot. To znamená, že pokud komponenta přijme jako parametr jiný objekt, ale se stejnými hodnotami, nevyhodnotí tuto událost jako změnu.

Tento princip ovšem neplatí pro *requestHeaders* a *environmentParams*. *RequestHeaders* hodnoty nového a starého objektu není možné jednoduše porovnat kvůli jejich prakticky neomezeným počtu variant. U *environmentParams* lze sledovat změny jednotlivých hodnot, ale množství sledovaných hodnot je i tak neudržitelné. Z toho plyne, že pokud jeden z těchto parametrů má být změněn, tak je třeba použít nový objekt. Na druhou stranu, pokud nemá být provedena žádná změna, je nutné, aby byl použit ten samý objekt!

Vstupní parametry	ModelView	ModelCompare	CompareViews
activeModelIndex	-	O	-
cameraOption	O	O	O
controlsOption	O	O	-
cursorOption	-	-	O
environmentParams	O	O	O
model(s)	X	X	X
requestHeaders	O	O	O
selectorOption	-	O	-
style(s)	O	O	O

nepodporováno -, volitelné O, povinné X

Tabulka 4.17: Komponenty a jejich vstupní parametry

activeModelIndex Volba aktivního modelu. 0 - první, 1 - druhý 3D model.

cameraOption Podle *enum CameraOption*. Přepínaní mezi perspektivní a ortografickou kamerou.

controlsOption Podle *enum ControlsOption*. Přepínání mezi orbitálním a trackball ovládáním.

cursorOption Objekt obsahující *style*, podle *enum CursorStyleOption*, a *event*, podle *enum CursorEventOption*. Přepínaní mezi *line*, *sphere cursor*em a eventy, při kterých se aktualizuje jeho pozice.

environmentParams Podle *interface EnvironmentParams*. Nastavení mlhy, podlahy a mřížky.

model(s) Objekt s *url* a *format*, podle *enum ModelFormat* (pole dvou objektů).

requestHeaders Hlavičky *HTTP request*ů, které jsou použity při získávání 3D modelů.

selectorOption Podle *enum SelectorOption*. Přepínání mezi *renderOrder* a *opacity selector*y.

style(s) Podle *React.CSSProperties*. Styl pro div, pod kterým operuje *canvas*.

4.7 Postupy při rozšiřování knihovny

Knihovna, která je výsledkem této diplomové práce, je napsána tak, aby byla co nejvíce modulární. Tento návod obsahuje popis postupu, jak přidávat *React* komponenty a navazující třídy s logikou nebo možnosti do *builder*ů.

React komponenty a třídy s logikou

Jak už bylo dříve vysvětleno tak, *React* komponenty jsou určeny k manipulaci s HTML a jsou primárním produktem knihovny. Třídy s logikou jsou určeny k manipulaci s 3D scénou a konzumentovi knihovny jsou skryty.

Vytváření nových tříd s logikou může být relativně jednoduché a elegantní. Doporučeným způsobem je vytváření tříd, které rozšiřují již existující třídy s logikou. Jednoduchost a elegance spočívá v tom, že stačí implementovat jen to, v čem se třídy liší.

Na druhou stranu vytváření nových *React* komponent je z velké části „*copy-paste*“. Z podstaty *function component* vyplývá, že tyto komponenty nelze rozšiřovat tak, jak jsou rozšiřovány třídy s logikou. Doporučeným postupem je tedy zkopírovat již existující komponentu a tu následně upravit. Aby *React* komponenta byla součástí knihovny po *build*u, je nutné aby byla přidána mezi ostatní exportované moduly v *index.ts* (v kořenovém adresáři).

Builders

Pokud je potřeba vytvořit složitě nějaký objekt nebo je potřeba se rozhodnout o typu objektu a následně ho vytvořit, tak je tato logika přesunuta do samostatného modulu.

Těmto modulům se říká *Builders*. *Builders* obsahují funkci *build*, která je často parametrizovaná. Parametrizovaný *build* obsahuje vstupní proměnnou typu *Option* (např. *SelectorOption*). *Option* je *enum* obsahující možnosti (např. *renderOrder* a *opacity*), na základě kterých se vybere, jaký objekt sestavit (např. *RenderOrderSelector* a *OpacitySelector*).

```
export enum SelectorOption{
    renderOrder,
    opacity
}

export function build(option: SelectorOption): ModelSelector{
    switch (option){
        case SelectorOption.renderOrder:
            return new RenderOrderSelector();
        case SelectorOption.opacity:
            return new OpacitySelector();

        default:
            throw new Error("Unknown selector option");
    }
}
```

Tabulka 4.18: SelectorBuilder

Přidání další možnosti je přímočaré. Nejdříve je potřeba vytvořit třídu, ve které bude implementována nová možnost. Tato třída musí respektovat návratový typ *build* funkce (např. další *ModelSelector*). Dále je nutné přidat název nové možnosti do odpovídajícího *enum* (např. do SelectorOption). Nakonec se přidá tato možnost do *switch*e uvnitř *build* funkce.

V tomto procesu existuje pár výjimek. U *CursorBuilder* je nutné přidávat kurzory buď rozšiřující *Cursor2D* nebo *Cursor3D*. Na základě jejich typu je vybrán způsob synchronizace. Po přidání možnosti u *CameraBuilder* je nutné ještě přidat způsob *resize*ování do *CanvasResizeObserver*.

ControlsBuilder sice lze rozšiřovat standardně, ale je u něj doporučena opatrnost. Příčinou je chybějící jednotný *interface*, který byl, pro potřeby této knihovny, vytvořen uměle.

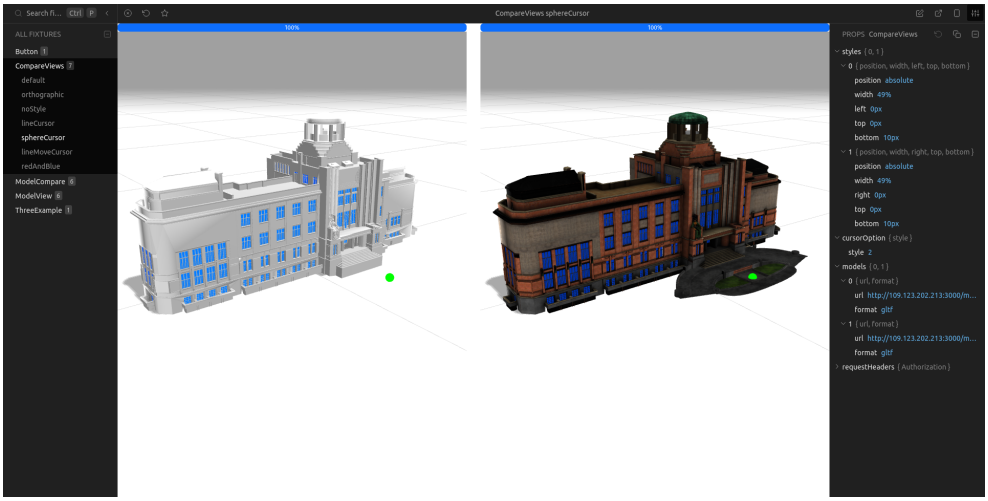
Kapitola 5

Testování

V rámci této kapitoly je popsáno, jak probíhalo testování během vývoje knihovny, uživatelské a akceptační testování. Jelikož výsledkem implementace je knihovna, bylo nutné, ještě před samotným uživatelským testováním, vytvořit aplikaci, která mohla být zpřístupněna testerům.

5.1 Testování během vývoje

Testování během vývoje bylo provedeno pomocí *React Cosmos*. Pro každou komponentu (*ModelView*, *ModelCompare*, *CompareViews*) byly vytvořeny testovací scénáře (*fixtures*). Testovací scénáře lze rozdělit na obecné a specifické pro komponentu. Obecné scénáře jsou například *default* (specifikovaný model a HTML styl), *orthographic* (*default* s ortografickou kamerou) nebo *redAndBlue* (*default* se silnou mlhou, červeným pozadím a modrou podlahou). Specifické scénáře jsou například *trackball* (*default* s *trackball* ovládáním kamery), *lineCursor* (*default* s *lineCursorem* a eventem *pointerdown*) nebo *opacity* (*default* s *OpacitySelector*em).



Obrázek 5.1: Testování pomocí React Cosmos

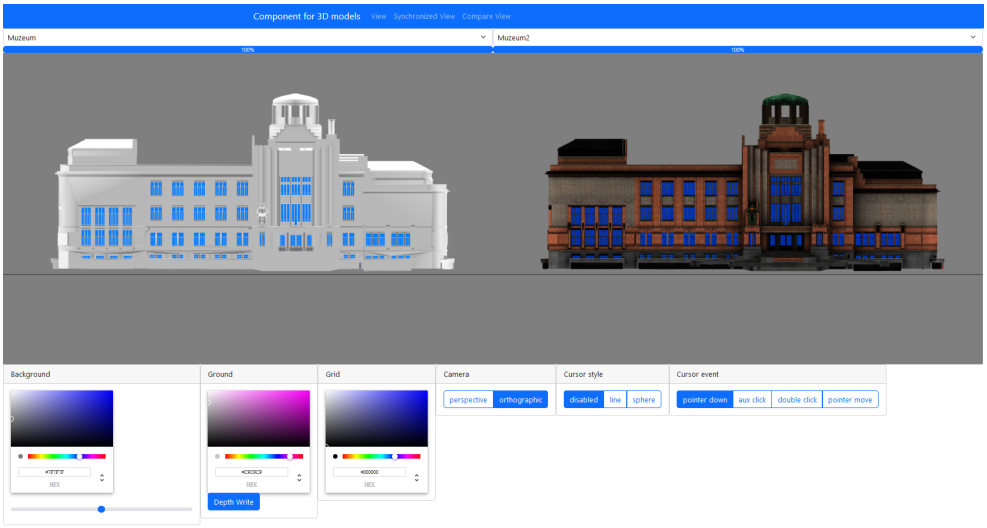
Psaní testovacích scénářů společně s implementací funkcionalit by nebylo možné bez *hot-reload*. Díky *hot-reload* dojde při každé uložené změně kódu k téměř okamžitému znovunačtení *React Cosmos*. Použití *React Cosmos* v kombinaci s logováním operací a id komponent znatelně urychlilo proces identifikace a odstranění chyb.

5.2 Aplikace pro uživatelské testování

Za účelem uživatelského testování bylo nutné použít prostředek, který demonstruje možnosti implementované knihovny. Relevantními možnostmi je použití *React Cosmos* nebo vytvoření nové webové aplikace.

První možností bylo využití již existujících testovacích scénářů v rámci *Testování během vývoje*. Tato možnost není ideální, protože *React Cosmos* není určený k uživatelským testům. Testeři by v tomto případě mohli být zmateni ovládacími prvky, které slouží k ladění jednotlivých komponent.

Druhou možností je vytvoření vlastní *React* webové aplikace, která bude sloužit pouze k uživatelskému testování. Tato možnost se ukázala být ideální kvůli „čistotě“ uživatelského rozhraní. Testovací aplikace byla implementována za pomoci knihoven *React Bootstrap* a *React Color*. Za použití *React Color* byly implementovány výběry barev a pomocí *React Bootstrap* byly implementovány všechny ostatní ovládací prvky (navbar a skupiny tlačítek).



Obrázek 5.2: Ukázka testovací aplikace

5.3 Uživatelské testování

Testování bylo provedeno na skupině čtyř testerů. Proběhlo vzdáleně pomocí platformy *Teams*. Konkrétněji se jednalo o individuální videohovory se sdílením obrazovky testera. Během testování byla každému testerovi dostupná testovací webová aplikace na stránce `http://vmck-dev.sic.cz:8080/`.

Výsledky testování byly uspořádány podle *issue types* inspirované *Jira Service Management issue types*. Prvním typem je *Incident* označující chyby, které přímo rozbíjí fungování aplikace. Dalším typem je *Change*, který zastřešuje chyby, jejichž řešení implikuje změnu v již existujících modulech. Tyto chyby působí diskomfort testerovi při používání prvků aplikace, ale nerozbíjí funkčnost. Posledním typem je *New feature* reprezentující chybějící možnosti, které jsou buď očekávány, nebo doporučeny testerem.

5.3.1 Incident issues

Zablokované otáčení kamery

Jedná se o chybu, která při dvojkliku a tažení, zabraňuje uživateli otáčet kamerou. Při uvolnění tlačítek myši je pohyb kamery odblokován. Chyba byla zachycena pouze u jednoho testera a u jiných testerů nebyla úspěšně replikována. Jelikož nebylo možné chybu replikovat, není možné ani vypátrat původ této chyby. Na základě provedených testů usuzuji, že tato chyba se projevuje jen vzácně a je částečně způsobena nestandardním způsobem manipulace s 3D scénou.

Scrollování

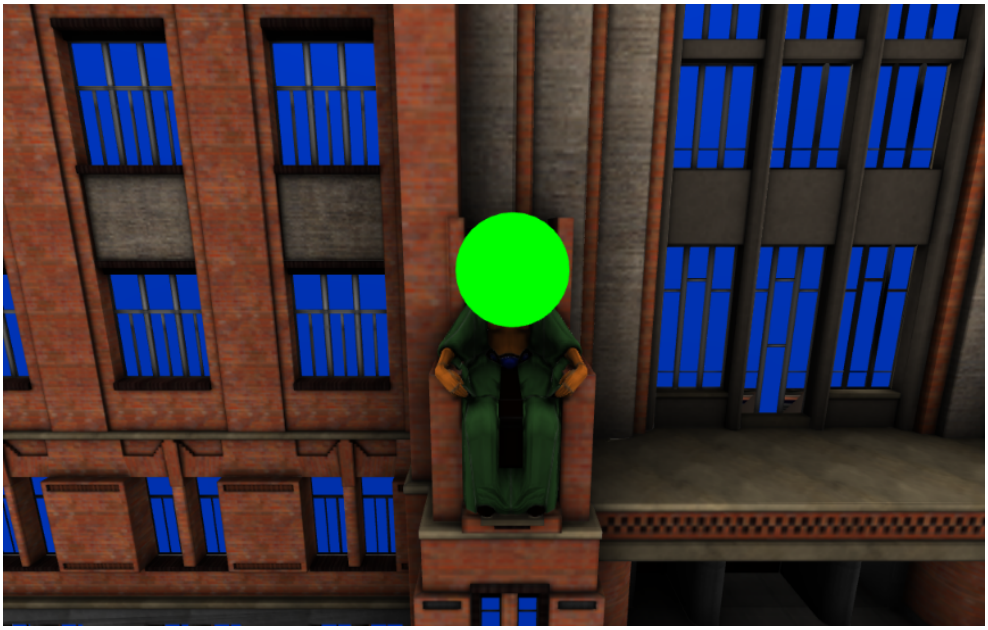
V jednom případě se objevila chyba při přibližování a oddalování kamery. Během manipulace s kamerou, pomocí prostředního tlačítka myši, došlo ke scrollování celé webové stránky.

Tuto chybu je možné opravit pomocí *CSS*. Nastavením *hover* eventu *scroll-behavior* na *unset* buď pro celou komponentu, nebo jen pro canvas.

5.3.2 Change issues

Podoba SphereCursoru

SphereCursor je zelená koule, která se zobrazí ve 3D scéně tam, kam ukazuje kurzor myši. Dokud uživatel používá *SphereCursor* v kombinaci s *pointermove* a ukazuje pohybem myši oblast na modelu, je vše v pořádku. Problém se projeví v případě, kdy uživatel označí bod na 3D modelu pomocí *SphereCursor* za účelem ukázání tohoto konkrétního bodu. V ten moment *SphereCursor* blokuje výhled na tento konkrétní bod. Vhodným řešením je vykreslování pouze obrysu *SphereCursor*.



Obrázek 5.3: Chyba SphereCursoru

5.3.3 New feature issues

Reset kamery

Každý tester při manipulaci s 3D scénou dospěl do bodu, kdy vyžadoval od aplikace možnost vrátit kameru zpět do výchozí pozice. Na tuto funkcionalitu nebylo během návrhu ani implementace myšleno.

Pro webové aplikace to znamená, že do nich bude přidáno tlačítko, které při stisknutí zavolá „metodu“ na použité komponentě z knihovny. V knihovní *React* komponentě bude třeba přidat tuto „metodu“ a v rámci ní volat metodu na odpovídající třídě s logikou. Tento postup je dopodrobna vysvětlen v článku *How To Call A Child Function From A Parent Component In React?*[22]. Metoda pro reset kamery nemusí být implementována ve všech třídách s logikou, ale pouze v *ModelViewLogic*. Tím bude distribuována do ostatních tříd principem dědění.

Nastavení citlivosti

Dalším objeveným problémem je citlivost ovládání 3D scény. Citlivost ovládání se mění v závislosti na použité kameře a ovládání. V rámci knihovny je citlivost pro každé ovládání fixně nastavena. Toto nastavení se ukázalo být nedostatečné. Konkrétně u kombinace ortografická kamera a *TrackballControls* byla zaznamenána velmi nízká citlivost.

Řešením by bylo přidání možnosti nastavení citlivosti. V té nejjednodušší formě by se mohlo jednat o jednu hodnotu, která bude násobičem původního nastavení. Tato hodnota by byla předána třídě s logikou a ta by následně citlivost nastavila. Pro webové aplikace by to znamenalo přidání například *sliderů* podobného tomu u nastavení mlhy.

Velikost kurzorů

Problémem obecně u kurzorů je jejich velikost. Velikost kurzorů je nastavena fixně tak, aby byly z větší vzdálenosti viditelné a aby neblokovaly z menší vzdálenosti výhled na velké části modelu.

Řešením je škálovat velikost na základě vzdálenosti od kamery. To by vyžadovalo rozšířit kurzory o metodu například *scale*, která by byla volána pokaždé, když bylo manipulováno s kamerou. Na podobném principu již funguje synchronizace kamery.

Přepínání mezi pozicemi kamery

Jako příjemná možnost bylo navrženo zjednodušené ovládání pomocí *NumPad*u. Tím je myšleno, že by klávesy 2, 4, 6, 8 byly využity jako směrové

klávesy. Například stlačením klávesy 6 by se kamera přesunula do pozice, kdy by uživatel viděl „pravou“ stranu 3D modelu.

Toho by se dalo docílit přidáním nového ovládání, nebo úpravou těch stávajících. Postup přidání nového ovládání je popsán v *Postupy pro rozšiřování knihovny.*

Zobrazení os

Pro lepší orientaci v prostoru bylo navrženo, nad rámec mřížky na podlaze, vytvořit možnost zobrazit souřadnicové osy. Při zobrazení os bude pro uživatele jednodušší rozpoznat, na kterou stranu modelu se právě dívá.

Nejjednodušším řešením je použít již existující *AxesHelper*. Tento *Helper* by se mohl vytvářet v rámci budování scény. Logicky se pak nabízí možnost zobrazení os přidat do *EnvironmentParams*.

Intenzita a barva světla

Pro budoucí využitelnost knihovny by bylo vhodné přidat nastavení osvětlení 3D scény. V současnosti jsou světla nastavená tak, aby nepřesvicovala standardní modely a zároveň aby maximálně nasvítila poškozené modely (modely z API, které se jeví extrémně tmavé). V případě, kdy by bylo poskytnuto nastavení intenzity osvětlení, by si mohl uživatel nasvicovat scénu dle vlastních preferencí.

Jelikož je manipulace se světly ve stejném modulu jako nastavení barvy země a pozadí, tak by bylo logické nastavení osvětlení přidat do *EnvironmentParams*. Ovládací prvky by mohly vypadat totožně jako u nastavení barvy pozadí a intenzity mlhy.

Shadeless režim

Dalším řešením tmavých modelů, navrženým během testování, je vytvořit pro 3D modely *shadeless* režim. Jednalo by se o režim, ve kterém budou 3D modely ignorovat veškeré osvětlení a samy budou „vyzařovat“ barvy svých textur.

To by vyžadovalo přidání nového parametru do *React* komponent a metodu na *ModelViewLogic* a *ModelCompareLogic*, která by byla schopna manipulovat s materiály 3D modelů.

5.3.4 Shrnutí

Uživatelské testování bylo velice úspěšné. Během něho bylo odhaleno relativně malé množství chyb a jejich většina je snadno opravitelná. Kromě chyb bylo posbíráno několik relevantních podnětů k dalšímu rozšíření aplikace. Mimo

výše zmíněných chyb bylo používání ovládacích prvků a ovládání 3D scény dle představ testerů.

5.4 Akceptační testování

Toto testování bylo provedeno se zadavatelem na fyzické schůzce. Zadavatelovi byl, pomocí testovací aplikace, odprezentován výsledek implementační části. Během tohoto testování bylo ověřeno, zda byly splněny požadavky na výslednou knihovnu. Ověřování probíhalo oproti zjištěným funkčním a nefunkčním požadavkům.

Kritéria	Výsledek
N1: Kompatibilita s React webovými aplikacemi	splněno
N2: Komunikace s již existujícím API	splněno
N3: Uživatelská přívětivost	splněno
F1: Podpora formátů FBX, GLB a OBJ	splněno
F2: Pokročilé ovládání kamery	splněno
F3: Variabilní prostředí uvnitř 3D scény	splněno
F4: Podpora externích ovládacích prvků	splněno
F5: Zobrazení dvou modelů do jedné 3D scény	splněno
F6: Synchronizované zobrazení 2 modelů do odlišných 3D scén	splněno
F7: Příručka pro integraci	splněno

Tabulka 5.1: Akceptační kritéria

Zadavatel byl spokojen s implementací knihovny a potvrdil splnění všech funkčních i nefunkčních požadavků. Tímto je akceptační testování považováno za úspěšně dokončené.

Závěr

Hlavním cílem této diplomové práce byla implementace knihovny obsahující universální webové komponenty pro zobrazení 3D modelů. Před samotnou implementací bylo třeba provést analýzu a návrh. Po dokončení implementace bylo vhodné výslednou knihovnu ještě otestovat.

Analytická část této práce se věnovala hlavně sběru požadavků a zkoumání již existujících aplikací na poli zobrazování 3D modelů. Požadavky na knihovnu byly zjištěny přímo od zadavatele práce a následně byly rozděleny na funkční a nefunkční požadavky. Zkoumané aplikace byly navázány na projekt *Věnná města českých královen*. Konkrétněji se jednalo o bakalářskou práci Pavla Antoše *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*[1] a o projekt Ing. Jiřího Chludila a Ing. Petr Pauše Ph.D. *Lite průvodce*[2] Analytickou část bylo ještě vhodné rozšířit o krátkou analýzu technologií vybraných pro implementaci (*React*, *React Cosmos*, *Three.js*).

V rámci kapitoly Návrh bylo ustanoveno rozdělení modulů na *React* komponenty a třídy s logikou. Velkou část kapitoly zabírá zamyšlení nad porovnáváním dvou 3D modelů. Zadavatelem bylo vyžádáno jak porovnávání v jedné 3D scéně, tak i porovnávání v dvou synchronizovaných scénách. V této kapitole bylo tedy řešeno, jak rozšířit základní komponentu pro zobrazování 3D modelů tak, aby splňovala zadavatelovy požadavky. Navazující problematikou, řešenou v rámci kapitoly Návrh, je synchronizace 3D scén, odlišení aktivních a ne-aktivních modelů a kurzory.

Kapitola Implementace obsahuje popis vybraných klíčových funkcionalit. Naprostým základem je implementace vztahů a interakcí mezi *React* komponentami a třídami s logikou. Důležitou součástí Implementace je popis načítání 3D modelů. Tento popis obsahuje jak způsoby načítání 3D modelů různých formátů, tak i způsob načítání více 3D modelů nebo textur v jeden moment. Na základě návrhu synchronizace 3D scén a odlišení aktivních a ne-aktivních modelů bylo vhodné popsat jejich realizaci. Pro možný navazující vývoj bylo vhodné popsat kromě postupů při rozšiřování knihovny i způsob jejího sestavení a nasazení. Pro zájemce o použití knihovny byl vytvořen návod pro

instalaci a integraci.

Na závěr bylo vhodné knihovnu otestovat. Proto byla knihovna podrobena uživatelským i akceptačním testům. Za účelem uživatelského testování byla vytvořena aplikace, která demonstruje možnosti výsledné knihovny. Během uživatelského testování bylo nalezeno několik chyb a bylo získáno několik podnětů k dalšímu rozšíření knihovny. Pro tyto zjištění bylo krátce navrženo jejich řešení/implementace. Akceptační testování bylo splněno bez problémů.

Veškeré cíle vytyčené v rámci této diplomové práce byly splněny. Přínosem této práce může být zjednodušení vývoje podobných aplikací jako je *Lite průvodce* nebo *Webová aplikace pro schvalovací proces 3D modelů*. Knihovna je implementována takovým způsobem, aby byla co nejsnadněji rozšiřitelná. Díky tomu je další práce na ní, dle mého názoru, reálná.

Literatura

[1] Antoš, P.: *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*. Bakalářská práce, České vysoké učení technické v Praze, 2021.

[2] Ing. Petr Pauš, Ph.D. a Ing. Jiří Chludil: Věnná města českých královen. [online], [cit. 2023-5-04]. Dostupné z: `https://litepruvodce.kralovskavennamesta.cz/`

[3] Ing. Petr Pauš, Ph.D. a Ing. Jiří Chludil: Technická dokumentace a manuál. [online], [cit. 2023-3-15]. Dostupné z: `https://gitlab.fit.cvut.cz/vmck/web-viewer/-/blob/master/dokumentace/litepruvodce.docx`

[4] Facebook Inc.: React. [online], [cit. 2021-3-26]. Dostupné z: `https://reactjs.org/`

[5] Facebook Inc.: Components and props. [online], [cit. 2021-3-26]. Dostupné z: `https://reactjs.org/docs/components-and-props.html`

[6] Facebook Inc.: State: A Component's Memory. [online], [cit. 2023-2-28]. Dostupné z: `https://beta.reactjs.org/learn/state-a-components-memory`

[7] Mozilla Corporation: Classes. [online], [cit. 2023-2-28]. Dostupné z: `https://www.x3dom.org/wp-content/uploads/2009/10/x3dom-fallback-Release-1.2.png`

[8] Facebook Inc.: React.Component. [online], [cit. 2023-2-28]. Dostupné z: `https://reactjs.org/docs/react-component.html`

[9] deepak710agarwal: Differences between Functional Components and Class Components in React. [online], [cit. 2023-2-28]. Dostupné z: `https://www.geeksforgeeks.org/differences-between-functional-components-and-class-components-in-react/`

[10] Facebook Inc.: Hooks at a Glance. [online], [cit. 2023-2-28]. Dostupné z: https://reactjs.org/docs/hooks-overview.html

[11] Facebook Inc.: What is the virtual DOM. [online], [cit. 2021-3-26]. Dostupné z: https://reactjs.org/docs/faq-internals.html

[12] Facebook Inc.: Rendering elements. [online], [cit. 2021-3-26]. Dostupné z: https://reactjs.org/docs/rendering-elements.html

[13] Mary Njeri: Getting Started With React Cosmos. [online], [cit. 2023-3-5]. Dostupné z: https://www.section.io/engineering-education/getting-started-with-react-cosmos/

[14] Three.js Authors: Fundamentals. [online], [cit. 2023-3-1]. Dostupné z: https://threejs.org/manual/en/fundamentals

[15] Sean Bradley: Scene, Camera and Renderer. [online], [cit. 2023-3-1]. Dostupné z: https://sbcode.net/threejs/scene-camera-renderer/

[16] Draceane: Komolý jehlan. [online], [cit. 2023-3-1]. Dostupné z: https://cs.wikipedia.org/wiki/Komol%C3%BD_jehlan

[17] Sergiusz & Dora from IntexSoft: Introduction to 3D: Three.js basics. [online], [cit. 2023-3-1]. Dostupné z: https://intexsoft.com/blog/introduction-to-3d-three-js-basics/

[18] Nvidia Corporation: Ray Tracing. [online], [cit. 2023-3-10]. Dostupné z: https://developer.nvidia.com/discover/ray-tracing

[19] Moonreach: Ray casting. [online], [cit. 2023-3-10]. Dostupné z: https://en.wikipedia.org/wiki/Ray_casting

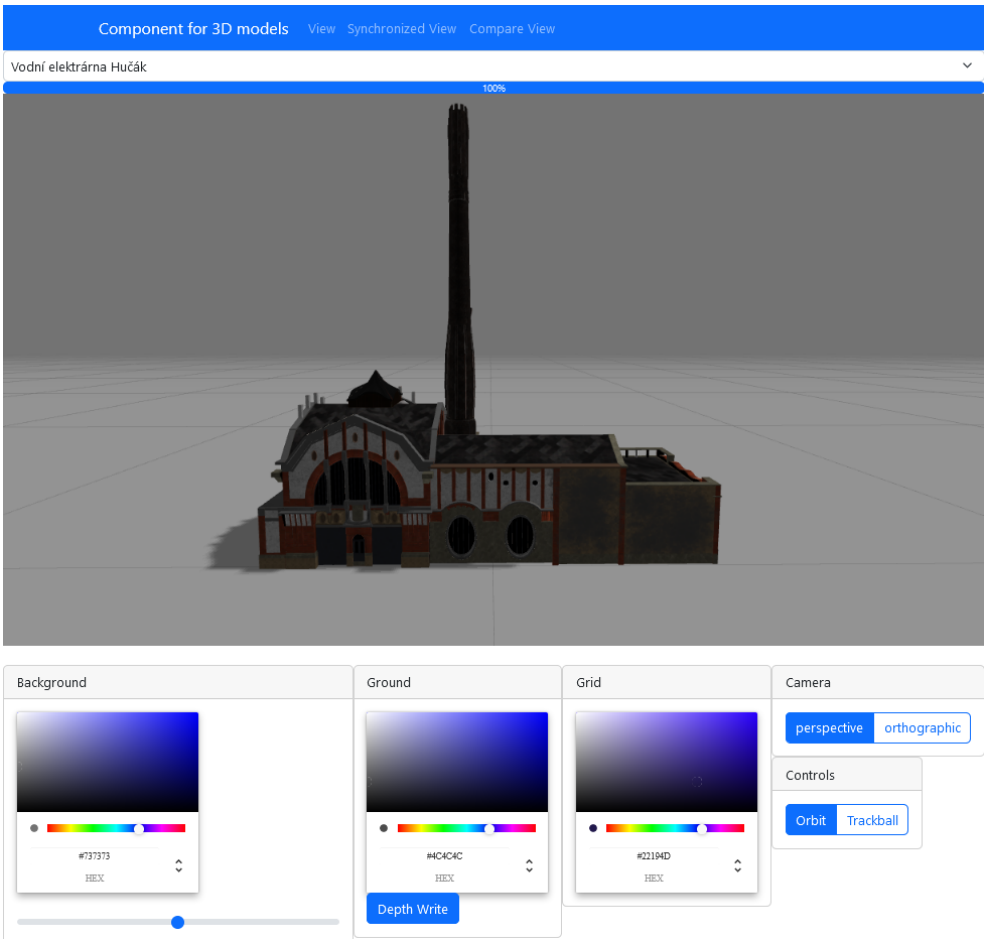
[20] Joey de Vries: Camera. [online], [cit. 2023-4-04]. Dostupné z: https://learnopengl.com/Getting-started/Camera

[21] Wisdood: Z-buffering. [online], [cit. 2023-3-28]. Dostupné z: https://en.wikipedia.org/wiki/Z-buffering

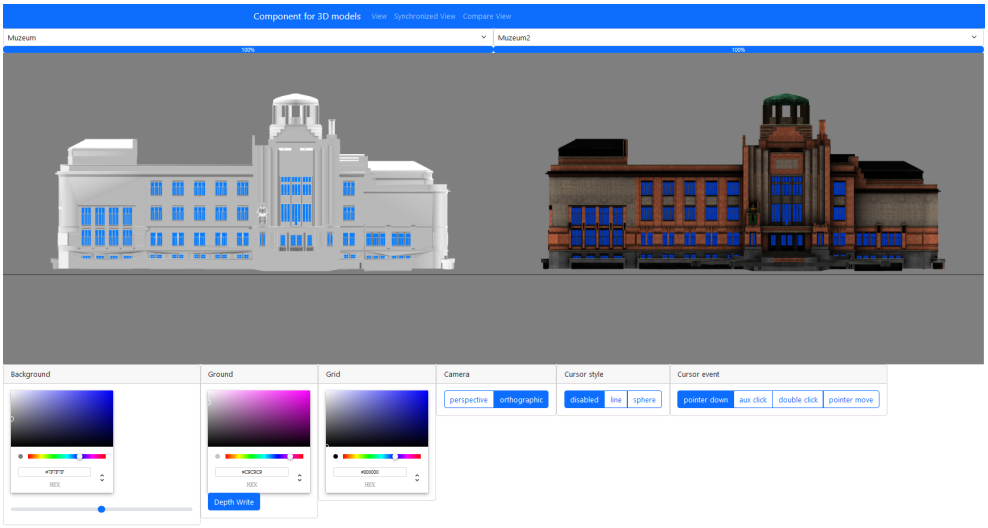
[22] Tim Mouskhelichvili: How To Call A Child Function From A Parent Component In React? [online], [cit. 2023-5-03]. Dostupné z: https://timmousk.com/blog/react-call-function-in-child-component/

Příloha A

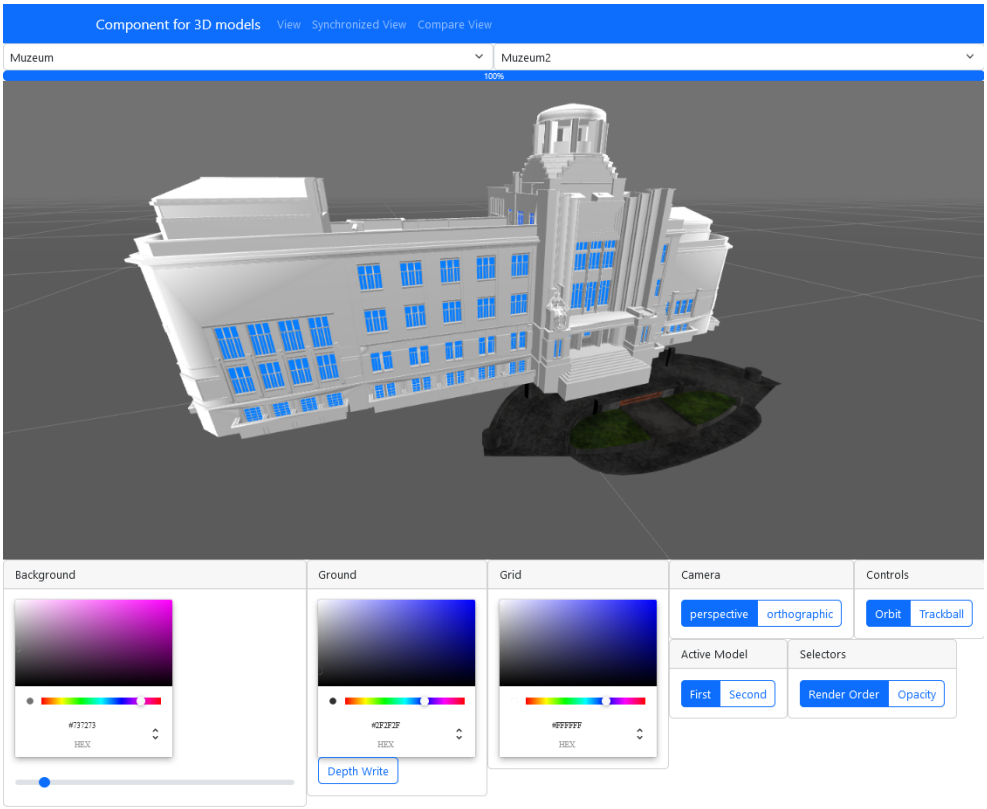
Testovací aplikace



Obrázek A.1: TA - ModelView



Obrázek A.2: TA - CompareViews



Obrázek A.3: TA - ModelCompare

Modely byly vytvořeny za finanční podpory Ministerstva kultury České republiky v rámci projektu NAKI II. č. DG18P02OVV015 s názvem Věnná města českých královen. (Živá součást historického vědomí a její podpora nástroji historické geografie, virtuální reality a kyberprostoru.)[2]

Příloha B

Seznam použitých zkratek

API Application Programming Interface

AR Augmented Reality

CSS Cascading Style Sheets

ES6 ECMAScript 6

HTML Hypertext Markup Language

npm Node.js package manager

PC Personal Computer

TDD Test-Driven Development

UI User Interface

URL Uniform Resource Locator

VR Virtual Reality

WebGL Web Graphics Library

Příloha C

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `tex` ..... zdrojové kódy textu práce
  - `img` ..... obrázky použité v textu
  - `DP_Antoš_Pavel_2023.tex` ..... text práce ve formátu LaTeX
- `DP_Antoš_Pavel_2023.pdf` ..... text práce ve formátu PDF