

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra elektrotechniky

Vizuální nástroj pro zpracování LIDARových dat

Visual Tool for LIDAR Data Processing

2020

Petr Vychodil

Zadání bakalářské práce

Student:

Petr Vychodil

Studijní program:

B2647 Informační a komunikační technologie

Studijní obor:

2612R025 Informatika a výpočetní technika

Téma:

Vizuální nástroj pro zpracování LIDARových dat
Visual Tool for LIDAR Data Processing

Jazyk vypracování:

čeština

Zásady pro vypracování:

Cílem práce je implementace nástroje pro označování objektů v LIDARových a obrazových datech. Takto označená data jsou pak vstupem pro analýzu obrazu, např. v prostředí autonomní jízdy, v úlohách detekce a klasifikace objektů v okolí vozidla.

Ve své práci proveděte:

1. Nastudujte dostupné frameworky pro implementaci požadovaného nástroje ve webovém prostředí.
2. Seznamte se s formátem ukládání LIDARových dat.
2. Naimplementujte zadanou úlohu ve webovém prostředí, která bude uživateli zobrazovat LIDARová a obrazová data.
3. Součástí aplikace bude i serverová část, která bude uládat výsledky pro pozdější zpracování.
4. Svou implementaci náležitě zdokumentujte a zhodnotěte.

Seznam doporučené odborné literatury:

- [1] Scott Domes: Progressive Web Apps with React: Create lightning fast web apps with native power using React and Firebase, ISBN-13: 978-1788297554, 2017
[2] Tony Parisi: Learning Virtual Reality: Developing Immersive Experiences and Applications for Desktop, Web, and Mobile 1st Edition, O'Reilly Media, ISBN-13: 978-1491922835, 2015

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Gaura, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019


doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry




prof. Ing. Pavel Brandstetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019

..........

Súhlasím so zverejnením tejto bakalárskej práce podľa požiadaviek čl . 26, odst. 9 Študijného a skúšobného poriadku pre štúdium v bakalárskych programoch VŠB-TU Ostrava.

V Ostrave 30. dubna 2019

..........

Chci poděkovat svému vedoucímu bakalářské práce Ing. Janu Gaurovi, Ph.D. za jeho vedení, konzultace a odbornou pomoc. Dále bych chtěl poděkovat všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tato bakalářská práce se zabývá implementací nástroje pro označování objektů a obrazových dat ve webovém prostředí. Označená data se stanou vstupem pro budoucí analýzu.

Klíčová slova: LIDAR, WebGL, bakalářská práce

Abstract

This bachelor thesis deals with implementation of tool for object selection and data visualization in web browser. Selected data will become input data for future analyzation.

Key Words: LIDAR, WebGL, bachelor thesis

Obsah

Seznam použitých zkratek a symbolů	9
Seznam obrázků	10
Seznam tabulek	11
Seznam výpisů zdrojového kódu	12
1 Úvod	13
2 LIDAR	14
2.1 Princip fungování LIDARu	14
2.2 Model VLP-16	14
2.3 Odpalovací sekvence	15
2.4 Režimy vrácení dat	15
2.5 Vertikální úhly a kanály	16
2.6 Mrak bodů	16
2.7 Výstupní data	16
2.8 Předzpracování dat	18
3 Vizualizace	20
3.1 Výběr technologií	20
3.2 Alternativní knihovny	20
3.3 Základy knihovny Three.js	21
3.4 Vytváření objektů v Three.js	21
3.5 Alternativní řešení	22
3.6 Princip fungování kamery	23
4 Implementace	25
4.1 REST API	25
4.2 Datový model	25
4.3 Scéna a kamera	26
4.4 Vizualizace LIDARových dat	28
4.5 Výběr prostoru	30
4.6 Zachycení obrazu selekce	34
4.7 Export vybraných bodů z selekce	37
4.8 Pohyb selekcí	39
5 Závěr	41

Seznam použitých zkratek a symbolů

- | | |
|--------|-------------------------------|
| LIDAR | – Light Detection And Ranging |
| WebGL | – Web Graphics Library |
| OpenGL | – Open Graphics Library |
| HTML | – Hypertext Markup Language |
| NPM | – Node.js package manager |
| XML | – Extensible Markup Language |
| XHR | – XML Http Request |
| HTTP | – Hypertext Transfer Protocol |
| FOV | – Field Of View |
| GPU | – Graphics processing unit |

Seznam obrázků

1	Zobrazení dat z modelu VLP-16 pomocí softwaru VeloView.	13
2	Paket modelu VLP-16 [2].	17
3	Paket modelu VLP-16 v režimu vrácení dvou [2].	18
4	Rozdíly v počtu segmentů	22
5	Perspektivní projekce kamery [3].	23
6	Datový model informačního systému.	26
7	Výsledná vizualizace.	30
8	První bod a druhý bod selekce.	33
9	Promítnutí bodů s již vytvořeným obrysem.	34
10	Ukázka kamerového frustumu před projekcí [21].	36
11	Ukázka kamerového frustumu po projekci [21].	36
12	Výsledný obraz po projekci [21].	36

Seznam tabulek

1	Mapa kanálů modelu VLP-16 i s vertikálními úhly.	16
---	--	----

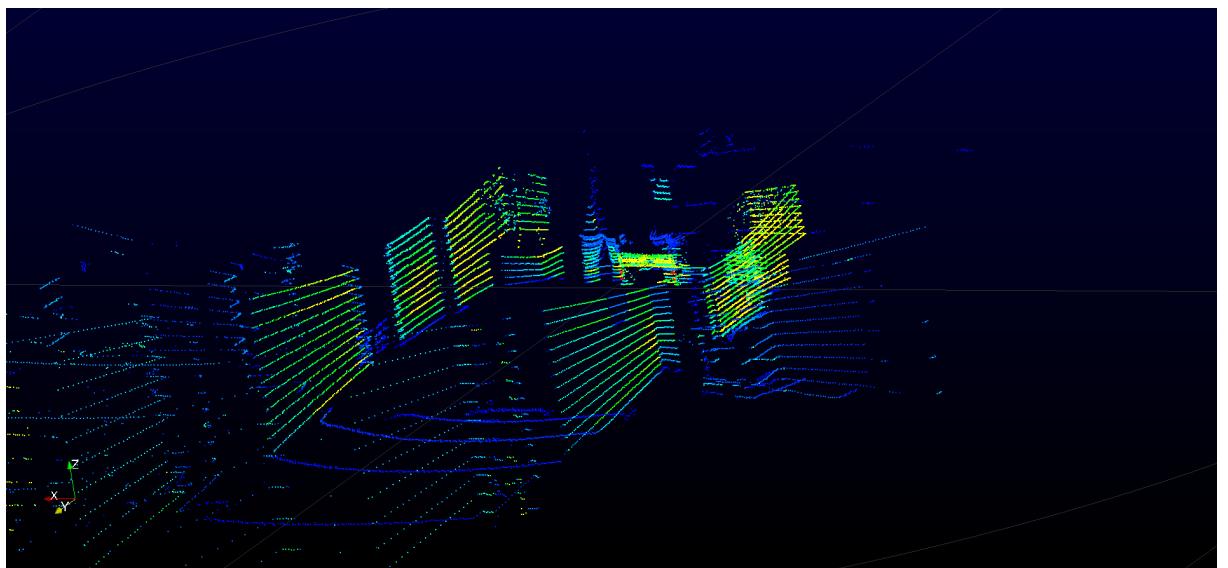
Seznam výpisů zdrojového kódu

1	Python script pro transformaci pcap souborů na JSON soubory	19
2	Vytvoření koule za pomocí knihovny Three.js	21
3	Vytvoření scény s kamerou a rendererem.	27
4	Metoda pro získání LIDARových dat ze serveru.	28
5	Metoda pro vykreslení LIDARových dat.	29
6	Projekce myši do 3D prostoru.	30
7	Zvětšení délky vektoru.	31
8	Promítnutí druhého bodu obrysu selekce.	31
9	Vytvoření obrysu selekce.	32
10	Použití THREE.Shape k vytvoření obdélníku.	32
11	Vytváření kamery pro sledování selekce.	36
12	Získání snímku z kamery.	37
13	Získání pozice vrcholu obrysu po provedení rotace.	37
14	Získání všech bodů v selekcii.	38
15	Získání obrysu na základě události myši.	39
16	Pohyb obrysu ve scéně.	39
17	Zrušení pohybu a zaměření kamery pro zaslání obrázových dat.	40

1 Úvod

Lidar funguje na stejném principu jako radar nebo sonar. Jen místo používání rádiových nebo zvukových vln používá světelné vlny. Využívá potenciál laseru na výpočet vzdálenosti od vzniku paprsku až po objekt, který je laserem zasažen. Objekt zasažen laserem odráží část paprsku laseru zpět ke zdroji.

V dnešní době, kdy se fyzické výpočetní jednotky zmenšují a zároveň se zvětšuje výkon a jejich výpočetní síla, roste využití LIDARu hlavně v automobilovém odvětví. Dynamické mapování terénu může pomoci různými způsoby. Například reagovat na akce, na které by člověk nestihl svými smysly zareagovat. Tato technologie je klíčovou složkou v úplném nahrazení řízení auta člověkem. Pro představu výsledných dat modelu VLP-16 1.



Obrázek 1: Zobrazení dat z modelu VLP-16 pomocí softwaru VeloView.

Cílem tohoto projektu je vytvořit nástroj ve webovém rozhraní, který bude sloužit pro označování objektů v LIDARových i obrazových datech. Tato data budou sloužit jako vstup pro budoucí analýzu dat. Projekt bude obsahovat i serverovou část pro implementaci přijímání a odesílání LIDARových dat a obrázků prostředí. Serverová část nejdříve pošle data, která ve webovém prostředí zobrazí. Na tomto prostředí bude naimplementované označování objektů, kde uživatel označí objekt, a pak všechna označená data posílá na serverovou část pro uložení. Na serverovou část se bude posílat i snímek bez LIDARových dat. Data na serveru budou použita pro následnou analýzu obrazu a vizualizační nástroj bude sloužit i k opětovnému přehrání označených dat. Snímek bude sloužit pro lepší představu o tom, o jaký objekt se jedná, protože u LIDARových dat to nemusí být vždy přesné.

2 LIDAR

V této kapitole vysvětlím princip fungování LIDARu, ukládání dat a přesněji představím použitý model. Také vysvětlím pojmy, které budu používat v dalších kapitolách.

2.1 Princip fungování LIDARu

V úvodu jsem Vám krátce představil, co je to LIDAR a jak fuguje. Nyní Vám ukážu fyzické fungování modelu VLP-16, se kterým jsem pracoval. Základní složkou, kterou LIDAR využívá a počítá s ní, je světlo a rychlosť světla. LIDAR střílí světelné paprsky, které rychlosťí světla letí k cíli. Část tohoto světelného paprsku se po nárazu s objektem odraží, a putuje zpět stejnou rychlosťí ke zdroji paprsku, kde paprsek zachytí senzor. Po dobu celého tohoto procesu je zaznamenáván čas, tzv. čas letu. Z těchto informací dostávám rovnici (1), která popisuje, jak LIDAR počítá vzdálenost bodu. Zkratka s značí vzdálenost, c rychlosť světla a t čas.

$$s = \frac{(c \cdot t)}{2} \quad (1)$$

Pro představu, některá zařízení dokáží vystřelit až 150 000 paprsků za sekundu. Vlnová délka se liší v závislosti na vzdálenosti a přesnosti. Pohybuje se v rozmezí 600-1000 nm. Čím nižší vlnová délka, tím je světlo méně pohlcováno vodou, ale za cenu vzdálenosti, na které je paprsek přesný. Z těchto poznatků je zřejmé, že stojí za to zvážit vlnovou délku v závislosti na využití. Každý paprsek má rozptyl 3 mrad. Tímto rozptylem se zároveň snižuje přesnost bodu na větší vzdálenost. Každý LIDAR má určitý počet kanálů, v rozmezí 8-128 kanálů laserových paprsků. S rostoucím počtem kanálů roste i rozlišení a větší přesnost mapovaného prostředí. Toto rozlišení je klíčová vlastnost, která je nezbytná v samořídících autech. Důvodem je víc informací o aktuálním prostředí, rychlejší a častější sběr informací. Samořídící auto, které je vybavené třicetidvou kanálovým LIDAREm může jet bezpečně 56 km h^{-1} . Auto s 128-mi kanálovým LIDAREm může jet rychlosťí až 104 km h^{-1} . S rostoucí rychlosťí potřebují mít senzory více kanálů.

2.2 Model VLP-16

Model VLP-16 byl využit při sběru dat pro mé účely. Zde je pář z jeho typických vlastností. Tento model má 16 kanálů. Sběr jedné kompletnej odpalovací sekvence trvá $55.296 \mu\text{s}$. Laser tohoto modelu pracuje na vlnové délce 905 nm. Mapa kanálů a úhlů, pod kterými operují paprsky laseru zmíněné v tabulce 1. Díky nízké vlnové délce laseru má nízkou spotřebu energie. Funkční vzdálenost je až 100 metrů. LIDAR má horizontální zorné pole 360° a vertikální zorné pole $\pm 30^\circ$. Těmito vlastnostmi dokáže model získat dostatek kvalitních dat o objektech okolo sebe. Navíc váží pouze 830 gramů s velmi nízkou spotřebou elektrické energie. V tomto modelu je zaintegrován webový server pro jednoduchý monitoring a konfiguraci od výrobce. Tento model je specializován pro přenos dat po síti. Ethernetové připojení dokáže posílat data rychlosťí až

100 MBps. Je schopen ukládat záznamy paketů posílané přes ethernet. Jaká data a v jakém formátu jsou obsažena v jednotlivých UDP paketech Vám ukážu v kapitole 2.7.

2.3 Odpalovací sekvence

Odpalovací sekvencí se označuje vždy čas, kdy byly vyslány paprsky laseru do všech možných úhlů v jednom azimutu. Jaké úhly a jaké množství paprsků přesně má LIDAR se liší model od modelu. Například model VLP-16 má 16 úhlů, do kterých se paprsky vysílají.

2.4 Režimy vrácení dat

LIDAR je schopný pracovat ve dvou režimech. Nejdříve budu muset vysvětlit, jaká data LIDAR může vrátit. Při putování paprsku je možné, že paprsek světla narazí do prvního objektu, tomuto bodu dotyku budu říkat bod A. Tento objekt bude průhledný, a protože je průhledný, tak nepohltí a neodrazí celý paprsek zpět. Z prvního střetu je část paprsku odražena zpět k LIDARu. Paprsek po průchodu prvním objektem pokračuje dál a narazí na jiný objekt, například zed. Místu střetu se zdí budu říkat bod B. Po tomto střetu se zbytek světelného paprsku odrazí zpět k senzoru. Nyní mám bod A a bod B, které LIDAR zaznamenal. Záleží na nastavení režimu, který rozhodne, zda uloží oba body nebo jen jeden. Bod A se vrátí k senzoru jako první. Díky průhlednosti materiálu se část paprsku vrátí a část projde dál. Bod, který LIDAR zaznamenal se vrátí jako nejsilnější vrácený bod, v angličtině nazvaný jako „strongest return“. Zbytek paprsku, který se odrazil od zdi, v mé případě bod B, je nazýván poslední vrácený bod, v původním znění jako „last return“. Toto jsou dva typy bodů, které je LIDAR schopen vrátit.

- Návrat dvou (Dual return)
- Návrat jednoho (Single return)

Režim vrácení dat si uživatel může při konfiguraci LIDARu nastavit. Režim návrat dvou zařizuje vrácení nejsilnějšího i posledního vráceného bodu. Tato konfigurace zaručí přesnější obraz prostředí. Na druhou stranu se pro jeden kompletní obraz posílají 2 rámce s daty. Tímto je zapříčiněno, že doba celkového obrazu se zdvojnásobí. Tento režim se používá převážně k mapování terénů. Zaznamená vegetaci (druhý nejsilnější bod) a samotný terén (nejsilnější bod).

Návrat jednoho, jak už název vypovídá, ukládá pouze jeden druh vrácených dat. Uživatel si při konfiguraci LIDARu může vybrat, jestli ho zajímá poslední vrácený nebo nejsilnější vrácený bod. Tento režim je dvojnásobně rychlejší, než režim návratu dvou a používá se v automobilovém průmyslu, kdy je potřeba rychlosť. Proto se většinou nastavuje tento režim s nastavením vrátit nejsilnějšího. Nejčastěji je potřeba vrátit pevný objekt, aby systém mohl nasimulovat překážky, na které během provozu může narazit. Pevný objekt bývá v převážné většině nejsilnější vrácený bod.

2.5 Vertikální úhly a kanály

V této části budu tyto úhly ukazovat na modelu VLP-16, ale princip zůstane stejný. Jediný rozdíl bude v počtu laserů a velikosti úhlů. Model VLP-16 má šestnáct kanálů, kde každý kanál má svůj vlastní paprsek laseru. Každý paprsek v laseru má svůj vlastní úhel. Data uložená v datovém bloku jsou indexována podle čísel kanálů.

Tabulka 1: Mapa kanálů modelu VLP-16 i s vertikálními úhly.

Kanály	Vertikální úhly
0	-15°
1	1°
2	-13°
3	-3°
4	-11°
5	5°
6	-9°
7	7°
8	-7°
9	9°
10	-5°
11	11°
12	-3°
13	13°
14	-1°
15	15°

Vertikální úhly jsou určeny vzhledem k pomyslné vodorovné ploše, procházející středem LIDARového senzoru. Každý vrácený bod v datovém bloku je uložen pod číslem kanálu.

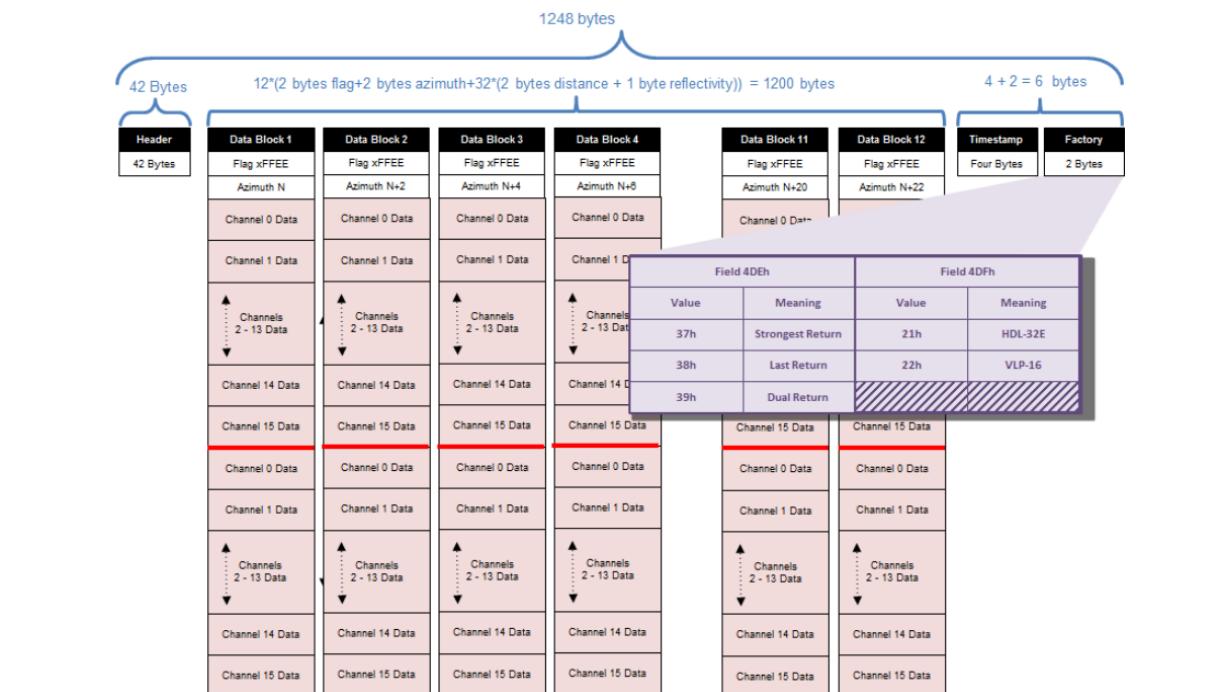
2.6 Mrak bodů

V angličtině známý jako „Point cloud“. Jak název již napovídá, je to množina bodů v prostoru. Mrak bodů bývá vytvořen 3D skenery jako je LIDAR. Tento pojem se používá při vizualizaci těchto dat, protože když zobrazím výsledek, tak vzhledem připomíná oblak bodů. Tato práce je o vizualizaci těchto bodů v libovolném webovém frameworku a následné selekci.

2.7 Výstupní data

Každý model ukládá data v různých formátech. Data jsou uložená v paketu, který je složen z jednoho až n datových bloků. Každý z těchto datových bloků je složen z jedné až n dat z odpalovacích sekvencí, které jsou uložené pod číslem kanálu. Tato část bude specifitěji zaměřena na model VLP-16. Celý paket má velikost 1248 bytů. Těchto 1248 bytů je rozděleno do hlavičky paketu, datových bloků, timestampu a factory. Hlavička paketu má velikost 42 bytů, je důležitá pro internetovou komunikaci pomocí TCP/IP protokolu. Následuje 12 datových bloků. Každý z

těchto datových bloků má velikost 100 byteů. Skládá se z označujících dvou byteů, které označují začátek nového datového bloku, dvou byteů velikosti azimutu a z 96 byteů čistých dat. Tyto data mají v sobě dvě odpalovací frekvence, kde jedna odpalovací frekvence má 16 kanálů. Dohromady je 32 kanálů, kde každý kanál se skládá ze 2 byteů vzdálenosti a jednoho bytu reflektivity. Celkem to bude $32(2+1)$, to je 96 byteů plus 4 byty. Dohromady je 100 byteů, které dávají jeden kompletní datový blok. Azimut, označený na začátku datového bloku, je pouze pro první odpalovací frekvenci. Azimut pro druhou odpalovací frekvenci není explicitně napsán v datovém bloku, ale může být vypočten přičtením jedničky. Stále zbývá posledních 6 byteů do 1248 byteů. V prvních 4 bytech je uložené časové razítka, které je synchronizováno s GPS systémem. A v posledních dvou bytech je uložené tovární nastavení. V prvním z těchto byteů je uložen režim vrácení dat, již zmíněno v kapitole 2.4, a v posledním je uloženo označení modelu. Hodnoty a význam těchto byteů ukážu na obrázku 2.

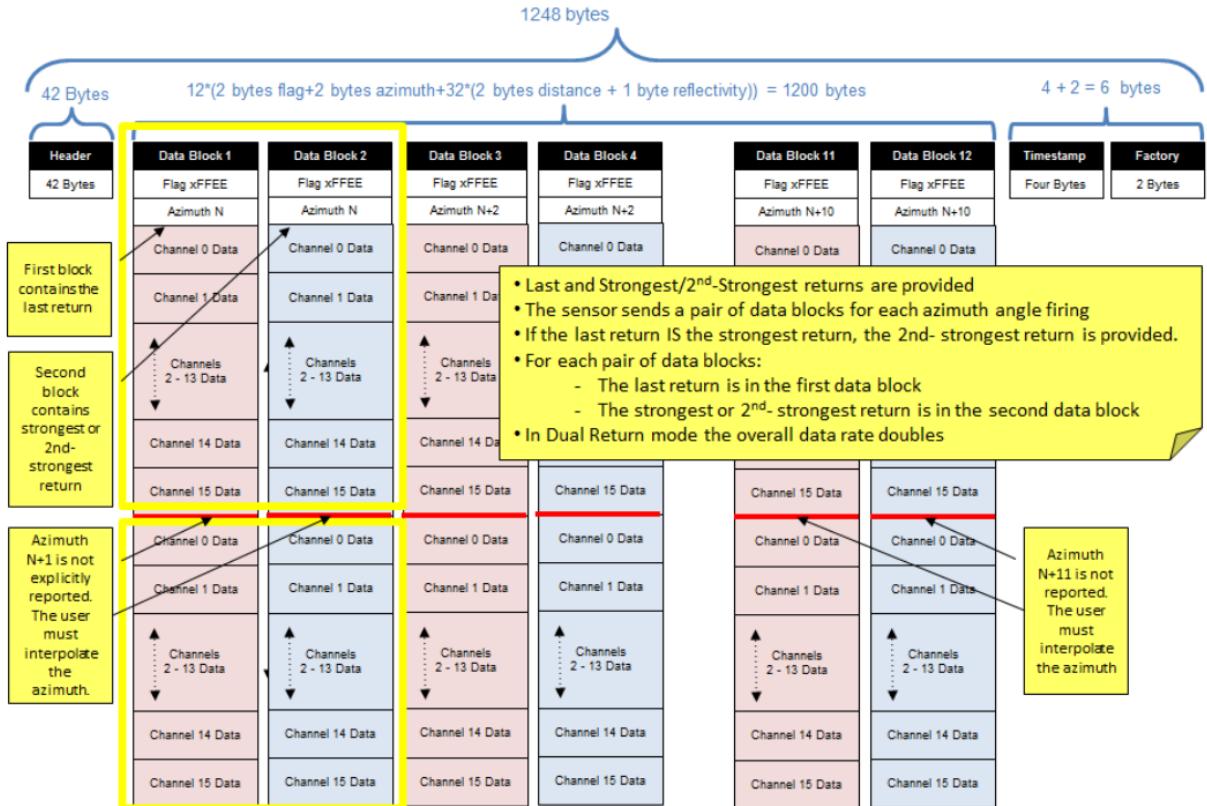


Obrázek 2: Paket modelu VLP-16 [2].

Značka **xFFEE** na začátku každého datového bloku značí zahájení nového datového bloku. Tato hodnota je určena výrobcem a je neměnná. **Factory**, tovární nastavení na konci paketu, s hodnotou 37 hexadecimálně označuje, že LIDAR je nastaven v režimu vrácení jednoho s nastavením vrat nejsilnější bod. Hodnota 38 hexadecimálně označuje také režim vrácení jednoho, ale vrací poslední vrácený bod. Poslední možnost s hodnotou 39 hexadecimálně znamená režim vrácení dvou. Poslední byte označuje model, ze kterého je tento paket.

Obsah paketu se liší v závislosti na režimu vrácení dat. Při režimu vrácení dvou se posílají stejné pakety s rozdílem, že se posílají párové datové bloky. Párovým blokem jsou myšleny dva

datové bloky paketu, které obsahují odpalovací sekvence se stejným azimutem. Na obrázku 3 jsou vidět dva databloky. Každý z nich obsahuje dvě odpalovací sekvence. Tyto dva databloky mají stejný azimut. První i druhý datablok obsahuje stejný azimut odpalovacích sekvencí, ale liší se typem vrácených dat. První z páru vždy obsahuje hodnotu posledního vráceného bodu, druhý datový blok obsahuje hodnotu nejsilnějšího vráceného bodu.



Obrázek 3: Paket modelu VLP-16 v režimu vrácení dvou [2].

2.8 Předzpracování dat

Předzpracování dat závisí na tom, zda chci provádět přenos dat v reálném čase ke zpracování, nebo ukládat pakety dat do souborů s příponou `.pcap`. Přenos dat v reálném čase se využívá v autonomních automobilech, kde automobil musí reagovat na překážky. Já nebudu potřebovat komunikaci a vizualizaci v reálném čase. Účel budovaného systému bude hlavně k vizualizaci LIDARových dat a následnému označování určitých částí mraku bodů, který bude vracet na serverovou část k uložení snímku vybrané části i všech LIDARových bodů z označené oblasti. Data proto nezískávám z aktivní komunikace LIDARu se zařízením, ale z `pcap` souborů vytvořených LIDARem. `Pcap` soubor je záznam komunikace přes TCP/IP protokol. Tento soubor je složen z mnoha paketů 2.7. K vizualizaci dat z `pcap` souboru je nutné nejdříve extrahovat informace, které jsou uvnitř jednotlivých paketů. Data jsem se rozhodl transformovat rovnou do kartézských souřadnic, abych předešel zbytečným a velmi komplikovaným výpočtům na straně

uživatele. K tomuto účelu jsem použil Python konzoli v programu VeloView. Hlavním důvodem využití této konzole bylo přístupné Python rozhraní pro práci s pcap soubory. Zmíněné rozhraní používá knihovnu napsanou v jazyce C++, aby zpracování bylo co nejrychlejší. Tato knihovna je specificky napsána pro práci s LIDARem a pcap soubory, proto jsem vytvořil jednoduchý script v jazyce Python, který využívá již zmíněné rozhraní a přetrasformuje pcap soubor na neznámý počet JSON souborů. Každý JSON soubor obsahuje jeden kompletní mrak bodů.

```
1 import json
2
3 try:
4     fileIndex = 0
5     while True:
6         vv.gotoNext()
7         cloudinfo = vv.getReader().GetClientSideObject().GetOutput()
8         points = cloudinfo.GetPoints()
9         data = {}
10        for i in range(0, cloudinfo.GetNumberOfPoints()):
11            data[i] = points.GetPoint(i)
12        path = 'C:\\\\Dev\\\\Lidar\\\\RawData\\\\data-standing'
13        fileName = "\\\\[LidarData_ " + str(fileIndex) + ".json"
14        with open(path + fileName, 'w') as outfile:
15            json.dump(data, outfile)
16        fileIndex += 1
17    except:
18        print("Loop ended or it threw exception.")
```

Výpis 1: Python script pro transformaci pcap souborů na JSON soubory.

Tento script spouštím v Python konzoli programu VeloView po načtení pcap souboru. Příkaz `vv.gotoNext()` nás přesune na další ze zmapovaných mraků bodů. V proměnné `cloudinfo` máme uložený aktuální snímek mraku bodů, který má uživateli zobrazen a všechna možná dostupná data, která byla v paketech uložena i s metadaty. Proměnná `points` vrátí pole všech bodů aktuálního mraku bodů. Každý tento vrácený bod obsahuje trojici čísel, která odpovídá kartézským souřadnicím v prostoru. Proměnná `data` je slovník, do kterého v cyklu ukládám potřebná data. Slovník je specifický tím, že se do něj ukládá klíč a na daný klíč je navázán hodnota, v mém případě klíč je číslo bodu a k tomu navazující hodnota z proměnné `points` na indexu klíče. Po ukončení všech iterací cyklu vytvořím soubor a pomocí externí knihovny `json`, kterou jsem na začátku scriptu importoval, vytvořím ze slovníku JSON řetězec a uložím ho do souboru. Každý soubor začíná prefixem `LidarData_X`, kde X je index aktuálního mraku bodů. Po spuštění tohoto scriptu mám všechna data, která potřebuji na vizualizaci mraku bodů a následnou práce s ním. Data následně uložím do databázového systému.

3 Vizualizace

Tato část je určena vizualizaci. V této kapitole Vám ukážu, jak jsem postupoval při vývoji systému. Od procesu výběru technologií a frameworků až po principy vykreslování. Ze zadání je zřejmé, že vizualizace dat musí probíhat v prohlížeči na straně uživatele a musí být zobrazená ve 3D. V serverové části budu mít možnost si uložit vybraná data, popřípadě obrázky.

3.1 Výběr technologií

Na straně uživatele používám programovací jazyk JavaScript. K těmto účelům bylo vytvořené JavaScriptové rozhraní pro použití WebGL, které využívá rozhraní OpenGL. OpenGL je abstraktní rozhraní pro práci s grafikou. Implementace OpenGL jsou vytvořeny pro všechny platformy a naprogramovány na rychlejších jazycích, pracují přímo s grafickou kartou pro vysokou výpočetní rychlosť. Knihovna, kterou jsem si vybral je Three.js. Three.js je knihovna, která detekuje verzi prohlížeče a použije nejnovější technologie a nejnovější implementace JavaScriptu, které jsou prohlížečem možné použít. Hlavními důvody, proč jsem použil tuto 3D knihovnu, byla její rozsáhlá a detailně zpracovaná dokumentace. V serverové části jsem se rozhodl použít programovací jazyk Java. Bude mít za úkol posílat mraky bodů na stranu uživatele a zpracovat, nebo uložit data odeslaná uživatelem. Pro vytvoření webové aplikace jsem využil knihovnu Spring Boot. Dalším rozšířením je knihovna Thymeleaf pro dynamické renderování HTML na straně serveru. Již zmíněné technologie jsou nedílnou součástí praktické části. Jako bundler statických prvků a minifikaci JavaScriptu jsem použil nástroj zvaný Webpack. Jeho výhodou je jednoduchá konfigurace a používání. Bundler spojí všechny potřebné použité javascript soubory do jednoho zminifikovaného souboru. Pro ukládání dat použiji objektově-relační databázový server PostgreSQL, který je zdarma a je připravený na vysoký počet požadavků.

3.2 Alternativní knihovny

Je mnoho alternativních knihoven, které jsem mohl použít. Ukáži Vám pář z nich. Za zmínku stojí Babylon.js. Tato knihovna se používá na vytváření her v prohlížeči. Kód z Babylon.js je možné exportovat do Unity 3D herního enginu. Zaměřuje se tedy více na kolize, události, rendering a vyhlazování. Oba dva jsou v tuto chvíli nejlepšími 3D JavaScriptovými knihovnami s open source licencí. Alternativou Webpack bundleru je Rollup. Konfigurací se velmi podobá Webpack bundleru. Je jen o zanedbatelnou dobu pomalejší. Na straně serveru jsem mohl použít mnoho různých alternativ, protože požadavek od serveru je odpovídat a přijímat žádosti o data a posílat je uživateli na jeho rozhraní, přijímat označená data z uživatelského rozhraní a ukládat je na straně serveru k budoucí analýze. Další možností byla webová aplikační knihovna Django s programovacím jazykem Python. Tato kombinace je velmi dobrá pro rychlý vývoj jednoduchých aplikací. Další alternativou bylo PHP s některým z frameworků, jako je například Laravel. Neměl jsem žádné specifické požadavky od systému, ve kterých bych mohl říct, že využiji určitý

programovací jazyk, protože by byl pro tuto problematiku lepší. Požadavky od serverové části by efektivně zvládlo mnoho jazyků. Nelze tedy z požadavků vybrat jen jeden jazyk a říct, že bude pro tento účel nejlepší. Pro databázový server jsem měl mnoho možností. Jedním z mnoha je MySQL. Tento databázový server je velmi rychlý i ve velkém množství čtecích operací. Je také úspornější na místo v paměti, ale za cenu problému s konkurencí. Pro náš systém jsem potřeboval databázový systém, který bude efektivně číst a psát větší množství dat.

3.3 Základy knihovny Three.js

Abych mohl pomocí knihovny Three.js cokoliv zobrazit, tak k tomu potřebuji tři nezbytné a základní věci, bez kterých bych se neobešel. Těmito třemi základními věcmi jsou scéna, kamera a renderer. Scéna je virtuální trojrozměrný prostor, do kterého přidávám objekty. Kamera zařizuje projekci objektů na 2D obrazovku, jaký typ projekce záleží na kameře. Poslední a nejdůležitější částí je renderer. Renderer zařizuje vykreslení scény z pohledu kamery, popřípadě více kamer. Stále mi tyto prvky nemají co zobrazit, nejdříve si musím vytvořit nějaké objekty, které mohu přidat do scény. Každý objekt musí mít pozici, na kterou ji přidávám do scény. Scéna v Three.js má souřadnicový systém pravotočivý. Když si vezmu pravou ruku a dám ji dlaní před sebe, tak osu x představuje palec, ukazováček směřující vzhůru je osa y a pokrčený prostředníček směřující k tělu je osa z.

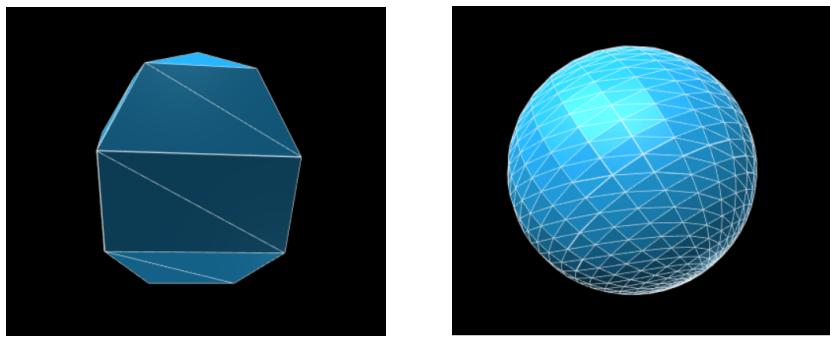
3.4 Vytváření objektů v Three.js

K vytvoření každého objektu potřebuji množinu vrcholů, stěn a materiálů ve WebGL. Three.js má již jednoduché rozhraní, které mi dovoluje vytvářet různé objekty, bez znalostí knihovny WebGL. Knihovná má již vyřešené vytváření základních 3D geometrických objektů, jako je krychle, koule, křivky a mnoho dalších. Zde je ukázka vytvoření koule o poloměru 0.005.

```
1 let geometry = new THREE.SphereGeometry(0.005, 5, 5)
2 let material = new THREE.MeshLambertMaterial({color: 0xffff00})
3 let sphere = new THREE.Mesh(geometry, material)
4 sphere.position.set(x, y, z)
5 scene.add(sphere)
```

Výpis 2: Vytvoření koule za pomocí knihovny Three.js

První potřebuji vrcholy a stěny. O to se postará `THREE.SphereGeometry`, který jako první parametr bere průměr koule, druhý parametr je počet segmentů na šířku a třetí počet segmentů na výšku. Čím více segmentů daná koule má, tím je přesnější a kulatější. Rozdíly jsou vidět na obrázku 4. Počty segmentů jsou shora i zdola omezeny. Maximum je 32 u výšky i šířky a minimum se liší. U výšky je minimum 2, a u šířky 3.



(a) Pět segmentů na výšku i šířku

(b) Třicetdva segmentů na výšku i šířku

Obrázek 4: Rozdíly v počtu segmentů

S narůstajícím počtem segmentů jsou nároky na GPU vyšší, protože musí vykreslit více vrcholů a hran koule. Proto jsem v projektu použil malý počet segmentů na výšku i šířku koulí 5. Když vezmu v potaz, že každé vykreslení mraku bodů má 25 000 takových koulí, tak budu chtít nároky na GPU co nejnižší. Dále potřebuji ještě materiál, který mi vyplní všechny stěny a vrcholy. Je hodně různých druhů materiálů. Každý materiál má své specifické vlastnosti, jak se zobrazuje ve světle. Tato vlastnost `THREE.MeshLambertMaterial` v kombinaci se světlem pomůže rozlišit jednotlivé koule (LIDARové body) od sebe. Kdybych použil materiál `THREE.Basic`, tak bych nerozeznal jednu kouli od druhé. Na tento materiál nepůsobí světlo, proto ať bych se podíval na jakoukoliv část koule, tak by měla konstantní odstín barvy materiálu. Koule by splývaly mezi sebou. Při vytváření nové instance materiálu musím přidat barvu do konstruktoru. `THREE.Mesh` si jako argument požaduje geometrii a materiál, který má vyplnit stěny dané geometrie. Mesh vytvoří výsledný objekt, v mém případě kouli. Tento objekt je možné vložit přímo do scény pro zobrazení koncovému uživateli.

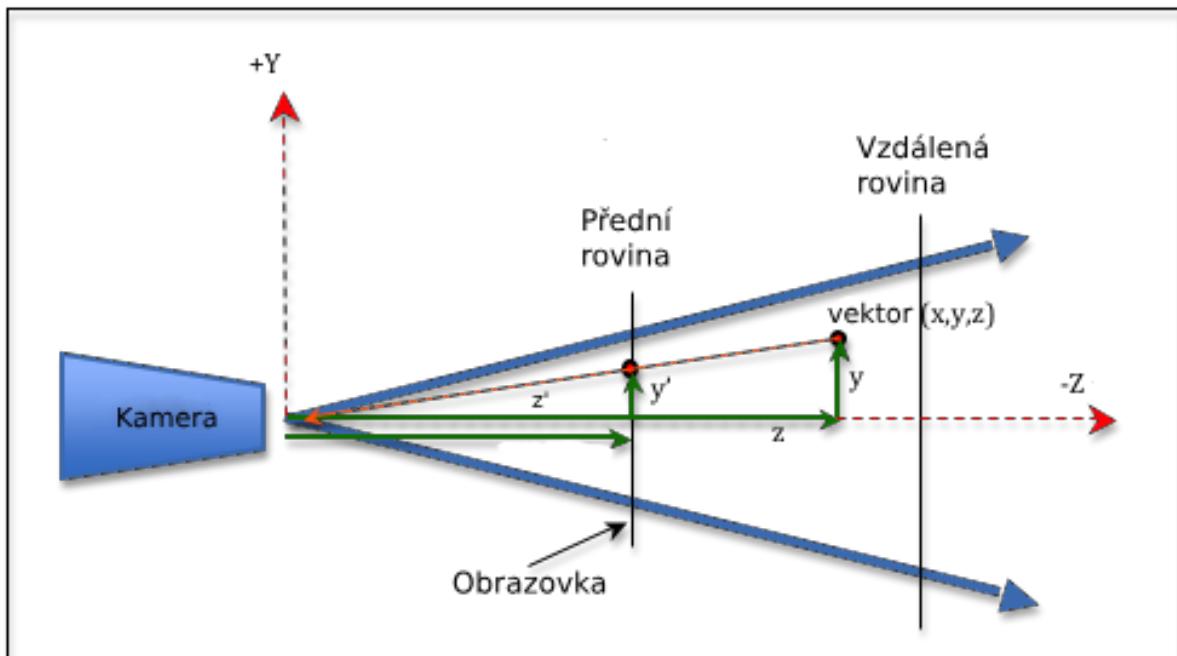
3.5 Alternativní řešení

Jedno z dalších možných řešení je program VeloView. Program VeloView je aplikace na stolní počítač, který slouží pouze k vizualizaci LIDARových dat z LIDARů od firmy Velodyne. Firma Velodyne se specializuje převážně na zlepšení a zdokonalení LIDARů zaměřených na automobilovou část průmyslu. Veloview, narozdíl od mého vytvářeného systému, není ve webovém rozhraní dostupný odkudkoliv a z jakéhokoliv systému. Dalším rozdílem je, že zobrazuje čistě LIDARová data a nedovoluje přidat pozadí, abych si mohl i graficky představit prostředí, v jakém se auto pohybuje. Označování objektů a následný export dat je v obou již zmínovaných LIDARových nástrojích. Další alternativou je Python knihovna s názvem Point Processing Toolkit, ve zkratce PPTK. Tato knihovna dokáže vizualizovat body v dvojrozměrném, ale i v trojrozměrném prostoru. Dále podporuje selekci bodů v prostoru přes uživatelské rozhraní, stejně jako program VeloView. Avšak narozdíl od programu VeloView, PPTK data pouze vizualizuje, proto potře-

buje data určující pozici v prostoru. VeloView používá data přímo z výstupních souborů, které generuje LIDAR. Stejně jako VeloView je možné tento program použít pouze na počítačích a bez možného přidání snímku do pozadí projekce.

3.6 Princip fungování kamery

Kamera vytváří perspektivní projekci na scénu. Než vysvětlím, jak funguje perspektivní projekce, nejdříve musím objasnit, na jakém principu pracuje kamera. Perspektivní kamera zachycuje scénu pohledem oka. Zóna zachycení kamery je specifikována jako frustum. Frustum je složeno z šesti rovin, které dělí prostor. Uvnitř frustumu je zachycena část scény, kterou chci kamerou zobrazit. Frustum se skládá z šesti rovin: horní, spodní, nalevo, napravo a dvě roviny vpřed a v dálce. Úhlu mezi horní a dolní rovinou se říká horizontální field of view, zkráceně FOV. Tento úhel bývá v rozmezí $30^\circ - 60^\circ$. Přední rovina ohraničená dalšími rovinami je obrazovka, do této roviny se provede projekce všeho, co je uvnitř frustumu.



Obrázek 5: Perspektivní projekce kamery [3].

Jak je vidět na obrázku, když chci promítnout bod na přední rovinu (obrazovku), tak vektor ze středu kamery k vrcholu objektu musí protnout přední rovinu. Bod střetu tohoto vektoru s rovinou je perspektivní projekce tohoto bodu na naši obrazovku. Takto je to provedeno se všemi hranami a vrcholy všech objektů. Tímto promítnutím na plátno zůstává zachovaný poměr $\frac{y'}{z'} = \frac{y}{z}$.

Pro zjišťování obrysů ve scéně za kurzorem budu používat Ray-tracing algoritmus. Ray-tracing, jak už z názvu vyplývá, bude sledovat paprsek, který vychází ze středu kamery přes

promítací rovinu a zachytí všechny stěny geometrie po jeho trase. Při získání geometrie, kterou protnul, si zjistí, o jaký objekt se jedná. Ukáži to na obrázku 5. Paprsek vznikne ve středu kamery a protíná určité místo promítací roviny. Tímto se snažím najít jakýkoliv objekt, který za tímto místem je. Používám implementaci tohoto algoritmu ve třídě **Raycaster**.

4 Implementace

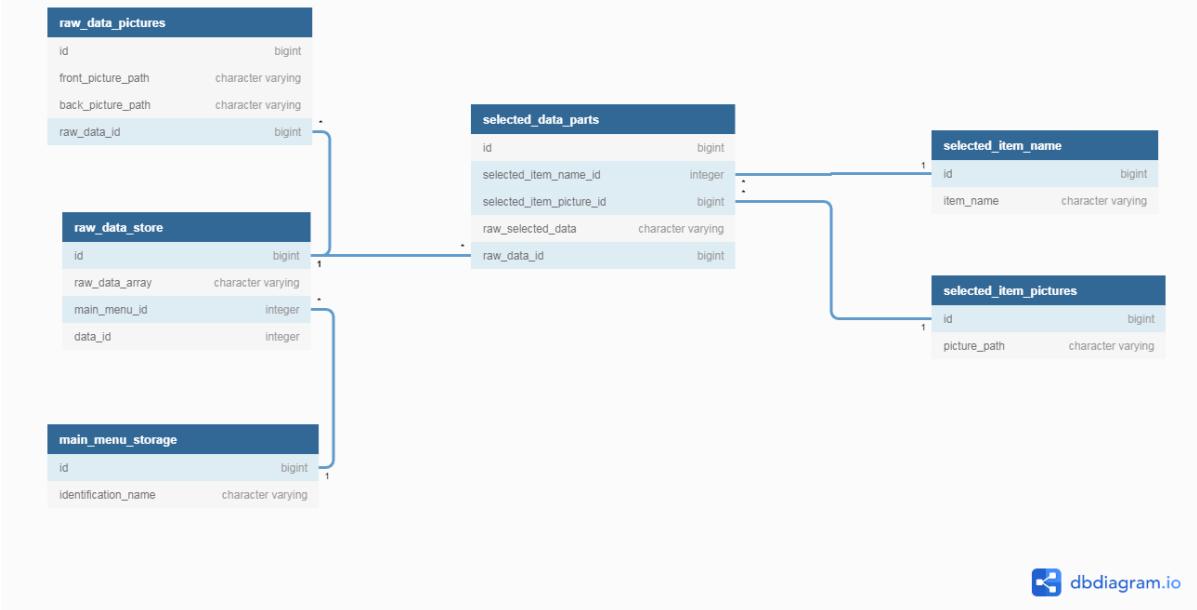
V této kapitole ukáž postup a implementaci systému od úplného začátku až do konce. Začnu na serveru od základní funkcionality až po datový model, pak se přesunu k vizualizaci a selekci i s odesíláním dat na straně uživatele.

4.1 REST API

REST API je rozhraní na straně serveru, které za pomoci HTTP požadavků provádí akce s daty. Cílem je vytvořit akce na url adresy pro získání/uložení LIDARových dat. Akce musí být parametrizovatelné, aby server věděl, jaká data má uživateli poskytnout, popřípadě uložit. Nejdříve potřebuji mít nějaká lidarová data a snímky z přední a zadní strany jízdy. Tato data musí být uložena na serveru, přesněji v databázi, abych je mohl poskytovat uživateli. Dále si uživatel bude moci vybírat určité části lidarového mraku bodů s obrazovými daty, a proto je budu muset uložit. Dále ještě musím poskytnout již zmíněné selekce k znovuzobrazení. Tyto služby budu pomocí REST API poskytovat, aby mohla aplikace na straně uživatele komunikovat se serverem.

4.2 Datový model

Důležitou částí je datový model mého informačního systému. Model si rozdělím podle požadovaných požadavků od serveru, ukládání selekcí objektů a získávání všech možných dat z jednoho mraku bodů. Na obrázku 6 je vidět rozdělení těchto požadavků zobrazených UML diagramem. Tabulka `main_menu_storage` obsahuje všechny unikátní výjezdy s LIDAR zařízením. Každý mrak bodů je uložen na samostatném řádku, tabulka `raw_data_store`, s přímým odkazem na to, v jaké jízdě byl zaznamenán a odkazem na další tabulkou `raw_data_pictures`. V této tabulce jsou uloženy odkazy ke snímkům na disku z přední a zadní strany vozidla. Tyto zmíněné tabulky jsou potřebné k posílání dat na stranu uživatele za účelem vizualizace. Druhou částí je ukládání selekce specifických objektů, které si uživatel vybral. V tabulce `selected_data_parts` jsou uloženy všechny body dané selekce s informací, ze kterého mraku bodů byla vytvořena. Jsou zde uloženy ještě další informace. Mezi tyto informace patří jméno objektu v selekci, tabulka `selected_item_name` a snímek selekce, tabulka `selected_item_pictures`.



Obrázek 6: Datový model informačního systému.

4.3 Scéna a kamera

Scéna je virtuální prostor, do kterého budu přidávat LIDARové body. K požadovanému zobrazení této scény potřebuji ještě kamery a renderer. Kamera ohraničuje specifickou část prostoru, kterou renderer zobrazí. Tato část prostoru se nazývá kamerové frustum. Renderer zajistí, aby se pracovalo pouze s objekty, které leží uvnitř tohoto frustumu. Kamera zajistí projekci viz. 3.6.

```

1 class MainScene {
2     constructor() {
3         this.scene = new THREE.Scene();
4         this.camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight,
5             0.1, 1000);
6         this.renderer = new THREE.WebGLRenderer();
7         this.renderer.setSize(window.innerWidth, window.innerHeight);
8         this.renderer.setClearColor(0xEEEEEE);
9         document.body.appendChild(this.renderer.domElement);
10        window.addEventListener('resize', this.onWindowResize.bind(this), false);
11    }
12
13    animate() {
14        this.frameId = requestAnimationFrame(this.animate.bind(this));
15        this.renderer.render(this.scene, this.camera);
16    }
17
18    stopAnimation() {
19        cancelAnimationFrame(this.frameId);
20    }
21
22    onWindowResize() {
23        this.camera.aspect = window.innerWidth / window.innerHeight;
24        this.camera.updateProjectionMatrix();
25        this.renderer.setSize(window.innerWidth, window.innerHeight);
26    }

```

Výpis 3: Vytvoření scény s kamerou a rendererem.

V kódu 3 je vidět konstruktor a metody `animate`, `stopAnimation` a `onWindowResize`. V konstruktoru vytvářím virtuální scénu, perspektivní kameru a renderer. Vytvoření je velmi snadné s Three.js knihovnou. Pro projekt jsem vybral perspektivní kameru, abych napodobil pohled lidského oka, který zkresluje velikost objektu s rostoucí vzdáleností. Tímto se liší ortografická kamera od perspektivní. K vytvoření kamery potřebuji nejdříve pář důležitých parametrů. Prvním parametrem je `field of view`, neboli zorné pole. Tímto číslem určím úhel ve stupních, který bude mít vertikální úhel kamerového frustumu. Vertikální FOV u běžných kamer jsou v rozmezí 60° - 110° . Druhým parametrem je poměr stran rámce, do kterého budu chtít zobrazit naši scénu. V mém případě chci výsledný rámec přes celou obrazovku prohlížeče, proto `window.innerWidth / window.innerHeight`. Na základě tohoto parametru si Three.js dopočítá horizontální FOV, aby obraz nebyl zkreslený. Třetí parametr značí vzdálenost od pozice kamery k přední stěně pyramidového frustumu. V této vzdálenosti vznikne promítací rovina, na kterou se uživateli promítne scéna a vytvoří obraz do rámce v prohlížeči. Poslední parametr značí také vzdálenost, ale k nejvzdálenější stěně kamerového frustumu. Renderer zajistí vytvoření výsledného HTML canvas plátna, proto je nutné určit mu velikost tohoto plátna. Poté přidám

do objektového modelu dokumentu stránky. To ještě není všechno, stále potřebuji funkci, která bude periodicky vykreslovat aktuální scénu na plátno. K tomuto účelu použiji metodu `animate`, která využívá funkci `requestAnimationFrame`. Tato funkce říká prohlížeči, že chci provést animaci a specifikovat chování překreslení. Proto do této funkce dávám jako parametr referenci na funkci `animate`. Tímto vytvořím nekonečný cyklus, kde v každé iteraci budu na rendereru volat metodu `render`. Ve zmíněném kódu ještě mám vytvořený posluchač na událost změna velikosti, který upraví poměr stran kamery a aktualizuje její projekční matici. Dále už jen aktualizuje velikost výsledného plátna. Výsledkem tohoto kódu je plátno v HTML stránce s pohledem na scénu. Bohužel nic na plátně není vidět, protože žádný objekt není ve scéně.

4.4 Vizualizace LIDARových dat

Zakladem informačního systému je uživateli vizualizovat mrak bodů z LIDARového zařízení. K tomuto účelu nejdříve potřebuji získat data ze serveru. Uživatel musí specifikovat, jaká data chce získat, který mrak bodů, a z jaké jízdy. Musím tedy vytvořit uživateli posuvník, kterým se bude orientovat mezi snímky mraku bodů. Tento posuvník bude nabývat hodnot $\langle 0, m \rangle \subset \mathbb{N}$, kde m je maximální počet uložených mraků bodů pro danou jízdu. Pro získání dat mraku bodů si musím vytvořit požadavek na server. Server prostřednictvím REST-API poskytne objekt, který obsahuje l objektů, kde l je množství lidarových bodů v našem mraku. Každý z těchto objektů obsahuje pole s pozicemi v kartézské soustavě souřadnic. Jak jsem získal takto předzpracovaná data najdete v kapitole 2.8. Pro získání dat použiji kód 4.

```

1  loadDataFromServerAndRenderPoints() {
2      let that = this;
3      let request = new XMLHttpRequest();
4      request.open('GET', '/rawData/' + that.state.menuId + '/' + that.state.stepNumber, true);
5
6      request.setRequestHeader('Content-Type', 'application/json');
7      request.onreadystatechange = function () {
8          if (request.readyState === 4 && request.status === 200) {
9              that.state.lidarPoints = JSON.parse(request.response);
10             that.renderPointsFromData();
11         }
12     };
13     request.send(null);
14 }
```

Výpis 4: Metoda pro získání LIDARových dat ze serveru.

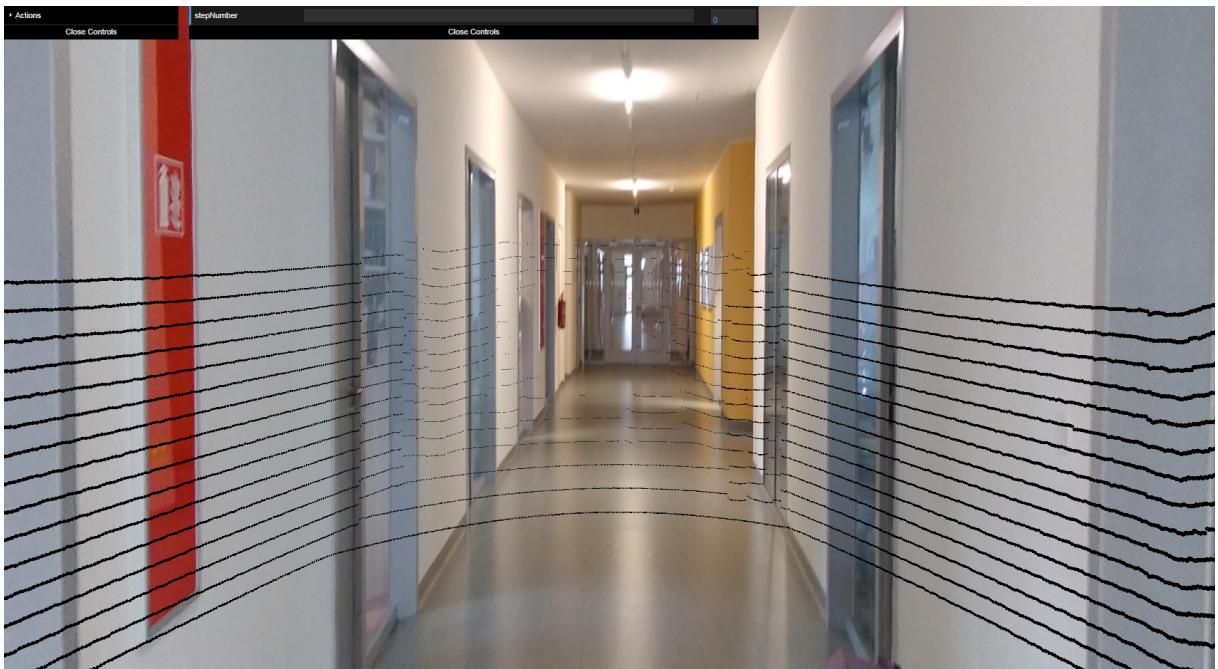
Nejdříve si uložím referenci kontextu do proměnné `that`, protože později budu mít kontext `this` jiný a nedokázal bych získat instanční proměnné z naší třídy. Dále specifikuj, jaký požadavek a na jakou url adresu chci požadavek posílat. Jak můžete vidět, vkládám do url proměnné z aktuálního kontextu. Budu totiž chtít specifikovat, z jaké jízdy brát mraky bodů a zároveň číslo mraku bodů, který uživatel chce vidět. Číslo mraku bodů bude získáno na základě posuv-

níku. Při zaslání požadavku nikdy přesně nevíme, kdy server odpoví a kdy dorazí data, kterých jsme se dotazovali. K tomu mi pomůže funkce `onreadystatechange`. Ta se spustí vždy, když se změní stav požadavku. Vytvořím si uvnitř této metody akce, které budu chtít provést jen když je komunikace ukončena a konečný status požadavku je 200 OK. Nasledně jen zpracuji tělo požadavku z JSON formátu do JavaScript objektu a přejdu k vykreslení mraku bodů.

```
1 renderPointsFromData() {
2     let spheres = this.groups.groupOfPoints;
3     let geometry = new THREE.SphereGeometry(0.005, 5, 5);
4     let material = new THREE.MeshLambertMaterial({color: 0x39ff14});
5
6     let thatPoints = this.state.lidarPoints;
7
8     for (let index in thatPoints) {
9         let value = thatPoints[index];
10        let sphere = new THREE.Mesh(geometry, material);
11        sphere.dynamic = true;
12        sphere.verticesNeedUpdate = true;
13        sphere.position.set(value[0], value[2], -value[1]);
14        spheres.add(sphere);
15    }
16}
```

Výpis 5: Metoda pro vykreslení LIDARových dat.

`Three.js` umožňuje vytvářet skupiny objektů, které slouží k lepší přehlednosti v kódu. Skupina `groupOfPoints`, je již přiřazená do scény. Kdykoliv přidám do této skupiny objekt, automaticky se vloží i do naší virtuální scény. K vytvoření meshe potřebuji dvě věci, geometrii a materiál 3.4. Protože budu přepoužívat geometrii a materiál, tak si je uložím do proměnných ještě před kompletními iteracemi lidarových bodů. V proměnné `thatPoints` jsou data ze serveru. Pak už jen vytvořím cyklus, který v každé iteraci vytvoří nový mesh objekt a nastaví mu pozici ve scéně.



Obrázek 7: Výsledná vizualizace.

4.5 Výběr prostoru

K označení určité části v prostoru musím vzít v potaz, že na obrazovce mám projekci z kamery 3.6. Tohle je důležité si uvědomit, když budu chtít z 2D obrazu promítnout klik do 3D prostoru. Budu to potřebovat na vyznačení určité části prostoru. Zde je vidět kód, který vytvoří vektor se směrem od kamery k promítací rovině.

```

1 let x = (screenX / window.innerWidth) * 2 - 1;
2 let y = -(screenY / window.innerHeight) * 2 + 1;
3
4 let mouse3D = new THREE.Vector3(x, y, 0);
5 mouse3D.unproject(mainCamera);

```

Výpis 6: Projekce myši do 3D prostoru.

Nejprve musím vytvořit událost, abych mohl zjistit pozici kliku. Tato hodnota je vydělena šírkou celého vnitřního okna prohlížeče, to celé vynásobené dvěmi a odečteno jedničkou. Tímto způsobem je vytvořena i y hodnota. Potřebuji, abych ve středu obrazovky měl koordinát (0,0), k tomu tato část kódu slouží. Po této kostrukci budou hodnoty x , y v intervalu $(-1, 1)$. Abych mohl vytvořit vektor v prostoru, tak mi nestačí x , y , proto musím vytvořit trojrozměrný vector s nově vypočítanými hodnotami x a y . Jako hodnota osy z bude nula. Hodnota z by na tomto místě pouze určovala posun od středu kamery k plátnu při interních výpočtech. `THREE.Vector3` má metodu `unproject` s parametrem typu `Camera`, vložím zde hlavní perspektivní kameru. Na směrový vektor z obrazovky se aplikuje projekční matice z kamery. Výsledkem se stane vector,

který ze středu kamery udá směr odpovídající přímce. Na této přímce jsou všechny body, které odpovídají projekci kliku z obrazovky. Obrázek 5 pomůže s lepší představou. Po kliknutí získám pozici na obrazovce, která na obrázku představuje rovinu blíže kamere a bod ležící na ní. Při provedení výše zmíněného kódu získám vektor, který určí směr v trojrozměrném prostoru. Pak už záleží, v jaké vzdálenosti od kamery požadovanou pozici chci.

Selekci musím z dvojrozměrné obrazovky přenést do trojrozměrného prostoru a vizualizovat obrys selekce. Zároveň musí mít obrys všech vyznačených prostorů takovou rotaci, že obrys je natočen kolmo ke středu celého světa. K vytvoření obrysů potřebuji dva body z obrazovky. Pro první bod použiji kód 6, a přidám k němu další část 7.

```

1 let length = Math.sqrt(mouse3D.x ** 2 + (mouse3D.y - cameraYOffset) ** 2 + mouse3D.z ** 2);
2 let scalingFactor = 3 / Math.abs(length);
3 return new THREE.Vector3((scalingFactor * mouse3D.x), ((scalingFactor * (mouse3D.y -
    cameraYOffset)) + cameraYOffset), (scalingFactor * mouse3D.z));

```

Výpis 7: Zvětšení délky vektoru.

Nejdříve spočítám délku vektoru, pak vydělím číslo tří délku vektoru. Číslo tři, protože chci, aby všechny selekce byly v okolí naší pozice maximálně tři jednotky od kamery. Později budu pohybovat mou vyznačenou selekcí okolo mé pozice. Pohybovat jí budu po kouli, která bude hlavní kameru obklopovat. A v posledním kroku vynásobím vektor proměnnou **scalingFactor**. Tímto zajistím zvětšení vektoru při zachování směru.

Druhý bod se bude trošku lišit od projekce prvního bodu.

```

1 function createSecondPointFromPlane(screenX, screenY, firstpoint) {
2     let raycaster = new THREE.Raycaster();
3     let plane = new THREE.Plane();
4     mouse.x = (screenX / window.innerWidth) * 2 - 1;
5     mouse.y = -(screenY / window.innerHeight) * 2 + 1;
6     var result = new THREE.Vector3();
7
8     raycaster.setFromCamera(mouse, mainCamera);
9     plane.setFromNormalAndCoplanarPoint(mainCamera.getWorldDirection(plane.normal), firstpoint);
10    raycaster.ray.intersectPlane(plane, result);
11    return result;
12 }

```

Výpis 8: Promítnutí druhého bodu obrysů selekce.

Zde poprvé ukáži třídu **Raycaster**. Tato třída slouží k detekci objektů. Po inicializaci **Raycasteru** musím namířit na správný směr a nastavit správnou pozici. O to se postará metoda **setFromCamera** se dvěma vstupními parametry. Prvním je **THREE.Vector2**, který obsahuje vypočítané údaje pro projekci bodu a druhým parametrem je hlavní kamera. Nyní si vytvořím rovinu z coplanar bodu a normály. Jako coplanar bod využívám ten, který jsem vytvořil na začátku této podkapitoly. Jako poslední krok řeknu **raycasteru**, aby vyslal paprsek, který má zkusit protnout rovinu a výsledek uložit do proměnné **result**.

```

1 function makeBorder() {
2     scene = document.querySelector('a-scene');
3     mainCamera = scene.camera;
4     let worldRotation = mainCamera.getWorldRotation();
5     firstPoint = create3DPoint(startPointX, startPointY);
6     secondPoint = createSecondPointFromPlane(endPointX, endPointY, firstPoint);
7     yLength = -(firstPoint.distanceTo(new THREE.Vector3(firstPoint.x, secondPoint.y, firstPoint.
8         z)));
8     xLength = firstPoint.distanceTo(new THREE.Vector3(secondPoint.x, firstPoint.y, secondPoint.z
9         ));
9     let line = create3DLine(xLength, yLength);
10    line.rotation.set(worldRotation._x, worldRotation._y, worldRotation._z);
11    line.position.set(firstPoint.x, firstPoint.y, firstPoint.z);
12    line.userData = {camera: camera, xLength: xLength, yLength: yLength, name: '_Selection_' +
13        selectionCounter};
13    groupOfLines.add(line);
14 }

```

Výpis 9: Vytvoření obrysů selekce.

Proměnné `firstPoint` a `secondPoint` odpovídají získaným bodům z této podkapitoly. Z těchto trojrozměrných vektorů musím zjistit délku x a y vytvářeného obdélníku. Tyto vzdálenosti jsou důležité při vytváření obrysového obdélníku.

```

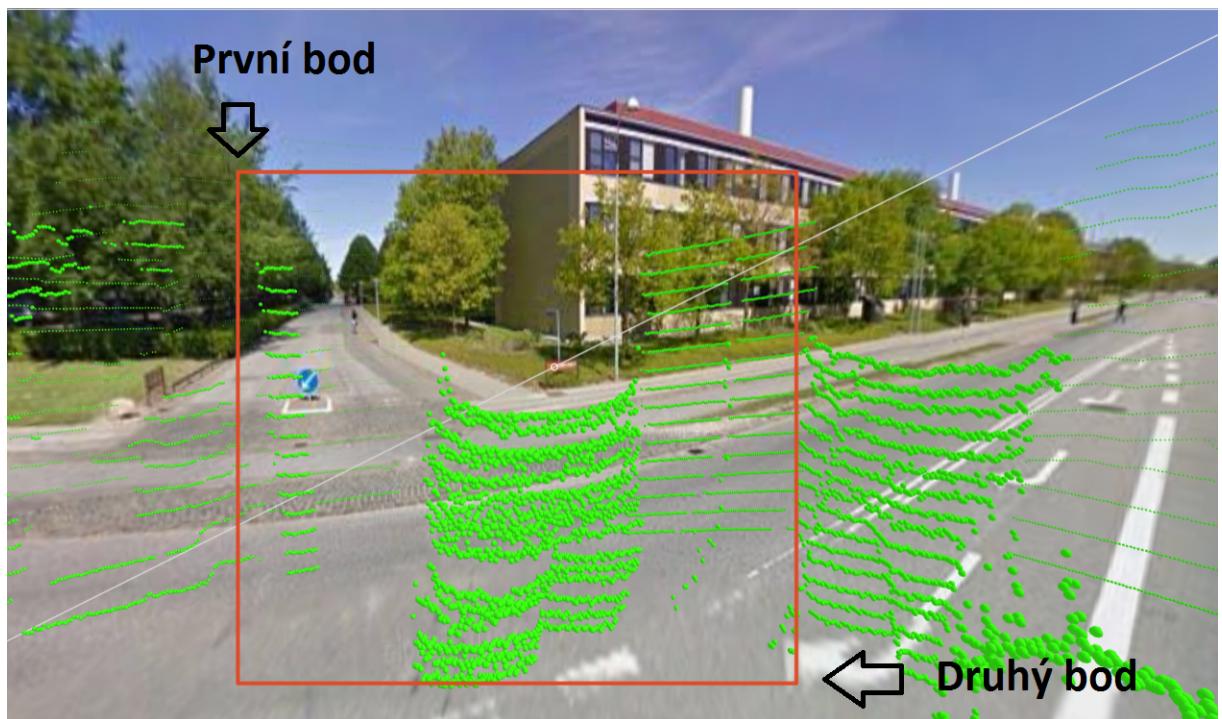
1 function create3DLine(xLength, yLength) {
2     var rectShape = new THREE.Shape();
3     rectShape.moveTo(-xLength / 2, yLength / 2);
4     rectShape.lineTo(xLength / 2, yLength / 2);
5     rectShape.lineTo(xLength / 2, -yLength / 2);
6     rectShape.lineTo(-xLength / 2, -yLength / 2);
7     rectShape.lineTo(-xLength / 2, yLength / 2);
8     rectShape.moveTo(0, 0);
9
10    let geometry = new THREE.ShapeBufferGeometry(rectShape);
11    return new THREE.Line(geometry, material);
12 }

```

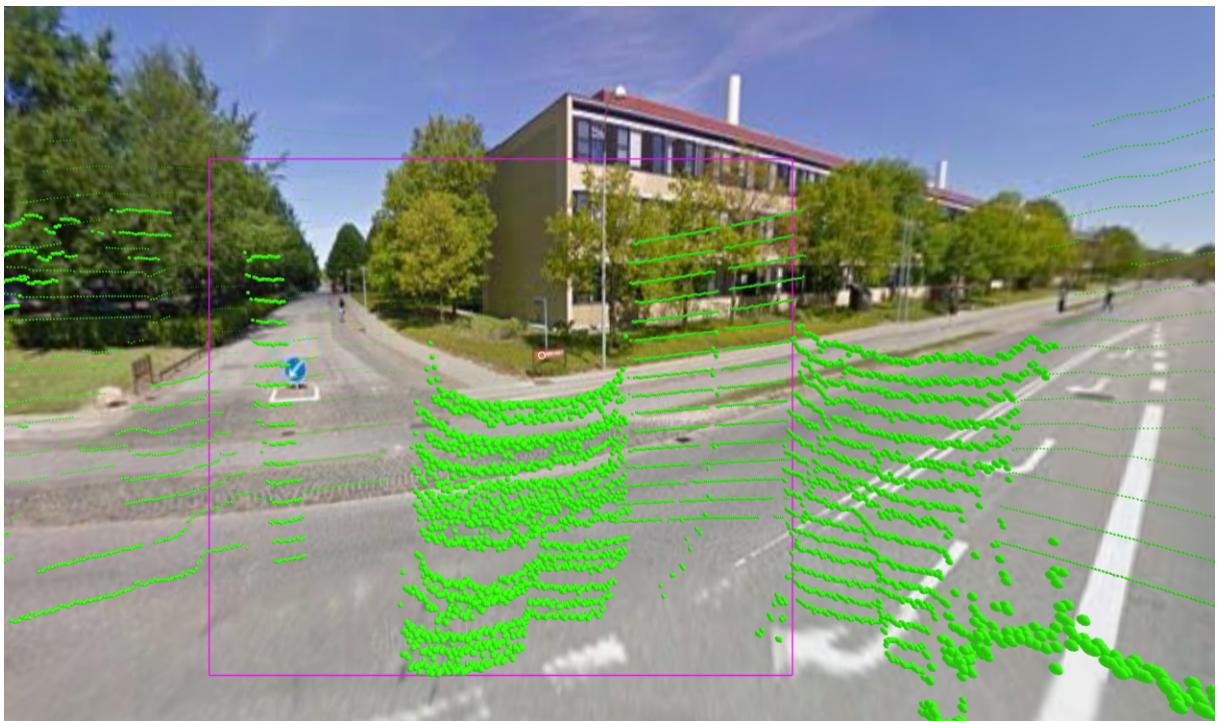
Výpis 10: Použití THREE.Shape k vytvoření obdélníku.

V tomto kódu vytvářím vlastní tvar. Postupně, díky délkám stran obdélníku, vytvářím obdélník s pomocí `THREE.Shape`. Tuto třídu používám z důvodu jednoduchého rozhraní pro vytvoření geometrie objektu v dvojrozměrném prostoru, která je jednoduše přenesitelná do trojrozměrného prostoru. Po vytvoření obdélníkového tvaru z něj získám geometrii, a s pomocí materiálu vytvořím nový objekt typu `THREE.Line`. Tomuto vrácenému objektu musím nastavit pozici, protože všechny objekty mají výchozí pozici ve středu scény. Na nový tvar musím ještě aplikovat rotaci podle rotace světa. Důvodem, proč musím aplikovat rotaci je, že tvar byl vytvořen v dvojrozměrném prostoru a přenesen do trojrozměrného, proto když nový tvar přidám do scény, tak

bude vodorovně s osou x , y , ale kolmo na osu z ve všech případech. Nakonec si ještě přidám do objektu pomocné informace pro budoucí použití a přidám do skupiny obrysů.



Obrázek 8: První bod a druhý bod selekce.



Obrázek 9: Promítnutí bodů s již vytvořeným obrysem.

Zmíněné ukázky kódu nevytvářejí oranžový obrys 8 na obrazovce. Již promítnutá selekce v prostoru 9 je výsledkem ze zmíněných ukázek kódu v kapitole 4.5. Obrys 8 nakreslený s pomocí canvas elementu na 2D obrazovce vypadá naprosto identicky, jako obrys v prostoru 9 díky projekci.

4.6 Zachycení obrazu selekce

V kapitole 3.6 jsem vysvětlil základní princip perspektivní projekce, teď ukáži, jakým způsobem je prováděna. K provedení projekce potřebuji projekční matici. Každá kamera má projekční matici. Tato projekční matice 4×4 je vytvořena při vytvoření kamery. Nejdříve vysvětlím na základě jakých parametrů je perspektivní projekční matice vytvořena, poté ukáži výsledný tvar objektu po aplikování a nakonec implementaci zachycení obrazů selekcí.

$$P = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (2)$$

Zjednodušený vzorec projekční matice kamery, kterou používá knihovna Three.js. Při vytváření nové kamery 11 jsem specifikoval ještě další údaje. Tyto údaje jsou horizontální FOV ve stupních, poměr stran a vzdálenost přední a zadní roviny kamerového frustumu. Všechny tyto informace budu potřebovat na vytvoření projekční matice.

$$y = \frac{1}{\tan(\frac{FOV}{2} * \frac{\pi}{180})}$$

Tímto vzorcem vypočítám délku odvěsný uvnitř kamerového frustumu. Používám funkci tangens s polovinou úhlu FOV, násobenou $\frac{\pi}{180}$ pro převod ze stupňů na radiány. Používám pouze polovinu úhlu FOV, protože mě zajímá pravý úhel uvnitř frustumu.

$$x = \frac{y}{aspect}$$

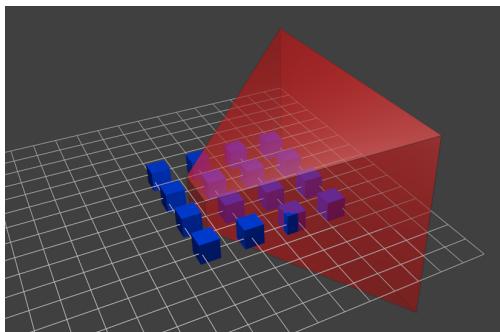
Pro získání této hodnoty využiji výsledek předešlé rovnice a vydělím ji poměrem stran. Tento poměr stran jsem získal při vytváření kamerového objektu. Tímto jsem upravil kamerové frustum tak, aby se nepodobalo pravidelnému rovnostrannému hranolu. Teď přední rovina kamerového frustumu již není čtverec, ale obdélník. Tyto dva parametry matice budou ovlivňovat hodnoty x, y v naší projekci. Hodnoty a, b budou ovlivňovat změnu měřítka objektu na vzdálenosti od pozice kamery.

$$a = \frac{-(f + n)}{f - n}$$

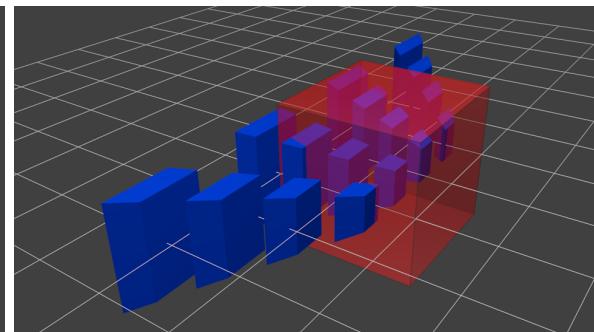
$$b = \frac{-2fn}{f - n}$$

Výsledek hodnoty a bude měnit měřítko souřadnice z v kartézské soustavě souřadnic, a hodnota b bude konstanta, která bude přičtena k výsledné souřadnici z . Proměnná f značí vzdálenost od pozice kamery po nejvzdálenější rovinu kamerového frustumu. Tuto vzdálenost jsem vkládal při vytváření kamerového objektu. Proměnná n označuje vzdálenost k nejbližší rovině kamerového frustumu.

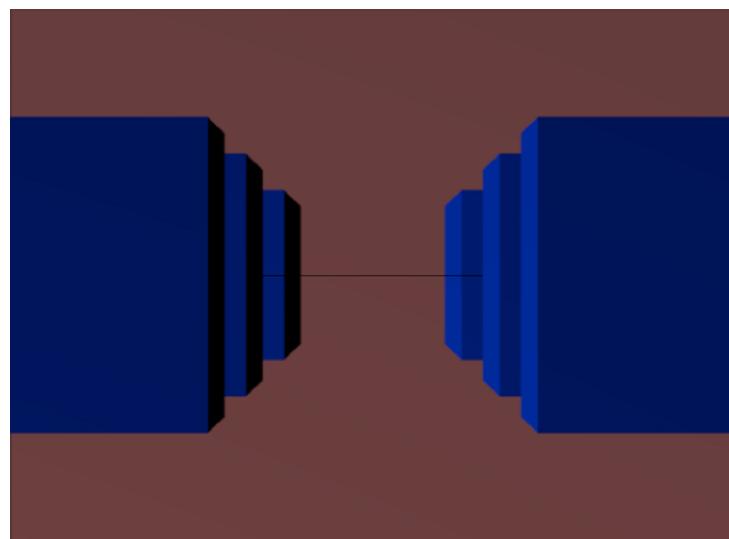
Ukáži, co se stane s objekty po aplikování projekční matice. Na obrázcích 10 a 11 jsou modře zbarvené objekty uvnitř červeného kamerového frustumu. Červeně vyznačená část je jediná část scény, která je viditelná. Pokud vše vynásobím projekční maticí, získám 11. Nyní se z frustumu stal kvádr a všechny objekty byly také transformovány. Objekty, které jsou blíže kameře, jsou větší, a s rostoucí vzdáleností se objekty stávají menšími, stejně jako v reálném životě. Na obrázku 12 vidíme vykreslení finálního výsledku.



Obrázek 10: Ukázka kamerového frustumu [21].



Obrázek 11: Ukázka kamerového frustumu po projekci [21].



Obrázek 12: Výsledný obraz po projekci [21].

K implementaci zachycení obrazu selekce budu potřebovat novou kameru, která bude směřovat na střed naší selekce. Kamery potřebuji k vytvoření nových projekcí. Vytvořím novou perspektivní kameru a namířím ji na střed mezi dvěma námi vytvořenými body v podkapitole 4.5

```

1 let camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 1, 100);
2 camera.position.set(0, 0, 0);
3 camera.lookAt(new THREE.Vector3((firstPoint.x + secondPoint.x), (firstPoint.y + secondPoint.y) ,
4                               (firstPoint.z + secondPoint.z)));
5 groupOfCameras.add(camera);

```

Výpis 11: Vytváření kamery pro sledování selekce.

Tento kód vytvoří novou perspektivní kameru a nastaví ji pozici ve středu scény. Dále ji nastavíme směr, na který se bude dívat. Vektoru s požadovaným směrem docílím tak, že sečtu

vektor prvního a druhého bodu. Výsledkem je vektor procházející středem mezi těmito body. Nakonec přidám tuto kameru do skupiny pomocných kamer.

```
1 createImageFromCameras(scene, renderer, groupOfCameras) {
2     let dataToSend = [];
3     for (let i = 0; i < groupOfCameras.children.length; i++) {
4         renderer.render(scene.object3D, groupOfCameras.children[i]);
5         let dataURL = renderer.domElement.toDataURL();
6         dataToSend.push({
7             key: 'camera' + i,
8             value: dataURL
9         });
10    }
11    this.sendImagesToServer(JSON.stringify(dataToSend));
12 }
```

Výpis 12: Získání snímku z kamery.

4.7 Export vybraných bodů z selekce

Pro získání bodů v označené selekci musím vzít na vědomí, že používám perspektivní kameru a na monitoru vidím projekci. Tím pádem obsah selekce bude připomínat jehlan, který má hlavní vrchol ve středu kamery a každá hrana obrysu leží na rovině procházející pozicí kamery. Roviny rozdělí prostor na požadovanou selekci. Tento prostor obsahuje všechny chtěné LIDARové body. Nejdříve musím vytvořit zmíněné roviny, abych ohraničil prostor selekce. U selekce znám střední bod a délky stran. Pro vytvoření roviny jsou potřeba tři body neležící na jedné přímce. Jeden bod bude společný u všech rovin, a tím bude pozice hlavní kamery. Další dva body budou vždy vrcholy hrany selekce. Problém je, že znám pouze jeden bod, který není ovlivněn rotací. U zbylých bodů budu muset vypočítat pozici po provedené rotaci. Obrys, který nemá žádnou rotaci, je vodorovný s osou x.

```
1 vec8 = that.rotateVectorAndReturnPosition(
2     new THREE.Vector3(line.userData.xLength/2, -line.userData.yLength/2, 0), line);
3
4 rotateVectorAndReturnPosition(vector, line) {
5     let positionOfRotationCenter = line.position.clone();
6     vector.applyEuler(line.rotation.clone());
7     return positionOfRotationCenter.add(vector);
8 }
```

Výpis 13: Získání pozice vrcholu obrysu po provedení rotace.

V ukázce chci získat pravý spodní bod obrysu. Vím, že pozice obrysu je vždy ve středu. Abych získal pravý spodní bod, tak musím posunout středový bod selekce do středu scény a zrušit rotaci. Nyní na základě vzdálenosti x a y našeho obrysu vytvořím vektor směřující k

pravému dolnímu rohu obrysu bez rotace a s pozicí ve středu scény. Na tento vektor použije stejnou rotaci, jako má selekci. Na rotaci se použijí transformační matice (3), (4), (5) [19].

$$A_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix} \quad (3)$$

$$A_y = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \quad (4)$$

$$A_z = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5)$$

Obrys má rotaci uloženou ve vlastnostech `x`, `y`, `z`, a každá z nich popisuje, o kolik radiánů se má otočit kolem stejnojmenné osy. Při zavolání metody `applyEuler()` na proměnné `vector` se tento vektor vynásobí maticí (3), kde $\alpha = \text{line.rotation.x}$. Stejný postup se provede u `y` a `z`.

Když mám všechny potřebné body po transformaci, tak vytvořím roviny jehlanu. Musím zajistit, aby normály všech rovin směřovaly do ohraničeného prostoru.

```

1 let frustum = new CustomFrustum(plane1, plane2, plane3, plane4);
2
3 let lineForRecreation = {
4     position: line.position,
5     rotationEuler: line.rotation,
6     xLength: line.userData.xLength,
7     yLength: line.userData.yLength
8 };
9 let eachResult = {};
10 eachResult.line = lineForRecreation;
11 spheres.children.forEach(function (sphere) {
12     if (frustum.containsPoint(sphere.position)) {
13         eachResult[index] = sphere.position;
14         index++;
15     }
16 });

```

Výpis 14: Získání všech bodů v selekci.

Třída `CustomFrustum` slouží pro uložení našeho jehlanu. Při sběru bodů selekce musím v cyklu proiterovat všechny LIDARové body a zjistit, jestli uvnitř jehlanu. Zjišťuji, jestli vzdálenost od všech rovin je kladné číslo. Pokud ne, tak vím, že bod leží mimo náš jehlan. Další, co si ke všem LIDARovým bodům uložím, je obrys selekce z důvodu opětovného přehrání dat ze selekcí.

4.8 Pohyb selekcí

Než začnu, tak musím vzít v úvahu, že obrys jsou v prostoru scény a já vidím jejich projekci. Takže budu muset zjistit s pomocí projekce na zadní rovinu frustumu kamery, jestli nějaký obrys byl protnuty při této akci. Když zjistím, že obrys byl zasáhnut, tak musím získat všechny potřebné údaje pro budoucí práci s ním. Musel jsem se rozhodnout, jakým způsobem chci, aby se obrys selekce pohyboval v trojrozměrném prostoru, a jakým způsobem budu ovládat pohyb z dvojrozměrné obrazovky. Rozhodl jsem se, že obrys musí být pohybovatelné okolo kamery, a proto pohyb selekce musí kopírovat poloměr koule. Rotace obrysu bude záviset na hlavní kamere, ze které budu pohyb ovládat.

```
1 raycaster.setFromCamera(mouse, camera);
2 var intersects = raycaster.intersectObjects(objects);
3
4 if (intersects.length > 0) {
5     selected = intersects[0].object;
6     if (raycaster.ray.intersectSphere(sphere, intersection)) {
7         offset.copy(intersection).sub(selected.position);
8     }
9 }
```

Výpis 15: Získání obrysu na základě události myši.

Budu potřebovat vytvořit události na adekvátní akce myši. K tomu budu potřebovat použít `THREE.Raycaster`, který pomocí bodu z mé obrazovky zjistí, zda nějaký objekt je ve scéně za touto projekcí. Nejdříve nastavím `raycaster`, aby použil projekční matici hlavní perspektivní kamery, pak použiji metodu `intersectObjects` s parametrem `objects`. Ten obsahuje všechny obrys selekce. Pokud nějaký obrys našel, tak si objekt uložím do třídní proměnné `selected`. Kdyby bylo více obrysů za sebou, metoda by vrátila oba obrysy, ale seřadila by je od nejbližšího po nejvzdálenější. Proměnnou `selected` použiji i při dalších událostech, pak pomocí metody `intersectSphere` získám pozici středu koule, po které chci obrys pohybovat s projekčním páprskem. Výsledek je pak uložen do proměnné `intersection`. Obrys má pouze jediný pevný bod, na který se nevztahuje rotace, a ten je uložen ve vlastnosti `position`. Od tohoto bodu je vytvořena geometrie, která podléhá rotaci. Když budu vybírat obrys v naší scéně, tak nebudu vždy mířit na tento bod, proto potřebuje odchylku kliknutí od pevného bodu. To mi zařizuje proměnná `offset`, která zkopiuje bod dotyku s koulí a odečte z něj pozici pevného bodu obrysu. Výsledkem bude vektor, který při pohybu budu muset odčítat, abych mohl z jakéhokoliv místa pohybovat obrysem a zachoval pohyb po ctené kouli. Kód 15 přiřadím k události `mousedown`.

```
1 var rect = domElement.getBoundingClientRect();
2
3 mouse.x = ((event.clientX - rect.left) / rect.width) * 2 - 1;
4 mouse.y = -((event.clientY - rect.top) / rect.height) * 2 + 1;
5
6 raycaster.setFromCamera(mouse, camera);
```

```
7
8 if (selected && scope.enabled) {
9     if (raycaster.ray.intersectSphere(sphere, intersection)) {
10         selected.position.copy(intersection.sub(offset)).setLength(radius);
11         worldRotation = camera.getWorldRotation();
12         selected.rotation.set(worldRotation._x, worldRotation._y, worldRotation._z);
13     }
14     return;
15 }
```

Výpis 16: Pohyb obrysu ve scéně.

Do proměnné `rect` získám velikost scény v prohlížeči a nastavím `raycaster`, aby směroval na pozici v prostoru, kterou jsem získal projekcí aktuální pozice z obrazovky. Stejně jako v předchozím příkladu použiji proměnnou `intersection` pro získání pozice střetu s koulí po projekci kurzoru na kouli, a nastavím délku tohoto vektoru na poloměr koule. Nakonec nastavím obrysu rotaci na základě kamery, aby obrys měl rotaci proti kameře. Tento kód 16 přiřadím k události `mousemove`.

```
1 if (selected) {
2     let positionOfRotationCenter = selected.position.clone();
3     let vectorToWantedCenter = new THREE.Vector3(selected.userData.xLength / 2, selected.
4         userData.yLength / 2, 0);
5     vectorToWantedCenter.applyEuler(selected.rotation.clone());
6     positionOfRotationCenter.add(vectorToWantedCenter);
7     selected.userData.camera.lookAt(positionOfRotationCenter);
8 }
```

Výpis 17: Zrušení pohybu a zaměření kamery pro zasílání obrázových dat.

Při událostech typu `mouseup`, `mouseleave` musím natočit kameru, s kterou zachycuji obrazová data selekcí 4.6. Musím zjistit bod ve středu obrysu, na který byla použita rotační transformace. K tomu vytvořím vektor, který bude popisovat směr ke středu obrysu bez rotace. Nasledně na tento vektor aplikují kompletní rotaci stejnou jako je na obrysу selekce a přičtu k němu pevný bod selekce. Tímto způsobem jsem získal pozici středu obrysu a nyní jen kameru sledující selekci nastavím, aby sledovala tuto pozici pomocí metody `lookAt`. Nakonec uvolním proměnnou `selected`. Tyto části kódu zařídí, že selekce bude pohybovatelná okolo naší kamery při zachování dosavadní funkcionality systému.

5 Závěr

Cílem bakalářské práce bylo nastudovat potřebné webové knihovny pro vytvoření vizualizačního nástroje LIDARových dat s podporou selekcí obrazových i datových částí. Součástí bylo také nastudování principu LIDARu, ukládání a formát jeho dat. Tato část byla nutná k vizualizování LIDARových dat.

V rámci této práce byla vytvořena vizualizace LIDARových dat ve webovém prohlížeči s podporou selekcí části prostoru. Selekce jsou na základě akcí upravovatelné. Export LIDARových bodů selekce na server plus obrazová data pozadí těchto bodů. Znovupřehrání selekcí i s obsaženými daty také zaimplementováno.

Tato práce mi byla velkým přínosem. Rozšířila mi obzory, a díky ní jsem získal obecný přehled ohledně počítačové grafiky, vizualizačních algoritmů a nepostradatelnou součást matematiky. Délka implementace systému se velmi lišila od mých odhadů z důvodu neznalosti základů geometrie, projekcí kamer, transformačních matic a nulových zkušeností s WebGL. I s brzkým začátkem implementační části projektu se z již zmíněných důvodů produktivní vývoj blížil až ke konci projektu.

Literatura

- [1] URL: http://www.hypack.com/File%20Library/Resource%20Library/Technical%20Notes/01_2018/Velodyne-Laser-Control-Window-in-HYSWEEP-Survey.pdf.
- [2] *63-9276 Rev B VLP-16 Application Note - Packet Structure & Timing Definition.* URL: https://velodynelidar.com/docs/notes/63-9276%20Rev%20A%20VLP-16%20Application%20Note%20-%20Packet%20Structure%20%20Timing%20Definition_Locked.pdf.
- [3] *8.3 - Perspective Projections.* URL: http://learnwebgl.brown37.net/08_projections/projections_perspective.html.
- [4] *About npm.* URL: <https://docs.npmjs.com/about-npm/>.
- [5] Acaron. *Bark of the byte.* URL: <http://barkofthebyte.azurewebsites.net/post/2014/05/05/three-js-projecting-mouse-clicks-to-a-3d-scene-how-to-do-it-and-how-it-works>.
- [6] *Getting KG.* URL: <https://www.goetting-agv.com/components/vlp-16-puck>.
- [7] Ayush Gupta. *HTML Web Component using Plain JavaScript.* URL: <https://www.codementor.io/ayushgupta/vanilla-js-web-components-chguq8goz>.
- [8] Heremaps. *heremaps/pptk.* Říj. 2018. URL: <https://github.com/heremaps/pptk>.
- [9] Sean Higgins. *Velodyne cuts VLP-16 lidar price to \$4k.* Led. 2018. URL: <https://www.spar3d.com/news/lidar/velodyne-cuts-vlp-16-lidar-price-4k/>.
- [10] Krasimir Hristozov. *MySQL vs PostgreSQL – Choose the Right Database for Your Project.* Čvc 2019. URL: <https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres>.
- [11] *HTML & Primitives – A-Frame.* URL: <https://aframe.io/docs/0.9.0/introduction/html-and-primitives.html#sidebar>.
- [12] Velodyne Lidar. *Velodyne LiDAR PUCK.* URL: <https://www.amtechs.co.jp/product/VLP-16-Puck.pdf>.
- [13] *Lincoln Lectures On Graphics - 09.* URL: <https://shearer12345.github.io/graphics/lincolnLecture09.html#/perspective-projection-diagram>.
- [14] Jason Miller. *WTF is JSX.* Ún. 2019. URL: <https://jsonformat.com/wtf-is-jsx/>.
- [15] Ada Nduka Oyom a Ada Nduka Oyom. *Understanding the MVC pattern in Django.* Čvc 2017. URL: <https://medium.com/shecodeafrica/understanding-the-mvc-pattern-in-django-edda05b9f43f>.
- [16] *React A JavaScript library for building user interfaces.* URL: <https://reactjs.org/>.

- [17] Scratchapixel. Lis. 2014. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/perspective-projection>.
- [18] Scratchapixel. Lis. 2014. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/visibility-problem>.
- [19] Eduard Sojka. *Pocítacová grafika II: pruvodce studiem*. Vysoka skola banska - Technicka univerzita, Fakulta elektrotechniky a informatiky, 2003. URL: http://mrl.cs.vsb.cz/people/sojka/pg/pocitacova_grafikaII.pdf.
- [20] *three.js docs*. [Cit. 26.3.2019]. URL: <https://threejs.org/docs/#api/en/geometries/SphereGeometry>.
- [21] *Tutorial 3 : Matrices*. URL: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices>.
- [22] *URL dispatcher*. URL: <https://docs.djangoproject.com/en/2.2/topics/http/urls/>.