

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 Základní pojmy	15
2.4 Odpalovací sekvence	15
2.5 Režimy vrácení dat	15
2.6 Vertikální úhly a kanály	16
2.7 Mrak bodů	16
2.8 Data	16
2.9 Předzpracování dat	18
3 Vizualizace	21
3.1 Výběr technologií	21
3.2 Alternativní knihovny	21
3.3 Základy knihovny Three.js	22
3.4 Vytváření objektů v Three.js	22
3.5 Konkurenční řešení	23
3.6 Princip fungování kamery	24
4 Implementace	26
4.1 REST API	26
4.2 Datový model	26
4.3 Scéna a kamera	27
4.4 Vizualizace LIDARových dat	29
4.5 Výběr prostoru	31
4.6 Zachycení obrazu selekce	35
4.7 Export vybraných bodů z selekce	38
4.8 Pohyb selekcí	40

5 Závěr	42
Literatura	43

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	23
5	Perspektivní projekce kamery [3].	24
6	Datový model informačního systému.	27
7	Výsledná vizualizace.	31
8	První bod a druhý bod selekce.	34
9	Promítnutí bodů s již vytvořeným obrysem.	35
10	Ukázka kamerového frustumu před projekcí [21].	37
11	Ukázka kamerového frustumu po projekci [21].	37
12	Výsledný obraz po projekci [21].	37

Seznam tabulek

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

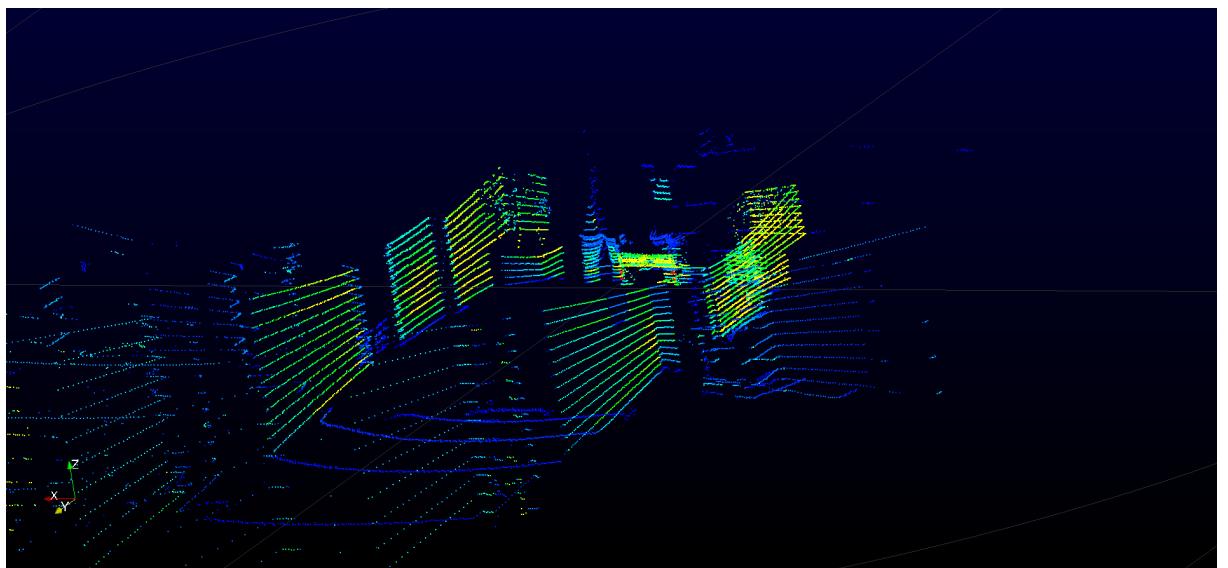
Seznam výpisů zdrojového kódu

1	Python script pro transformaci pcap souborů na JSON soubory.	19
2	Vytvoření koule za pomocí knihovny Three.js	22
3	Vytvoření scény s kamerou a rendererem.	28
4	Metoda pro získání LIDARových dat ze serveru.	29
5	Metoda pro vykreslení LIDARových dat.	30
6	Projekce myši do 3D prostoru.	31
7	Zvětšení délky vektoru.	32
8	Promítnutí druhého bodu obrysu selekce.	32
9	Vytvoření obrysu selekce.	33
10	Použití THREE.Shape k vytvoření obdélníku.	33
11	Vytváření kamery pro sledování selekce.	37
12	Získání snímku z kamery.	38
13	Získání pozice vrcholu obrysu po provedení rotace.	38
14	Získání všech bodů v selekcii.	39
15	Získání obrysu na základě události myši.	40
16	Pohyb obrysu ve scéně.	40
17	Zrušení pohybu a zaměření kamery pro zaslání obrázových dat.	41

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 fyzické výpočetní jednotky se 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. Tyto 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 nám nejdříve pošle data, která ve webovém prostředí zobrazíme. Na tomto prostředí bude naimplementované označování objektů, kde uživatel označí objekt, a pak všechna označená data posíláme na serverovou část pro uložení. Na serverovou část se bude posílat i snímek bez LIDARových dat. Data na serveru budou použita pro následovnou 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 si vysvětlíme princip fungování LIDARu, ukládání dat a přesněji představíme námi použitý model. Také si vysvětlíme pojmy, které budeme používat v dalších kapitolách.

2.1 Princip fungování LIDARu

V úvodu jsme Vám krátce představili, co je to LIDAR a jak fuguje. Pojdme se nyní podívat na fyzické fungování modelu VLP-16, se kterým jsme pracovali. Základní složkou, kterou LIDAR využívá a počítá s ní je světlo a rychlosť světla. LIDAR střílí světelné paprsky, které rychlosť světla letí k cíli. Část tohoto světelného paprsku se po nárazu s objektem odrazí, a putuje zpět stejnou rychlosťí ke zdroji paprsku, kde paprsek zachytí senzor. Po dobu celého tohoto procesu je zaznamenáván čas, takzvaný čas letu. Z těchto informací dostáváme 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 zvážit vlnovou délku v závislosti na využití. 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 stovaceti osmi 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 naše účely. Zde je pář z jeho typických vlastností. Tento model má 16 kanálů. Sběr jedné kompletní 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 elektické 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 se dozvímme v kapitole 2.8.

2.3 Základní pojmy

V této kapitole najdeme nejdůležitější informace, nutné k porozumění LIDARu. Laserovému paprsku se se zvětšující vzdáleností zvětšuje velikost. Velikost rozptylu je 3 mrad. Tímto rozptylem se zároveň snižuje přesnost bodu na větší vzdálenost.

2.4 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 model LIDARu? Například model VLP-16 má 16 úhlů, do kterých se paprsky vysírají.

2.5 Režimy vrácení dat

LIDAR je schopný pracovat ve dvou režimech. Nejdříve si ale budeme muset vysvětlit, jaká data nám LIDAR může vrátit. Při putování paprsku je možné, že paprsek světla narazí do prvního objektu, tomuto bodu dotyku budeme říkat bod A. Tento objekt bude průhledný. Protože je objekt 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 zeď. Místu střetu se zdí budeme říkat bod B. Po tomto střetu se zbytek světelného paprsku odrazí zpět k senzoru. Nyní máme bod A a bod B, které LIDAR zaznamenal. Záleží na nastavení režimu, který rozhodne zda uloží oba body nebo jen jeden. Bod A se nám vrátí k sensoru 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 našem případě bod B, nazýváme 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 nám zařizuje vrácení nejsilnějšího i posledního vráceného bodu. Tato konfigurace nám zaručí přesnější obraz prostředí. Na druhou stranu se pro jeden kompletní obraz posílají 2 rámce s daty. K ukládání dat do rámců se dozvím více v sekci 2.8. Tímto zapříčiníme, že doba celkového obrazu se zdvojnásobí. Tento režim se používá převážně k mapování terénů. Zaznamenaná 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. 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 vrát

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.

2.6 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 vodorovné ploše procházející středem LIDARového senzoru. Každý vrácený bod v datovém bloku je uložen pod číslem kanálu.

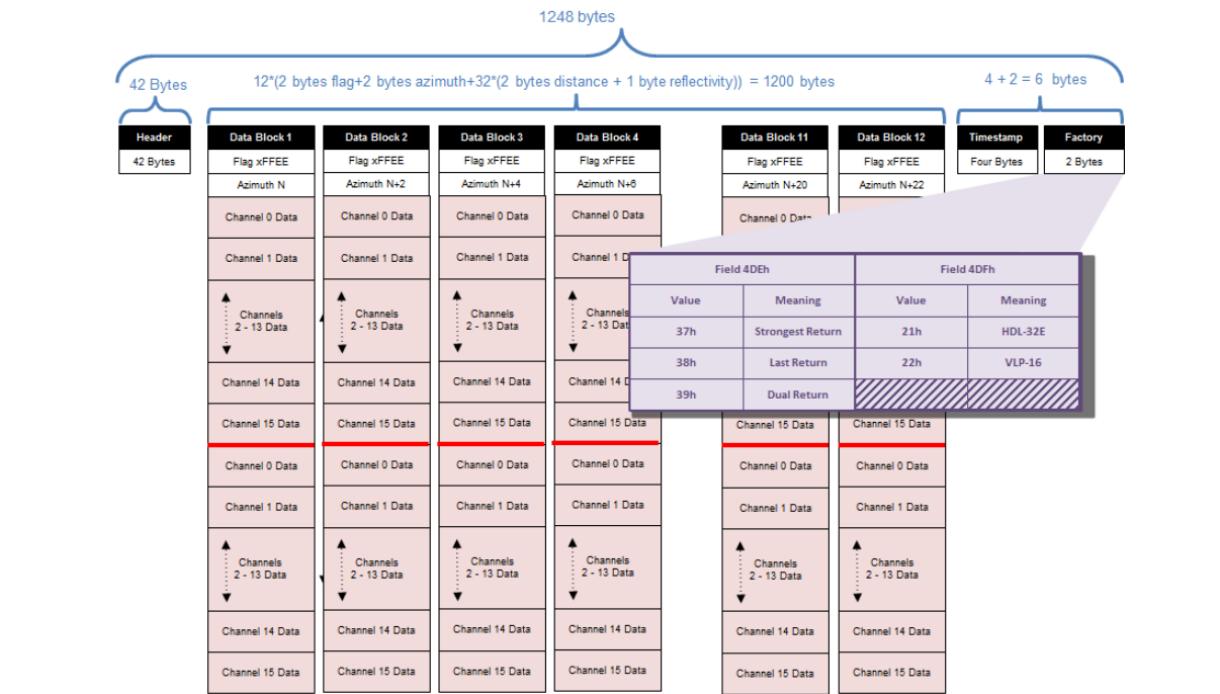
2.7 Mrak bodů

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 se podíváme na výsledek tak vážně vidíme oblak bodů. Tato práce je o vizualizaci těchto bodů v libovolném webovém frameworku a následnou selekcí.

2.8 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 specifickým zaměřením na model VLP-16. Celý paket má velikost 1248 byteů. Těchto 1248 byteů je rozděleno do hlavičky paketu, datových bloků, timestampu a factory. Hlavička paketu má velikost 42 byteů, 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 azimuthu a z 96 byteů čistých dat. Tyto data mají v sobě dvě odpalovací frekvence, kde jedna odpalovací frekvence má 16 kanálů. Dohromady máme 32 kanálů, kde každý kanál se skládá ze 2 byteů vzdálenosti a jednoho bytu reflektivity. Celkem máme $32(2+1)$, to je 96 byteů plus 4 byte. Dohromady máme 100 byteů, které nám 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 nám zbývá posledních 6 byteů do 1248 byteů. V prvních 4 bytech máme 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 2.5, a v posledním bytu je uloženo označení modelu. Hodnoty a význam těchto byteů uvidíme lépe na obrázku 2.

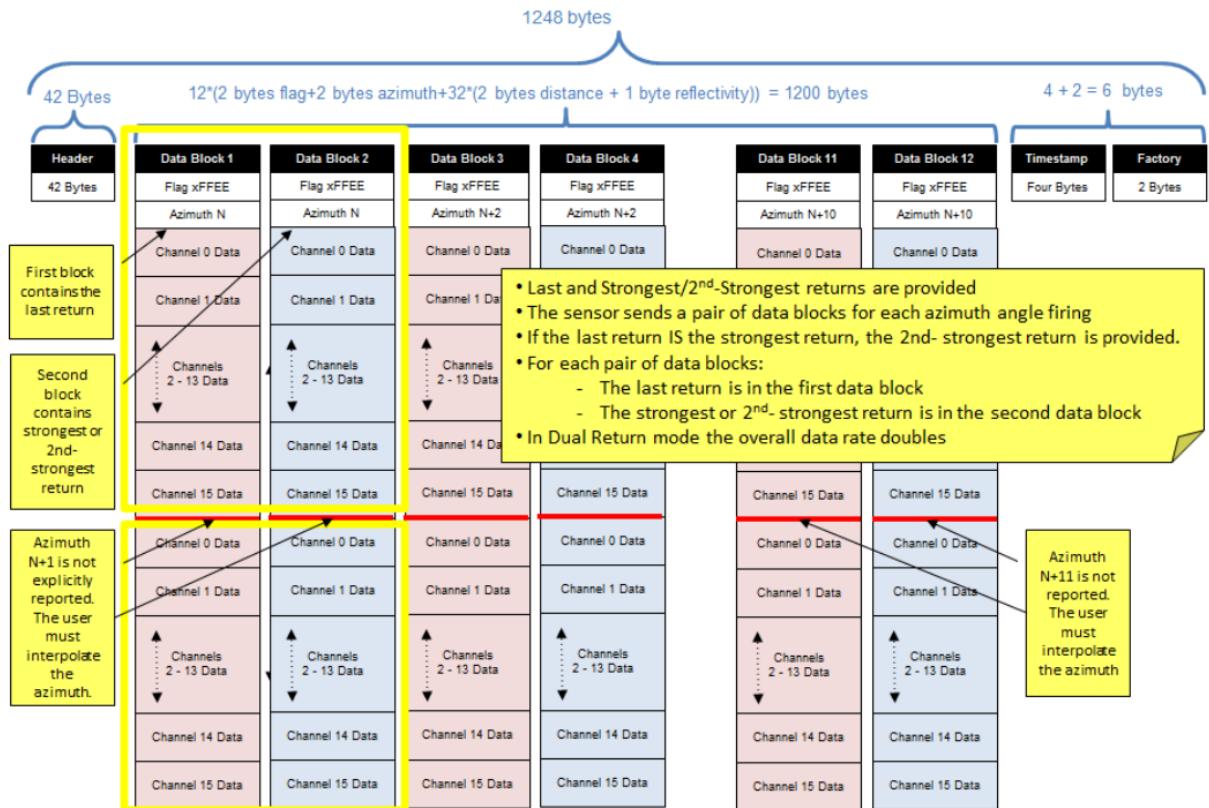


Obrázek 2: Paket modelu VLP-16 [2].

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 vrát 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 obsahu při režimu vrácení jednoho a vrácení dvou. 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 můžeme vidět dva databloky. Každý z nich obsahuje dvě odpalovací sekvence. Tyto dva databloky mají stejný azimut. První i druhý datablok obsahují stejný azimut odpalovacích sekencí, 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].

2.9 Předzpracování dat

Předzpracování dat závisí na tom, zda chceme 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. My nebudeme potřebovat komunikaci a vizualizaci v reálném čase. Účel budovaného systému bude hlavně k vizualizaci LIDARových dat a následnému označování určitých částí mraku bodů, který budeme vracet na serverovou část k úložení snímku vybrané části i všech LIDARových bodů z označené oblasti. Data proto nezískáváme z aktivní komunikace LIDARu se zařízením, ale z pcap souborů vytvo-

řených LIDARem. Pcap soubor je záznam komunikace přes TCP/IP protokol. Tento soubor je složen z mnoha paketů 2.8. K vizualizaci dat z pcap souboru nejdříve musíme extrahovat informace, které jsou uvnitř jednotlivých paketů. Data jsme se rozhodli transformovat rovnou do kartézských souřadnic, abych předešel zbytečným a velmi komplikovaným výpočtům na straně uživatele. K tomuto účelu jsme použili 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 jsme vytvořili jednoduchý script v Pythonu, který využívá již zmíněné rozhraní a přetransformuje nám 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 spouštíme 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 uživatel zobrazen a všechna možná dostupná data, která byla v paketech uložena i s metadaty. Proměnná points nám 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 ukládáme námi potřebná data. Slovník je specifický tím, že se do něj ukládá klíč a na daný klíč je navázáná hodnota, v našem případě klíč je číslo bodu a k tomu navazující hodnota z proměnné points na indexu klíče. Po ukončení všech iterací cyklu vytvoříme soubor a pomocí externí knihovny json, který jsme si na začátku scriptu importovali, vytvoříme ze slovníku JSON řetězec a uložíme ho do souboru. Každý soubor začíná prefixem LidarData_X, kde X je index aktuálního mraku

bodů. Po spuštění tohoto scriptu máme všechna data, která potřebujeme na vizualizaci mraku bodů a následné práce s ním.

3 Vizualizace

Tato část je určena vizualizaci. V této kapitole si ukážeme, jak jsme postupovali 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 budeme mít možnost si uložit námi vybraná data popřípadě obrázky.

3.1 Výběr technologií

Na straně uživatele používáme 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 jsme si vybrali 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 jsme použili tuto 3D knihovnu, byla její rozsáhlá a detailně zpracovaná dokumentace. V serverové části jsme se rozhodli použít programovací jazyk Java. Bude mít za úkol posílat body mraku bodů na stranu uživatele a zpracovat, nebo uložit data odeslaná uživateli. Pro vytvoření webové aplikace jsme využili 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 jsme použili nástroj zvaný **Webpack**. Jeho výhodou je jednoduchá konfigurace a používání. Bundler nám spojí všechny potřebné použité javascript soubory do jednoho zminifikovaného souboru. Pro ukládání dat použijeme objektově-relační databázový server PostgreSQL, který je zdarma a je připravený na vysoký počet požadavků.

3.2 Alternativní knihovny

Je mnoho alternativních knihoven, které jsme mohli použít. Ukážeme si pár z nich. Za zmínu 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 jsme mohli 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. Neměli jsme žádné specifické požadavky od systému, ve kterých bychom mohli říct, že využijeme určitý

programovací jazyk, protože by byl pro tuto problematiku lepší. Požadavky od naší serverové části by efektivně zvládlo mnoho jazyků. Nelze tedy z našich požadavků vybrat jen jeden jazyk a říct, že pro nás bude nejlepší. Pro databázový server jsme měli 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 jsme potřebovali databázový systém, který bude efektivně číst a psát větší množství dat.

3.3 Základy knihovny Three.js

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

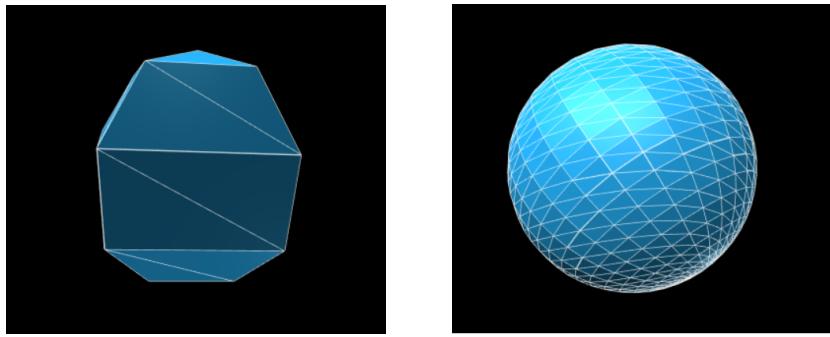
3.4 Vytváření objektů v Three.js

K vytvoření každého objektu potřebujeme množinu vrcholů, stěn a materiálů ve WebGL. Three.js má již jednoduché rozhraní, které nám 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 potřebujeme vrcholy a stěny. O to se nám 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 můžeme 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 je minimum 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 jsme v projektu použili malý počet segmentů na výšku i šířku koulí 5. Když si uvědomíme, že každé vykreslení mraku bodů má 25 000 takových koulí, tak chceme nároky na GPU co nejnižší. Dále potřebujeme ještě materiál, který nám 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 nám velmi pomůže rozlišit jednotlivé koule (LIDARové body) od sebe. Kdybychom použili materiál `THREE.Basic`, tak bychom nerozeznali jednu kouli od druhé. Na tento materiál nepůsobí světlo, proto ať už bychom se podívali na jakoukoliv část koule, tak by měla konstantní odstín barvy materiálu. Koule by splývaly mezi sebou. Při vytváření nové instance materiálu nesmíme zapomenout přidat barvu do konstruktoru. `THREE.Mesh` si jako argument požaduje geometrii a materiál, který má vyplnit stěny dané geometrie. Mesh nám vytvoří výsledný objekt, v našem případě kouli. Tento objekt je možné vložit přímo do scény pro zobrazení koncovému uživateli.

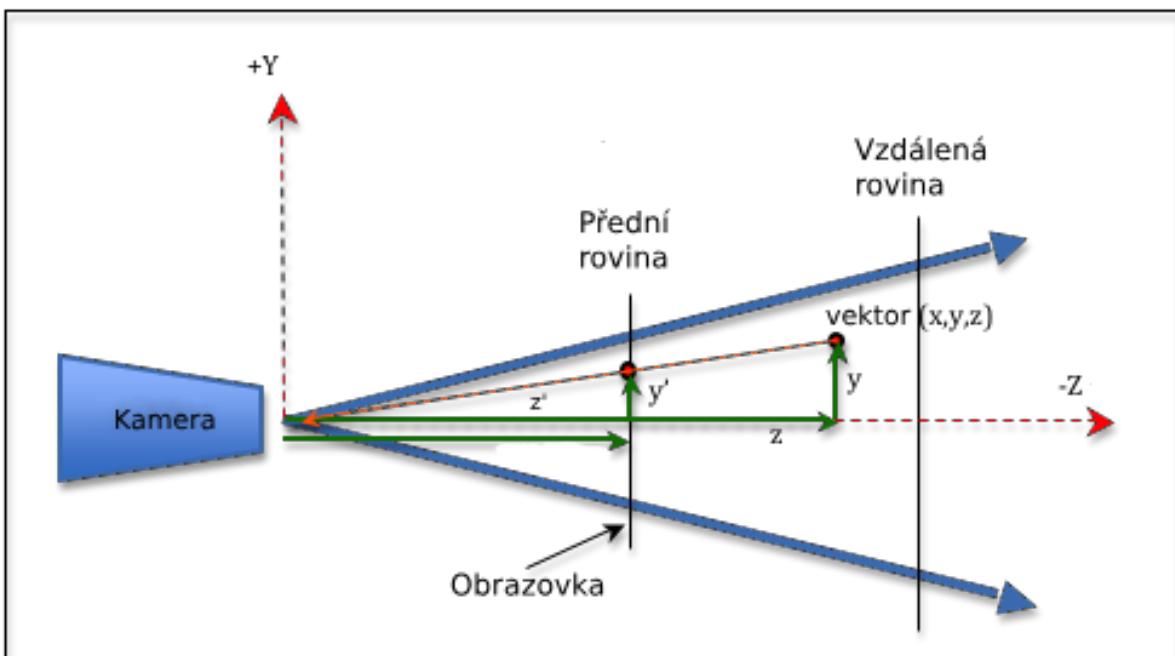
3.5 Konkurenční řešení

Jedno z konkurenčních 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, narozený od našeho vytvářeného systému, není ve webovém rozhraní dostupný odkudkoliv a z jakéhokoliv systému. Dalším rozdílem je, že zobrazuje čistě LIDARová data a nedovoluje nám přidat pozadí, aby bychom si mohli i graficky představit prostředí, v jakém se pohybujeme. 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 narozený od programu VeloView, PPTK data pouze vizualizuje, proto potřebuje

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 nám vytváří perspektivní projekci na scénu. Než si vysvětlíme, jak funguje perspektivní projekce, tak si musíme vysvětlit, na jakém principu pracuje kamera. Perspektivní kamera nám 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 chceme 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 naše obrazovka, do této roviny se provede projekce všeho, co je ve frustumu.



Obrázek 5: Perspektivní projekce kamery [3].

Jak můžeme vidět na obrázku, když chceme promítnout bod na přední rovinu (naše 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 zjištování obrysů ve scéně za našim kurzorem budeme používat Ray-tracing algoritmus. Ray-tracing, jak už z názvu vyplívá, 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. Ukážeme si to na obrázku 5. Paprsek nám vznikne ve středu kamery a protíná určité místo promítací roviny. Tímto se snažíme najít jakýkoliv objekt, který za tímto místem je. Používáme implementaci tohoto algoritmu ve třídě **raycaster**.

4 Implementace

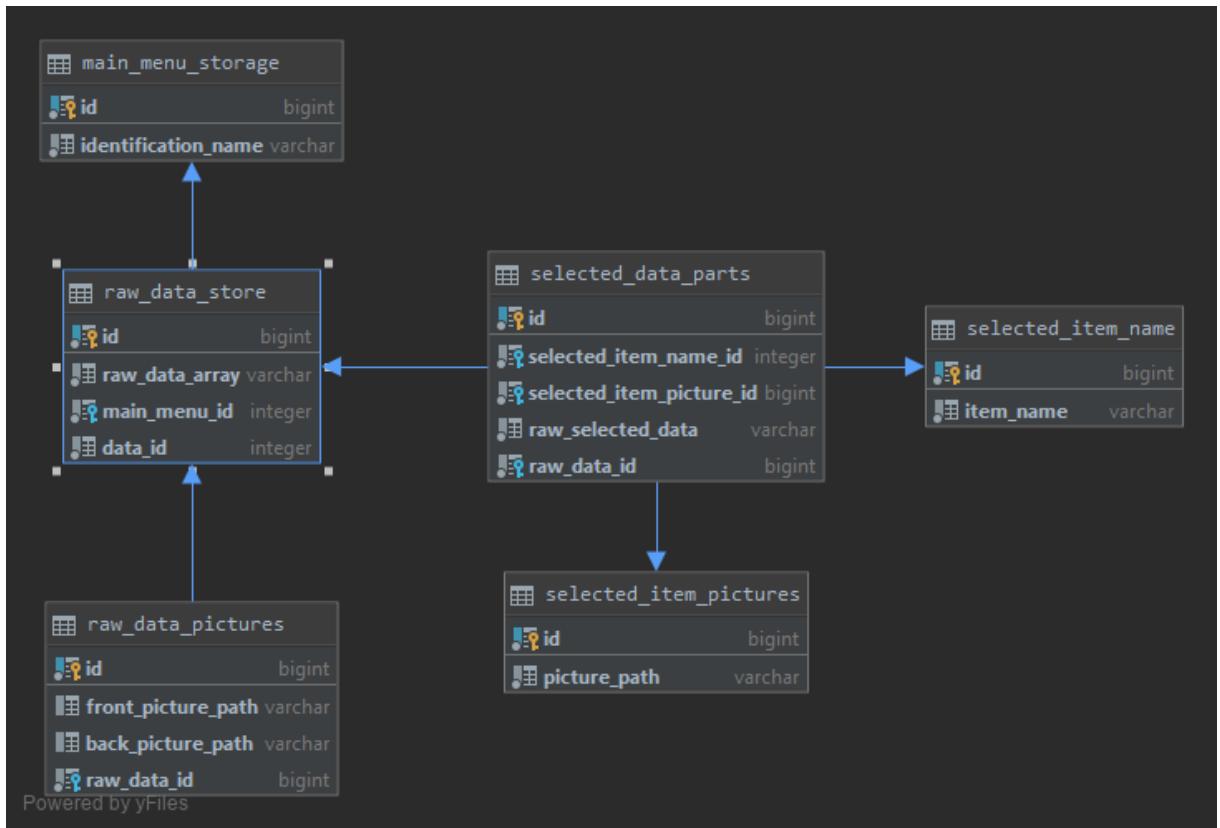
V této kapitole najdeme postup a implementaci systému od úplného začátku až do konce. Začneme na serveru od základní funkcionality až po datový model, pak se přesuneme na vizualizaci a selekci i s odesíláním dat na straně uživatele.

4.1 REST API

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

4.2 Datový model

Důležitou částí je datový model našeho informačního systému. Model si rozdělíme podle chtěných požadavků od serveru, ukládání selekcí objektů a získávání všech možných dat z jednoho mraku bodů. Na obrázku 6 můžeme 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 tabulku `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 uložené 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 náš virtuální prostor, do kterého budeme přidávat LIDARové body. K chtěnému zobrazení této scény potřebujeme ještě kameru a renderer. Kamera nám ohraničuje specifickou část prostoru, kterou nám renderer zobrazí. Tato část prostoru se nazývá kamerové frustum. Renderer nám zajistí, aby se pracovalo pouze s objekty, které leží uvnitř tohoto frustumu. Kamera nám 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 vidíme konstruktor a metody `animate`, `stopAnimation` a `onWindowResize`. V konstruktoru vytvoříme virtuální scénu, perspektivní kameru a renderer. Vytvoření je velmi snadné s `Three.js` knihovnou. Pro náš projekt jsme si vybrali perspektivní kameru, abychom napodobili pohled lidského oka, které zkresluje velikost objektu s rostoucí vzdáleností. Tímto se liší ortografická kamera od perspektivní. K vytvoření kamery potřebujeme nejdříve pář důležitých parametrů. Prvním parametrem je `field of view`, neboli zorné pole. Tímto číslem určíme úhel ve stupních, který bude mít vertikální úhel kamerového frustumu. Vertikální FOV u běžných kamer jsou v rozmezí $60^\circ - 110^\circ$. Druhým parametrem je poměr stran rámce, do kterého budeme chtít zobrazit naši scénu. V našem případě chceme 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 nám vznikne promítací rovina, na kterou se nám promítne scéna a vytvoří nám obraz do našeho rámce v prohlížeči. Poslední parametr značí také vzdálenost, ale k nejvzdálenější stěně kamerového frustumu. Renderer nám zajistí vytvoření výsledného `HTML canvas` plátna, proto mu musíme určit velikost tohoto

plátna. Už stačí jen přidat do objektového modulu dokumentu stránky. To ještě není všechno, stále potřebujeme funkci, která nám bude periodicky vykreslovat aktuální scénu na plátno. K tomuto účelu použijeme metodu `animate`, která využívá funkci `requestAnimationFrame`. Tato funkce říká prohlížeči, že chceme provést animaci a specifikovat chování překreslení. Proto do této funkce dáváme jako parametr referenci na funkci `animate`. Tímto vytvoříme nekonečný cyklus, kde v každé iteraci budeme na rendereru volat metodu `render`. Ve zmíněném kódu ještě máme vytvořený posluchač na událost `změna velikosti`, který nám upraví poměr stran kamery a aktualizuje její projekční matici. Dále už jen aktualizuje velikost výsledného plátna. Výsledkem tohoto kódu máme na naší HTML stránce plátno, s pohledem na scénu. Bohužel nic na našem plátně neuvidíme z důvodu, že nemáme žádný objekt ve scéně.

4.4 Vizualizace LIDARových dat

Zakladem našeho informačního systému je uživateli vizualizovat mrak bodů z LIDARového zařízení. K tomuto účelu nejdříve potřebujeme získat data ze serveru. Uživatel musí specifikovat jaká data chce získat, který mrak bodů a z jaké jízdy. Musíme tedy vytvořit uživateli posuvník, kterým se bude orientovat mezi snímky mraku bodů. Tento posuvník bude nabívat hodnot $\langle 0, m \rangle \subset \mathbb{N}$, kde m je maximálního počtu uložených mraků bodů pro danou jízdu. Pro získání dat mraku bodů si musíme vytvořit požadavek na server. Server nám prostřednictvím REST-API poskytne objekt, který obsahuje 1 objektů, kde 1 je množství lidarových bodů v našem mraku. Každý z těchto objektů obsahuje pole s pozicemi v kartézské soustavě souřadnic. Jak jsme získali takto předzpracovaná data najdeme v kapitole 2.9. Pro získání dat použijeme 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.

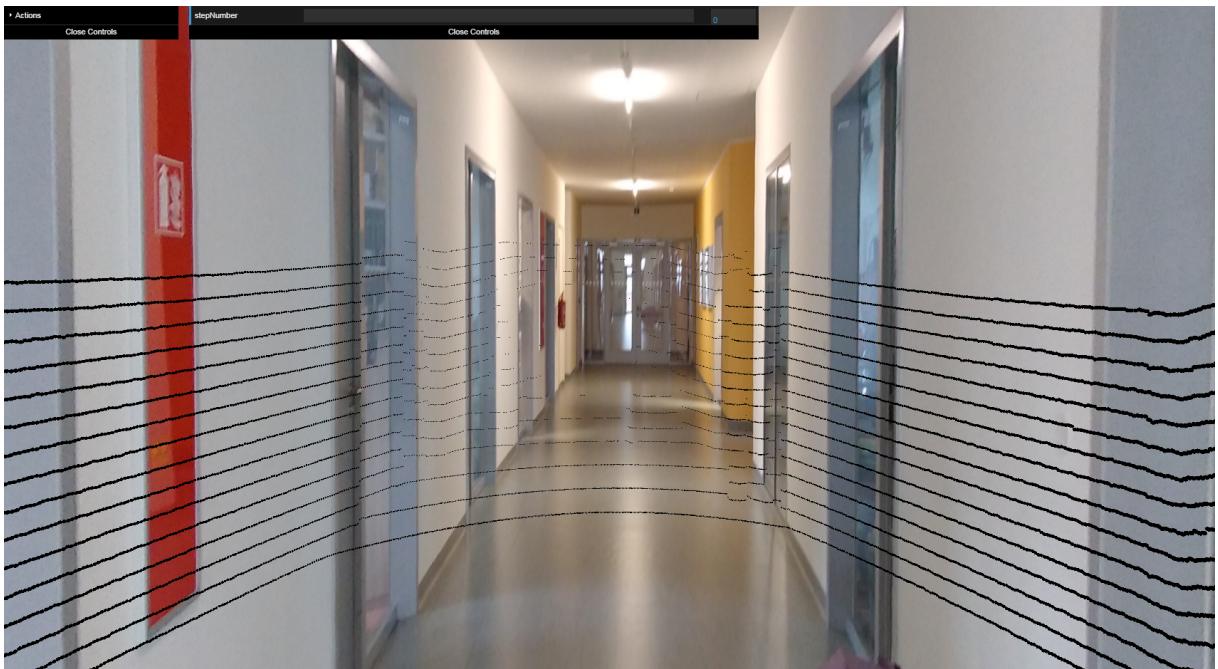
Nejdříve si uložíme referenci kontextu do proměnné `that`, protože později budeme mít kontext `this` jiný a nedokázali bychom se získat instanční proměnné z naší třídy. Dále specifikujeme jaký požadavek a na jakou url adresu chceme požadavek posílat. Jak můžeme vidět vkládáme do url proměnné z aktuálního kontextu. Budeme totiž chtít specifikovat z jaké jízdy máme brát

mraky bodů a zároveň číslo mraku bodů, který uživatel chce vidět. Číslo mraku bodů bude získáno na základě posuvníku. Při zaslání požadavku nikdy přesně nevíme kdy nám server odpoví a kdy nám dorazí data, které jsme chtěli. K tomu nám pomůže funkce `onreadystatechange`. Ta se spustí vždy, když se změní stav požadavku. My si vytvoříme uvnitř této metody akce, které budeme chtít provést jen když je komunikace ukončena a konečný status požadavku je 200 OK. Nasledně jen zpracujeme tělo požadavku z JSON formátu do JavaScript objektu a přejdeme 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` nám umožnuje vytvářet skupiny objektů, které slouží k lepší přehlednosti v kódu. Naši skupinu, `groupOfPoints`, máme již přiřazenou do scény. Kdykoliv přidáme do této skupiny objekt, automaticky se nám vloží i do naší virtuální scény. K vytvoření meshe potřebujeme dvě věci, geometrii a materiál 3.4. Protože budeme přepoužívat geometrii a materiál tak si je uložíme do proměnných, ještě před kompletními iteracemi lidarových bodů. V proměnné `thatPoints` máme data ze serveru. Pak už jen vytvoříme cyklus, který v každé iteraci vytvoří nový mesh objekt a nastavíme mu pozici ve scéně.



Obrázek 7: Výsledná vizualizace.

4.5 Výběr prostoru

K označení určité části v prostoru si musíme uvědomit, že na naší obrazovce máme projekci z kamery 3.6. Tohle je důležité si uvědomit, když budeme chtít z našeho 2D obrazu promítat náš klik do 3D prostoru. Budeme to potřebovat na vyznačení určité části prostoru. Zde můžeme vidět kód, který nám 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 musíme vytvořit událost, abychom mohli 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. Potřebujeme, abychom ve středu obrazovky měli koordinát (0,0), k tomu tato část kódu slouží. Po této konstrukci nám budou hodnoty x, y v intervalu $\langle -1, 1 \rangle$. Aby jsme mohli vytvořit vektor v prostoru, tak nám nestačí x, y, proto musíme vytvořit trojrozměrný vector s nově vypočítanými hodnotami x a y. Jako hodnota osy z bude nula. Hodnota z by nám na tomto místě pouze určovala posun od středu kamery k plánu při interních výpočtech. `THREE.Vector3` má metodu `unproject` s parametrem typu `Camera`, vložíme zde naši hlavní perspektivní kamery. Na vektor se aplikuje projekční matice z kamery.

Výsledkem se stane vector, který nám ze středu kamery udá směr odpovídající přímce. Na této přímce jsou všechny body, které odpovídají projekci naší akce z obrazovky. Když se podíváme na obrázek 5, tak nám pomůže si to lépe představit. Při kliknutí získáme pozici na obrazovce, která na obrázku představuje rovinu blíže kameře a bod ležící na ní. Při provedení výše zmíněného kódu získáme vektor, který nám určí směr v trojrozměrném prostoru. Pak už záleží na nás, v jaké vzdálenosti od kamery požadovanou pozici chceme.

Selekci musíme z dvojrozměrné obrazovky přenést do trojrozměrného prostoru a vizualizovat obrys selekce. Zároveň musí mít obrys všech vyznačených prostorů rotaci takovou, že obrys je natočen kolmo ke středu celého světa. K vytvoření obrysů potřebujeme dva body z obrazovky. Pro první bod použijeme kód 6, a přidáme 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 spočítáme délku vektoru, pak vydělíme číslo tří délku vektoru. Číslo tři, protože chceme, aby všechny selekce byly v okolí naší pozice maximálně tři jednotky od nás. Později budeme pohybovat naší vyznačenou selekcí okolo naší pozice. Pohybovat jí budeme po kouli, která nás bude obklopat. A v posledním kroku vynásobíme náš vektor proměnnou `scalingFactor`. Tímto zajistíme 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 se poprvé setkáváme s třídou `Raycaster`. Tato třída slouží k detekci objektů. Po inicializaci `raycasteru` musíme namířit na správný směr a nastavit správnou pozici. O to se nám postará metoda `setFromCamera` se dvěma vstupními parametry. Prvním je `THREE.Vector2`, který obsahuje vypočítané údaje pro projekci bodu a druhým parametrem je naše hlavní kamera. Nyní si vytvoříme rovinu z coplanar bodu a normály. Jako coplanar bod využíváme ten, který jsme vy-

tvořili na začátku této podkapitoly. Jako poslední krok řekneme raycasteru, aby vyslal paprsek, který má zkoušet 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)));
9     xLength = firstPoint.distanceTo(new THREE.Vector3(secondPoint.x, firstPoint.y, secondPoint.z
9         ));
10    line = create3DLine(xLength, yLength);
11    line.rotation.set(worldRotation._x, worldRotation._y, worldRotation._z);
12    line.position.set(firstPoint.x, firstPoint.y, firstPoint.z);
13    line.userData = {camera: camera, xLength: xLength, yLength: yLength, name: '_Selection_' +
14        selectionCounter};
14 }
```

Výpis 9: Vytvoření obrysů selekce.

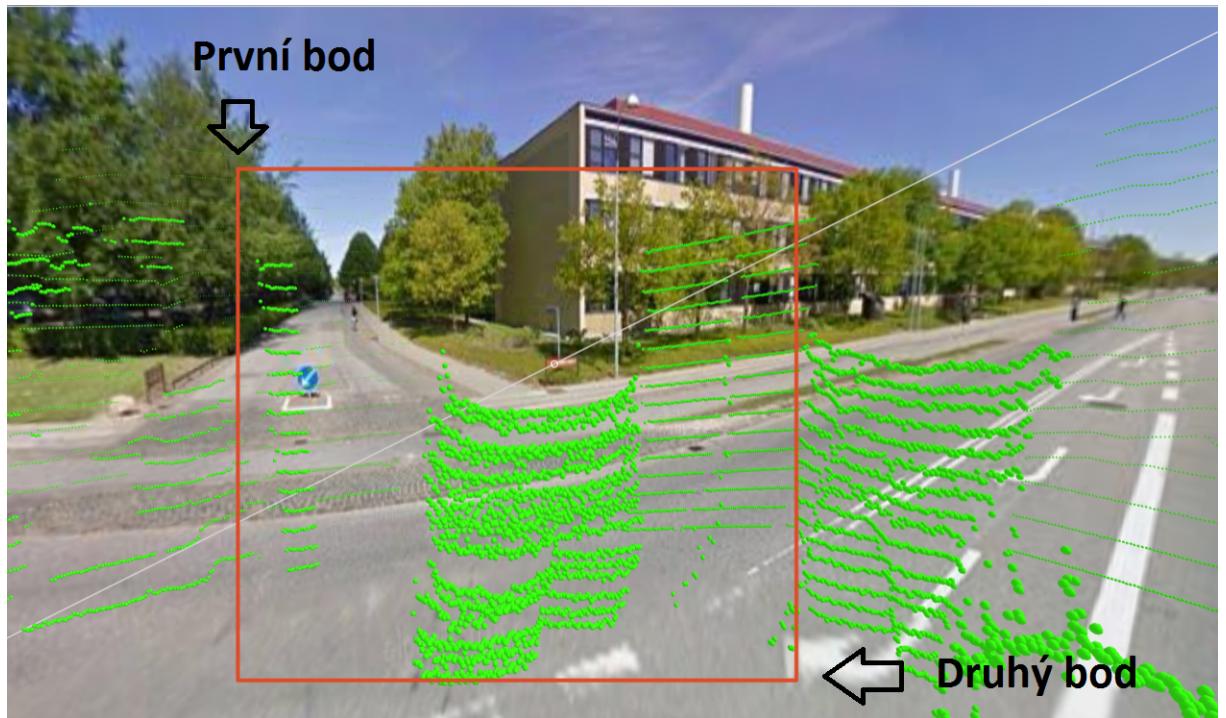
Proměnné `firstPoint` a `secondPoint` odpovídají získaným bodům z této podkapitoly. Z těchto trojrozměrných vektorů musíme zjistit délku x a y vytvářeného obdélníku. Tyto vzdálenosti jsou důležité při vytváření našeho obrysového obdélníku.

```
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 }
```

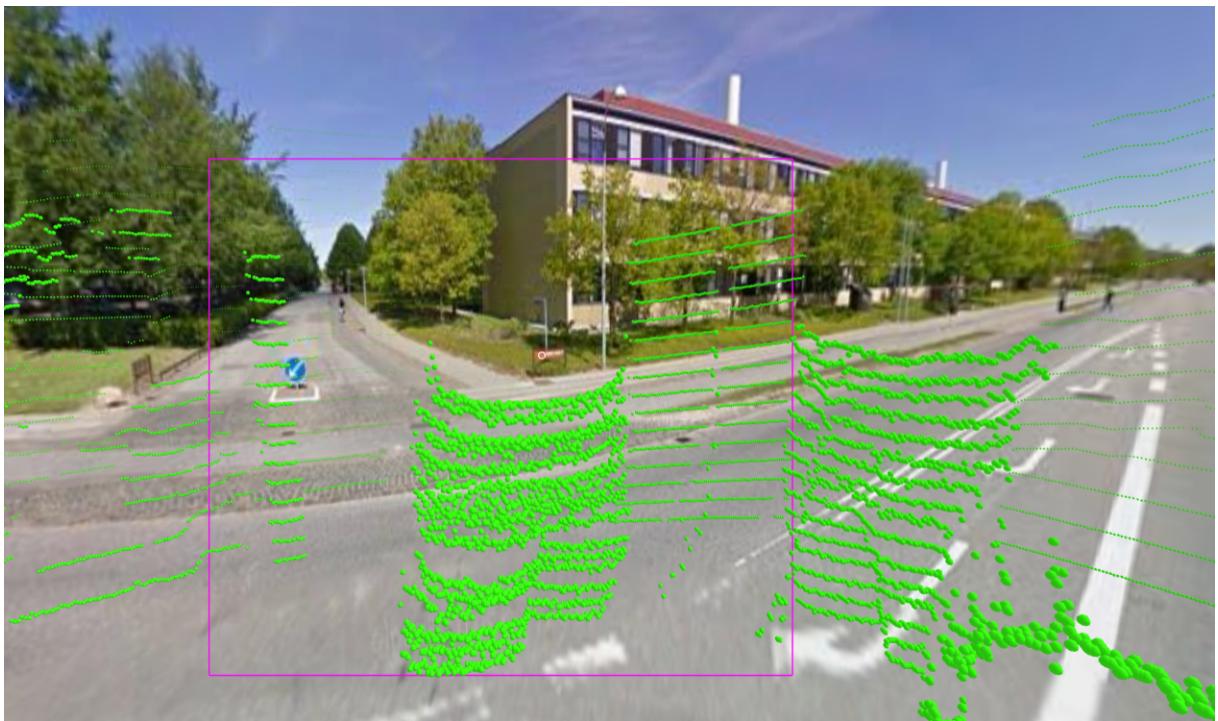
Výpis 10: Použití THREE.Shape k vytvoření obdélníku.

V tomto kódu vytváříme náš vlastní tvar. Postupně, díky délkám stran obdélníku, vytváříme náš vlastní obdélník s pomocí `THREE.Shape`. Tuto třídu používáme z důvodu jednoduchého rozhraní pro vytvoření geometrie objektu v dvojrozměrném prostoru, která je jednoduše přenesitelná do trojrozměrného prostoru. Po vytvoření našeho obdélníkového tvaru z něj získáme geometrii, a s pomocí materiálu vytvoříme nový objekt typu `THREE.Line`. Tomuto vrácenému objektu musíme nastavit pozici, protože všechny objekty mají výchozí pozici ve středu scény. Na náš nový tvar musíme ještě aplikovat rotaci podle rotace světa. Důvodem, proč musíme aplikovat

rotaci je, že tvar byl vytvořen v dvojrozměrném prostoru a přenesen do trojrozměrného, proto když náš nový tvar přidáme do scény, tak bude vodorovně s osou x, y, ale kolmo na osu z ve všech případech. Nakonec si ještě přidáme do našeho objektu pomocné informace pro budoucí použití a přidáme 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 z námi 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

Z kapitoly 3.6 jsme se dozvěděli základní princip perspektivní projekce, teď si ukážeme jakým způsobem je prováděna. K provedení projekce potřebujeme projekční matici. Každá kamera má projekční matici. Tato projekční matice 4×4 je vytvořena při vytvoření kamery. Nejdříve si vysvětlíme na základě jakých parametrů je perspektivní projekční matice vytvořena, poté si ukážeme výsledný tvar objektu po aplikování a nakonec implementaci zachycení obrazů selekcí.

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

Zde můžeme vidět zjednodušený vzorec projekční matice kamery, kterou používá knihovna `Three.js`. Při vytváření nové kamery 11 jsme specifikovali ještě další údaje. Tyto údaje jsou horizontální FOV ve stupních, poměr stran, vzdálenost přední a zadní roviny kamerového frustumu. Všechny tyto informace budeme potřebovat na vytvoření projekční matice.

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

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

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

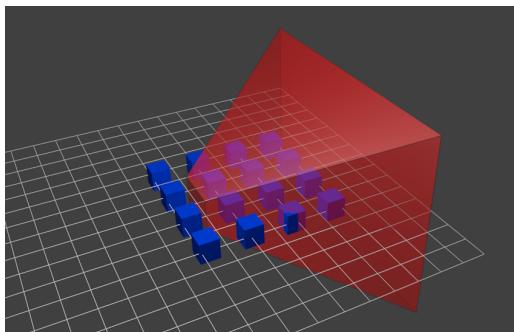
Pro získání této hodnoty využijeme výsledek předešlé rovnice a vydělíme ji poměrem stran. Tento poměr stran jsme získali při vytváření kamerového objektu. Tímto jsme upravili kamerové frustum tak, aby se nepodobalo pravidelnému rovnostrannému hranolu. Teď přední rovina kamerového frustumu již není čtverec, 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.

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

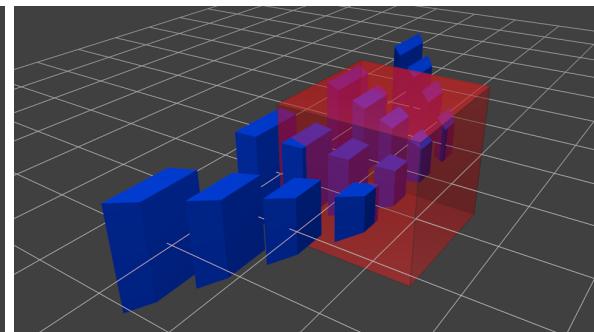
$$b = \frac{-2fn}{f - n}$$

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 nám značí vzdálenost od pozice kamery po nejvzdálenější rovinu kamerového frustumu. Tuto vzdálenost jsme vkládali při vytváření kamerového objektu. Proměnná n nám označuje vzdálenost k nejbližší rovině kamerového frustumu.

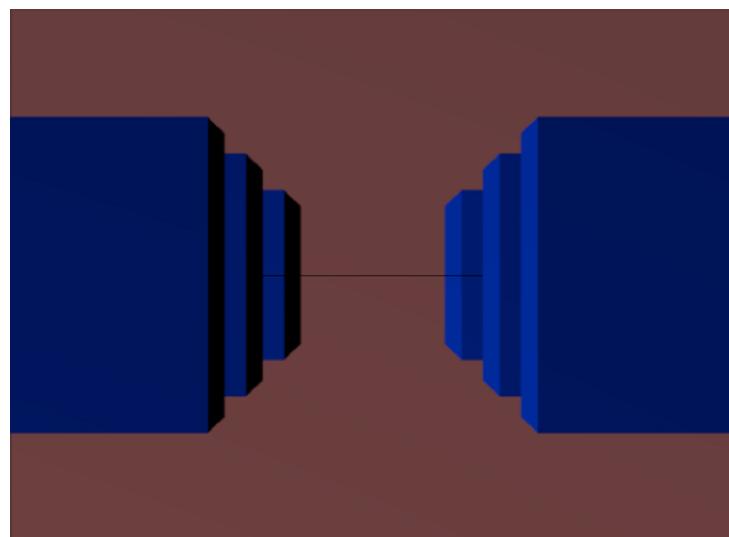
Ukážeme si, co se stane s objekty po aplikování projekční matice. Na obrázcích 10, 11 vidíme modře zbarvené objekty uvnitř červeného kamerového frustumu. Červeně vyznačená část je jediná část scény, kterou jsme schopni vidět. Pokud vše vynásobíme projekční maticí, získáme 11. Nyní se z našeho frustumu 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 můžeme vidět 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 budeme potřebovat novou kameru, která bude směřovat na střed naší selekce. Kameru potřebuje, aby nám vytvořila novou projekci. Vytvoříme si novou perspektivní kameru a namíříme 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)));
groupOfCameras.add(camera);

```

Výpis 11: Vytváření kamery pro sledování selekce.

Při vizualizování selekce také vytvoříme novou perspektivní kameru pro zachycení snímku selekce. Tento kód nám vytvoří novou perspektivní kameru a nastaví ji pozici ve středu scény. Dále ji nastavíme směr, na který se bude dívat. Vektoru s naším chtěným směrem docílíme

tak, že sečteme vektor prvního a druhého bodu. Výsledkem je vektor procházející středem mezi těmito body. Nakonec přidáme naši 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 si musíme uvědomit, že používáme perspektivní kamery a na monitoru vidíme projekci. Tím pádem obsah naší selekce bude připomínat jehlan, který má hlavní vrchol ve středu kamery a každá hrana obrysu leží na rovině procházející pozicí kamery. Roviny nám rozdělí prostor na požadovanou selekci. Tento prostor obsahuje všechny námi chtěné LIDARové body. Nejdříve musíme vytvořit naše roviny, abychom ohraničili náš prostor selekce. U našeho obrysu známe 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 obrysu. Problém je, že známe pouze jediný bod, který není ovlivněn rotací. U zbylých bodů budeme 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 chceme získat pravý spodní bod obrysu. Víme, že pozice obrysu je vždy ve středu. Abychom získali pravý spodní bod, tak musíme posunout středový bod selekce do středu scény a zrušit rotaci. Nyní na základě vzdálenosti x a y našeho obrysu vytvoříme vektor směřující k pravému dolnímu rohu obrysu bez rotace a s pozicí ve středu scény. Na tento vektor se použije

stejná rotace, jako má náš obrys. Vysvětleme si, jak se rotace provede. 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 nám tento vektor vynásobí maticí (3), kde $\alpha = \text{line.rotation.x}$. Stejný postup se provede u `y` a `z`.

Když máme všechny potřebné body po transformaci, tak vytvoříme naše roviny jehlanu. Musíme zajistit, aby normály všech rovin směřovali 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` nám slouží pro uložení našeho jehlanu. Při sběru bodů selekce musíme v cyklu proiterovat všechny LIDARové body, a jestli jsou v našem jehlanu. Zjištujeme, jestli vzdálenost od všech rovin je kladné číslo, pokud ne, tak víme, že bod leží mimo náš jehlan. Další, co si ke všem LIDARovým bodům uložíme, je náš obrys selekce z důvodu opětovnému přehraní dat ze selekcí.

4.8 Pohyb selekcí

Než začneme, tak si musíme uvědomit, že naše obrysy jsou v prostoru scény a my se díváme na jejich projekci. Takže budeme muset zjistit za pomoci projekce na zadní rovinu frustumu kamery, jestli nějaký obrys byl protnut při této akci. Když zjistíme, že obrys byl zasáhnut, tak musíme získat všechny potřebné údaje pro budoucí práci s ním. Další se musíme rozhodnout, jakým způsobem chceme, aby se náš obrys pohyboval v trojrozměrném prostoru, a jakým způsobem budeme ovládat pohyb z dvojrozměrné obrazovky. Rozhodli jsme se, že naše obrysy musí být pohybovatelné okolo naší kamery a proto pohyb selekce musí kopírovat poloměr koule. Rotace obrysu bude záviset na hlavní kameře, ze které budeme 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.

Budeme potřebovat vytvořit události na adekvátní akce myši. K tomu budeme potřebovat použít THREE.Raycaster, který nám pomocí bodu z naší obrazovky zjistí, zda nějaký objekt je ve scéně za touto projekcí. Nejdříve nastavíme raycaster, aby použil projekční matici naší hlavní perspektivní kamery, pak použijeme metodu intersectObjects s parametrem objects, ten obsahuje všechny naše obrysy. Pokud nějaký obrys našel, tak si náš objekt uložíme 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 použijeme i při dalších událostech, pak pomocí metody intersectSphere získáme pozici středu koule, po které chceme obrys pohybovat s projekčním paprskem. 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 budeme vybírat obrys v naší scéně, tak nebudeme vždy mířit na tento bod, proto potřebuje odchylku našeho kliknutí od pevného bodu. To nám zařizuje proměnná offset, která zkopiřuje bod dotyku s koulí a odečte z něj pozici pevného bodu obrysu. Výsledkem bude vektor, který při pohybu budeme muset odčítat, abychom mohli z jakéhokoliv místa pohybovat obrysem a zachovali pohyb po chtěné kouli. Kód 15 přiřadíme 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` získáme velikost naší scény v prohlížeči a nastavíme `raycaster`, aby směřoval na pozici v prostoru, kterou získáme projekcí aktuální pozice z obrazovky. Stejně jako v předchozím příkladu použijeme proměnnou `intersection` pro získání pozice střetu s koulí po projekci našeho kurzoru na kouli, a nastavíme délku tohoto vektoru na poloměr koule. Nakonec nastavíme obrysu rotaci na základě kamery, aby obrys měl rotaci proti kameře. Tento kód 16 přiřadíme 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     selected = null;
}

```

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

Při událostech typu `mouseup`, `mouseleave` nesmíme zapomenout natočit kameru, kterou zahycujeme obrazová data selekcí 4.6. Musíme zjistit bod ve středu obrysu, na který byla použita rotační transformace. K tomu vytvoříme vektor, který bude popisovat směr ke středu obrysu bez rotace. Nasledně na tento vektor aplikujeme kompletní rotaci stejnou jako je na obrysům a přičteme k němu jeho pevný bod. Získáme pozici středu obrysu a nyní jen kameru sledující selekci nastavíme, aby sledovala tuto pozici pomocí metody `lookAt`. Nakonec vyprázdníme proměnnou `selected`. Tyto části kódu nám zařídí, že naše 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 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 částí 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 obzor, 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 geometrii, 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/FileLibrary/ResourceLibrary/TechnicalNotes/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://velodyneldar.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.* 2018. URL: <https://github.com/heremaps/pptk>.
- [9] Sean Higgins. *Velodyne cuts VLP-16 lidar price to \$4k.* 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.* 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.* 2019. URL: <https://jasonformat.com/wtf-is-javascript/>.
- [15] Ada Nduka Oyom a Ada Nduka Oyom. *Understanding the MVC pattern in Django.* 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. 2014. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/perspective-projection>.
- [18] Scratchapixel. 2014. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/visibility-problem>.
- [19] Eduard Sojka. *Pocítacova 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/>.