



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

**ZOBRAZENÍ LIDAROVÝCH, KAMEROVÝCH
A VEKTOROVÝCH DAT Z ŽELEZNIČNÍHO
MOBILNÍHO MAPOVACÍHO SYSTÉMU**

VISUALIZATION OF LIDAR, CAMERA, AND VECTOR DATA FROM A RAILWAY
MOBILE MAPPING SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ZUZANA MIŠKAŇOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ONDŘEJ KLÍMA, Ph.D.

BRNO 2025

Zadání bakalářské práce



Ústav: Ústav počítačové grafiky a multimédií (UPGM) 164356
Studentka: **Miškaňová Zuzana**
Program: Informační technologie
Název: **Zobrazení lidarových, kamerových a vektorových dat z železničního mobilního mapovacího systému**
Kategorie: Uživatelská rozhraní
Akademický rok: 2024/25

Zadání:

1. Seznamte se možnostmi vizualizace obrazových a geometrických dat z mobilních mapovacích systémů.
2. Navrhněte uživatelský systém pro vizualizaci agregovaných dat ze senzorů umístěných na čele vlaku včetně vektorových mapových dat.
3. Navržený systém implementujte v podobě uživatelské aplikace v prostředí Python s využitím existujících nástrojů a knihoven.
4. Proveďte experimenty s dodanými daty a vyhodnoťte vlastnosti a uživatelskou přívětivost aplikace.
5. Prezentujte dosažené výsledky.

Literatura:

Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:

Body 1, 2 a částečně 3.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Klíma Ondřej, Ing., Ph.D.**

Vedoucí ústavu: Černocký Jan, prof. Dr. Ing.

Datum zadání: 1.11.2024

Termín pro odevzdání: 14.5.2025

Datum schválení: 12.11.2024

Abstrakt

Cieľom tejto práce je vytvoriť systém pre vizualizáciu dát z železničného mobilného mapovacieho systému, ktorý dátu zobrazí praktickým, prehľadným a prispôsobiteľným spôsobom. Tento systém bol implementovaný ako webová aplikácia s využitím programovacieho jazyka Python a frameworku Dash. Pre optimalizáciu výkonu prebieha vykreslovanie mračna bodov na strane klienta pomocou frameworku deck.gl. Vyvinutá aplikácia poskytuje do-statočnú funkcionality a výkonnosť. Avšak stále by boli možné ďalšie zlepšenia, najmä čo sa týka prispôsobiteľnosti. Táto nová webová aplikácia má potenciál byť využitá v oblasti vizualizácie dát z mobilných mapovacích systémov. Znalosti získané počas tvorby práce by tiež mohli byť užitočné pre ďalších vývojárov, ktorí vytvárajú systémy pre vizualizáciu dát.

Abstract

The aim of this thesis is to create a system visualizing data from a railway mobile mapping system, which will display data in a practical, clear and customizable way. The system was implemented as a web application using programming language Python and framework Dash. For performance optimization, the rendering of the point cloud and vector data is done on the client's side using framework deck.gl. The created application offers sufficient functionality and performance. However, there is still room for further improvement, mainly in customizability. This new web application has a potential to be exploited within the field of mobile mapping system data visualizations. The knowledge acquired throughout the work could also be useful to other developers creating visualization systems.

Kľúčové slová

vizualizácia dát, mobilný mapovací systém, lidar, mračno bodov, deck.gl, webové aplikácie, Dash

Keywords

data visualization, mobile mapping system, lidar, point cloud, deck.gl, web applications, Dash

Citácia

MIŠKAŇOVÁ, Zuzana. *Zobrazení lidarových, kamerových a vektorových dat z železničného mobilného mapovacieho systému*. Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Klíma, Ph.D.

Zobrazení lidarových, kamerových a vektorových dat z železničního mobilního mapovacího systému

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Ing. Ondřeja Klímu, Ph.D. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Zuzana Miškaňová
3. mája 2025

Poděkování

Ďakujem môjmu vedúcemu Ing. Ondřejovi Klímovi, Ph.D. za poskytnutú pomoc a cenné rady, ktoré mi v priebehu tvorby práce výrazne pomohli.

Obsah

1	Úvod	2
2	Vizualizácia dát z mobilných mapovacích systémov	3
2.1	Dierkový model kamery	3
2.2	Skreslenie kamery	6
2.3	Vykresľovací reťazec	8
2.4	Framework deck.gl a jeho nadstavba Pydeck	12
2.5	Nastavenie polohy kamery vo frameworku deck.gl	13
2.6	Frameworky pre tvorbu webových aplikácií	16
3	Návrh aplikácie	19
3.1	Návrh používateľského rozhrania	20
3.2	Problém nahrávania dát do aplikácie	20
4	Implementácia navrhnutej aplikácie	23
4.1	Popis modulov	23
4.2	Rozdelenie dát do vrstiev	26
4.3	Implementácia animácie pohybu vlaku	28
4.4	Implementácia skreslenia	29
5	Testovanie a evaluácia	31
5.1	Optimalizácia vykreslovania mračna bodov	31
5.2	Výkonnosť	35
5.3	Vyhodnotenie používateľskej prívetivosti	35
6	Záver	36
	Literatúra	37

Kapitola 1

Úvod

Kapitola 2

Vizualizácia dát z mobilných mapovacích systémov

Výstupom mobilného mapovacieho systému je množstvo dát z rôznych senzorov. Dáta, ktoré boli poskytnuté k tvorbe tejto práce, obsahovali v prvom rade mračná bodov získané z lidaru. Aby bolo možné s týmito mračnami bodov pracovať a vhodne ich zobrazovať, je potrebné, aby mobilný mapovací systém zaznamenával údaje o svojej polohe. V tomto prípade boli k dispozícii údaje o transláciách a rotáciách kamery s časovými razítkami. Ďalej býva súčasťou mobilného mapovacieho systému klasická kamera. Pre zobrazenie mračna bodov takým spôsobom, aby sa zobrazenie čo najviac zhodovalo s kamerovým záznamom, je nutné poznáť parametre kamery – kalibračnú maticu a prípadne aj parametre skreslenia.

Pri vývoji aplikácie, ktorá má umožniť vizualizáciu týchto dát a navyše aj ďalších vektorových dát, je nutné poznáť základné princípy zobrazovania 3D dát v počítačovej grafike. Ďalej je potrebné vybrať si a naštudovať technológie, pomocou ktorých bude aplikácia vytvorená, a to konkrétnie niektorý z frameworkov pre tvorbu aplikácií v jazyku Python a vhodný framework pre zobrazenie grafických dát. Všetko toto je predmetom tejto kapitoly.

2.1 Dierkový model kamery

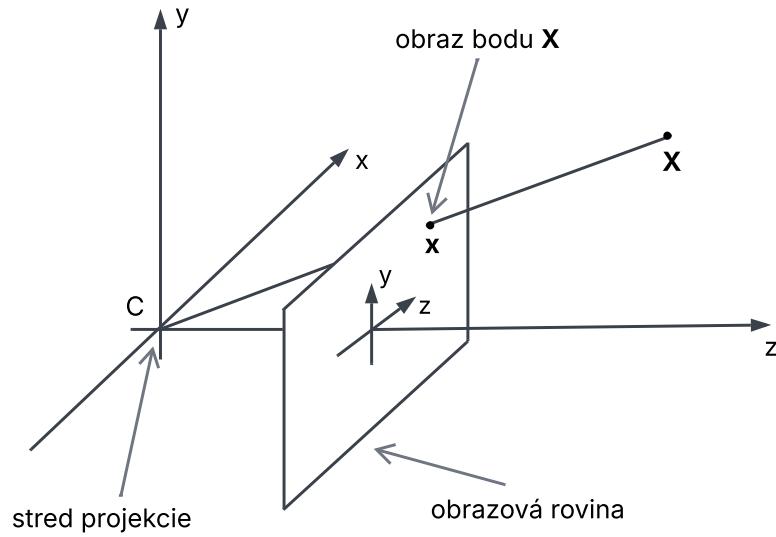
Zdrojom všetkých informácií, ktoré budú uvedené v tejto sekcii, je kniha *Multiple View Geometry in Computer Vision* od autorov R. Hartley a A. Zisserman [7].

Kamera je v oblasti počítačovej grafiky a počítačového videnia pojem, ktorý označuje projekciu bodov z trojrozmerného priestoru do roviny. Túto projekciu je možné vyjadriť pomocou matíc a je väčšinou bodová (*central projection*).

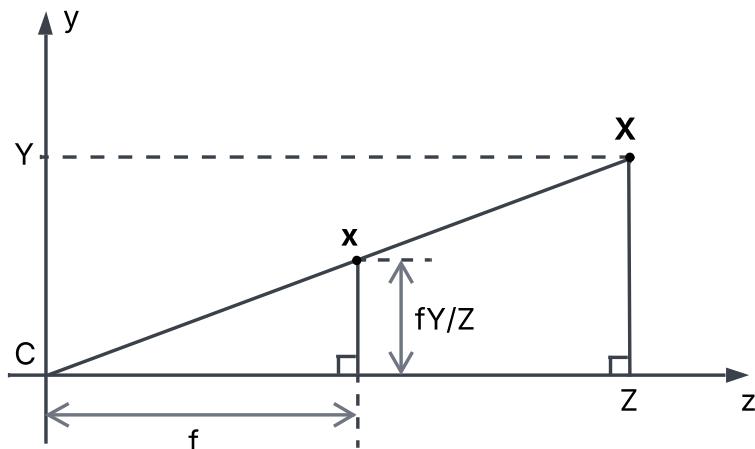
Existuje niekoľko rôznych modelov kamery, z ktorých najjednoduchší je dierkový model kamery (*pinhole camera model*). U tohto modelu je stred projekcie **C** (*camera centre*) v počiatku Euklidovského súradnicového systému a body sa premietajú do roviny $z = f$, ktorá sa označuje ako obrazová rovina (*image plane*).

Princíp zobrazenia je nasledovný: obrazom bodu $\mathbf{X} = (X, Y, Z)^T$ je bod \mathbf{x} , kde priamka vedúca bodom \mathbf{X} a stredom projekcie **C** pretína obrazovú rovinu (obrázok 2.1). Z podobnosti trojuholníkov je možné odvodiť, že bod \mathbf{x} má súradnice $(fX/Z, fY/Z, f)^T$, postup je naznačený na obrázku 2.2.

Kedže všetky obrazy bodov ležia v obrazovej rovine $z = f$, je možné poslednú súradnicu zanedbať a zapisovať súradnice bodu \mathbf{x} v súradnicovom systéme obrazovej roviny ako $(fX/Z, fY/Z)^T$.



Obr. 2.1: Zakladný princíp dierkového modelu kamery [7].



Obr. 2.2: Náčrt odvodenia súradníc bodu x [7].

Túto projekciu môžeme v homogénnych súradničiach zapísť pomocou násobenia matíc nasledovným spôsobom:

$$\mathbf{x} = \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 \\ f & 0 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{bmatrix} f & 0 \\ f & 0 \\ 1 & 0 \end{bmatrix} \mathbf{X} = \mathbf{P}\mathbf{X}$$

Maticu P nazývame projekčnou maticou kamery (*camera projection matrix*).

Rozšírenia základného dierkového modelu kamery

V praxi väčšinou chceme vyjadriť body na obrazovej rovine v súradnicovom systéme, ktorý nemá stred v bode $(0, 0, f)^T$, ale v ľubovoľnom bode $(-p_x, -p_y, f)^T$ (obrázok 2.3). Obrazom bodu $\mathbf{X} = (X, Y, Z)^T$ (v súradnicovom systéme kamery) potom bude bod $\mathbf{x} = (fX/Z + p_x, fY/Z + p_y, f)^T$ (v súradnicovom systéme obrazovej roviny). To je možné zahrnúť do projekčnej matice nasledovným spôsobom:

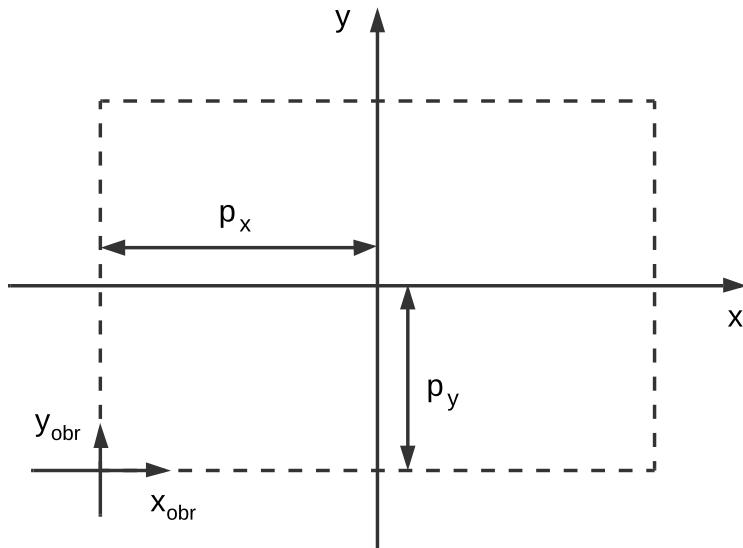
$$P = \begin{bmatrix} f & p_x & 0 \\ f & p_y & 0 \\ 1 & 0 & \end{bmatrix}$$

$$\mathbf{x} = \begin{pmatrix} fX/Z + p_x \\ fY/Z + p_y \\ 1 \end{pmatrix} = \begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{bmatrix} f & p_x & 0 \\ f & p_y & 0 \\ 1 & 0 & \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Maticu

$$K = \begin{bmatrix} f & p_x \\ f & p_y \\ 1 & \end{bmatrix}$$

nazývame kalibračnou maticou kamery (*camera calibration matrix*) a medzi ňou a projekčnou maticou platí vzťah $P = [K|0]$, kde 0 predstavuje nulový stĺpcový vektor.



Obr. 2.3: Súradnicový systém so stredom v bode $(-p_x, -p_y, f)^T$ obrazovej roviny [7].

U niektorých kamier je možné, že súradnicový systém obrazovej roviny nemá rovnakú mierku na oboch osiach. Vtedy sú v kalibračnej matici namiesto parametra f parametre f_x , f_y a matica má podobu

$$K' = \begin{bmatrix} f_x & p_x \\ f_y & p_y \\ 1 & \end{bmatrix}.$$

Parametre p_x , p_y a f , prípadne f_x a f_y označujeme ako **vnútorné parametre kamery**.

Až doteraz sme predpokladali, že kamera má stred v počiatku súradnicovej sústavy a je „otočená“ v smere osi z , t. j. že obrazová rovina je rovnobežná s rovinou xy . V praxi to tak však nebýva a kamera má v súradnicovom systéme, v ktorom sú definované zobrazované body, istú rotáciu a transláciu. V takom prípade rozlišujeme všeobecný súradnicový systém a súradnicový systém kamery (*world coordinate frame* a *camera coordinate frame*).

Ak $\tilde{\mathbf{X}}$ je vektor súradníc bodu \mathbf{X} vo všeobecnom súradnicovom systéme a vektor $\tilde{\mathbf{X}}_{\text{cam}}$ reprezentuje ten istý bod v súradnicovom systéme kamery, tak platí vzťah

$$\tilde{\mathbf{X}}_{\text{cam}} = \mathbf{R}(\tilde{\mathbf{X}} - \tilde{\mathbf{C}}),$$

kde $\tilde{\mathbf{C}}$ sú súradnice stredu kamery vo všeobecnom súradnicovom systéme a \mathbf{R} je rotačná matica 3×3 reprezentujúca orientáciu súradnicového systému kamery. To vedie k novému vyjadreniu projekčnej matice ako $\mathbf{P} = \mathbf{K}\mathbf{R}[\mathbf{I}] - \tilde{\mathbf{C}}$, kde \mathbf{I} je jednotková matica 3×3 .

Parametre \mathbf{R} a $\tilde{\mathbf{C}}$ označujeme ako **vonkajšie parametre kamery**.

2.2 Skreslenie kamery

Dierkový model kamery zodpovedá spôsobu, akým zachytávajú obraz reálne kamery, iba približne, ale nie presne. Obraz z reálnej kamery má zvyčajne navýše ešte skreslenie.

Existujú rôzne modely skreslenia, z ktorých najčastejšie používané sú radiálne skreslenie (*radial distortion*) a tangenciálne skreslenie (*tangential/decentering distortion*) [16].

Radiálne skreslenie vzniká v dôsledku rozdielneho lomu lúčov v strede a na okraji šošovky a je tým väčšie, čím menšia je šošovka. Príklad radiálneho skreslenia je na obrázku 2.4. Tangenciálne skreslenie vzniká vtedy, keď šošovka nie je rovnobežná s obrazovou rovinou [18]. Reálne kamery zvyčajne majú radiálne skreslenie väčšie než tangenciálne [9].

Radiálne skreslenie sa počíta vzťahmi

$$\begin{aligned} x_{\text{distorted}} &= x(1 + k_1r^2 + k_2r^4 + k_3r^6) \\ y_{\text{distorted}} &= y(1 + k_1r^2 + k_2r^4 + k_3r^6) \end{aligned}$$

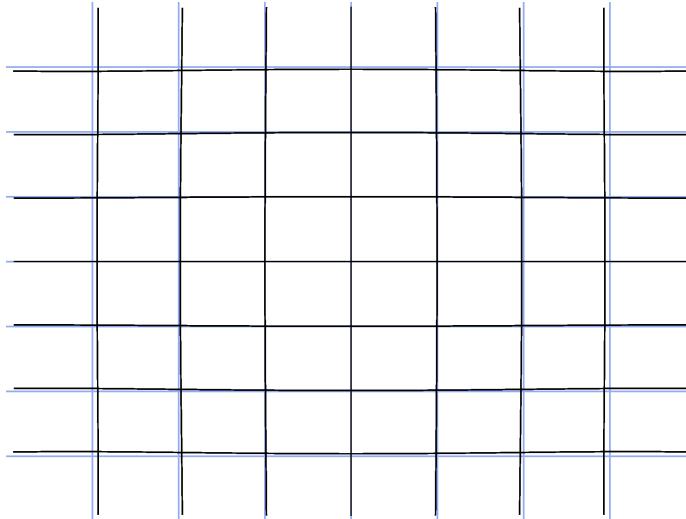
kde

- x, y sú normalizované obrazové súradnice (*normalized image coordinates*) bodu pred skreslením,
- k_1, k_2, k_3 sú koeficienty radiálneho skreslenia,
- $r^2 = x^2 + y^2$ [8].

Normalizovaný obrazový súradnicový systém je súradnicový systém, ktorý má tieto vlastnosti:

- Jeho stred je v strede obrazu,
- Väčšia dimenzia obrazu má veľkosť 1.

Teda napríklad obraz s pomerom strán 4:3 bude v normalizovanom obrazovom súradnicovom systéme obdĺžnik s ľavým dolným rohom v bode $[-0.5, 3/4 \times (-0.5)]$ a pravým horným rohom v bode $[0.5, 3/4 \times 0.5]$.



Obr. 2.4: Radiálne skreslenie ilustrované na mriežke. Bledomodrá mriežka je pôvodná, čierna skreslená. Pri tvorbe obrázku boli použité skutočné parametre radiálneho skreslenia kamery mobilného mapovacieho systému, ktoré boli súčasťou dát dodaných k tvorbe tejto práce: $k_1 = -0.183217$, $k_2 = 0.026917$, $k_3 = 0.335446$

Prevod zo súradníc pixelu v obrazu s rozlíšením $w \times h$ pixelov do normalizovaných obrazových súradníc je nasledovný [12]:

$$x_{norm} = \frac{x_{pix} - \frac{w-1}{2}}{\max(w, h)}, \quad y_{norm} = \frac{y_{pix} - \frac{h-1}{2}}{\max(w, h)}.$$

Tangenciálne skreslenie sa počíta vzťahmi

$$\begin{aligned} x_{distorted} &= x + [2p_1xy + p_2(r^2 + 2x^2)] \\ y_{distorted} &= y + [p_1(r^2 + 2y^2) + 2p_2xy] \end{aligned}$$

kde

- x, y, r^2 sú rovnaké ako u radiálneho skreslenia,
- p_1, p_2 sú koeficienty tangenciálneho skreslenia [8].

Vzťahy pre výpočet radiálneho a tangenciálneho skreslenia zároveň sú nasledovné [16]:

$$\begin{aligned} x_{distorted} &= x(1 + k_1r^2 + k_2r^4 + k_3r^6) + [2p_1xy + p_2(r^2 + 2x^2)] \\ y_{distorted} &= y(1 + k_1r^2 + k_2r^4 + k_3r^6) + [p_1(r^2 + 2y^2) + 2p_2xy] \end{aligned}$$

Koeficienty skreslenia sa zvyčajne udávajú ako päťica [8]

$$(k_1, k_2, p_1, p_2, k_3).$$

Proces zisťovania parametrov kamery, vrátane koeficientov skreslenia, sa nazýva kalibrácia kamery (*camera calibration*) [18].

2.3 Vykresľovací reťazec

Proces vykreslovania trojrozmerných objektov na dvojrozmernú obrazovku v počítačovej grafike pozostáva z viacerých transformácií. Vykreslované objekty sú postupne prevádzané cez niekoľko súradnicových systémov pomocou transformačných matíc.

Abstraktný model, ktorý popisuje sériu krokov potrebných k zobrazeniu trojrozmernej scény zloženej z viacerých objektov, sa nazýva vykresľovací reťazec (*graphics/rendering pipeline*). Vykresľovací reťazec zahŕňa popis prenosu dát na grafickú kartu, výpočet finálnej polohy objektu na obrazovke, rasterizáciu (určenie, ktoré pixely zodpovedajú vektorovo popísaným objektom) a výpočet farby pixelov [15].

Vykresľovací reťazec môže mať rôzne podoby. V tejto sekcii je popísaná verzia, ktorú používa knižnica OpenGL, pretože, ako bude popísané v sekcií 2.4, pri implementácii tejto práce bol použitý framework, ktorý používa rozhranie WebGL, ktoré je založené práve na knižnici OpenGL [17].

Vykreslované objekty sa typicky skladajú z bodov (*vertices*) pospájaných hranami do trojuholníkov [15]. Táto sekcia sa bude zaoberať iba popisom transformácie bodov z lokálneho súradnicového systému do finálnej polohy vo vykreslenom obraze.

Na začiatku vykreslovacieho reťazca sú body v lokálnom súradnicovom systéme (*local/object space*, každý objekt má vlastný súradnicový systém), s homogénnymi súradnicami v tvare $V_{local} = (x, y, z, 1)^T$. Potom prechádzajú transformáciami, ktoré sa počítajú násobením súradníc bodu maticami a sú znázornené na obrázku 2.5. Sú to nasledujúce transformácie:

1. Modelová transformácia (*model transformation*). Každý objekt vo vykreslovanej scéne má vlastnú modelovú maticu M_{model} (*model matrix*), ktorá popisuje jeho pozíciu, orientáciu a veľkosť v scéne. Po modelovej transformácii sú súradnice bodov v súradnicovom systéme scény (*world space*) [5, 15].
2. Pohľadová transformácia (*view transformation*). Prevádzza body zo súradnicového systému scény do súradnicového systému kamery (*view/camera/eye space*). Zodpovedá jej pohľadová matica M_{view} (*view matrix*), ktorá popisuje polohu a orientáciu kamery v súradnicovom systéme scény [5, 15].
3. Projekcia (*projection transformation*). Je buď ortografická, alebo perspektívna. Prevádzza body zo súradnicového systému kamery do orezávacieho súradnicového systému (*clip space*). Určuje, ktoré body scény budú viditeľné a ktoré nie, pretože sa zobrazia iba body, ktoré budú mať v orezávacom súradnicovom systéme všetky súradnice v intervale $(-1, 1)$ [15]. Týmto súradniciam sa hovorí aj *normalized device coordinates* (NDC). Oblast v súradnicovom systéme scény, ktorá bude vo finálnom obraze viditeľná, sa nazýva *frustum*. U ortografickej projekcie má tvar kvádra a u perspektívnej projekcie má tvar zrezaného ihlana [5], čo ilustruje obrázok 2.6. Zodpovedá jej projekčnej matici (*projection matrix*), ktorá má tvar

$$M_{projection} = \begin{pmatrix} k_1 & 0 & k_2 & 0 \\ 0 & k_3 & k_4 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix},$$

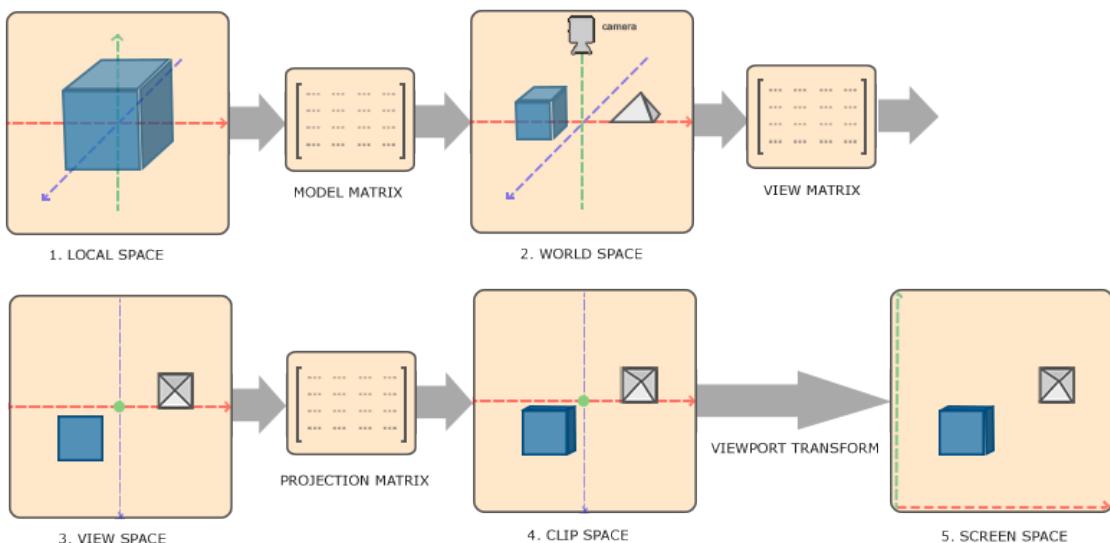
kde n a f sú vzdialosti prednej a zadnej orezávacej rovniny (*near plane* a *far plane*, znázornené na obrázku 2.6) [1].

Po vynásobení súradníc bodov projekčnou maticou je potrebné ešte vydeliť ich súradnice v tvare $(x, y, z, w)^T$ homogénnou zložkou, čím vznikne tvar $(x/w, y/w, z/w, 1)^T$ aby bolo možné zistiť, či sú všetky súradnice v intervale $(-1, 1)$, a teda či bude bod súčasťou výsledného obrazu. Táto operácia sa nazýva perspektívne delenie (*perspective division*) [5].

4. Transformácia do súradnicového systému obrazu (*viewport transform*). Prevádzza body z orezávacieho súradnicového systému do súradnicového systému finálneho obrazu (*screen space*) [5]. Matica, ktorá vykonáva túto transformáciu, sa nazýva *viewport transform matrix* a má tvar

$$M_{viewport} = \begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

kde w a h je výška a šírka obrazu v pixeloch [2].



Obr. 2.5: Transformácie medzi rôznymi súradnicovými systémami, ktoré sú súčasťou vykresľovacieho retazca [5]. [PREKRESLIŤ VEKTOROVY]

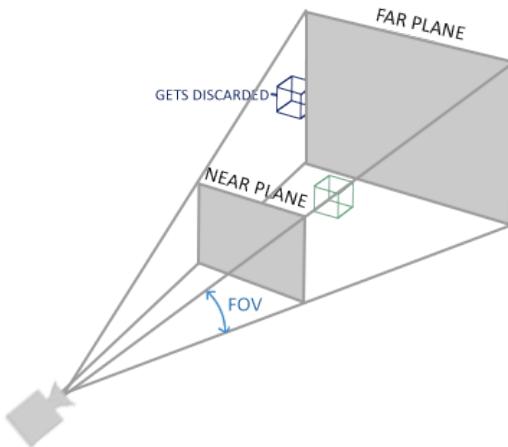
Pozíciu bodu V_{local} vo finálnom vykreslenom obraze scény je teda možné matematicky vyjadriť ako súčin

$$M_{viewport} \cdot M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}.$$

Transformácia z lokálneho súradnicového systému do NDC prebieha na programovateľnej časti grafickej karty, ktorá sa nazýva *vertex shader* [5].

Vzťah vykresľovacieho reťazca k dierkovému modelu kamery

Vykresľovací retazec a dierkový model kamery sú modely, ktoré sú si podobné, pretože sa oba zaoberajú transformáciou trojrozmernej scény do dvojrozmerného obrazu. Používajú sa však



Obr. 2.6: *Frustum* – oblasť v súradnicovom systéme scény, ktorá bude vo finálnom obraze viditeľná – u perspektívnej projekcie [5]. [PREKRESLIŤ VEKTOROVО]

v odlišných oblastiach – dierkový model kamery sa typicky používa v teórii počítačového videnia a vykreslovací reťazec sa týka počítačovej grafiky a zobrazovania trojrozmerných modelov.

Rozdiely a podobnosti oboch modelov je možné zhrnúť týmito bodmi:

- Dierkový model kamery nemá nič zodpovedajúce modelovej transformácii. Zobrazované body sú už od začiatku v rovnakom súradnicovom systéme ako kamera.
- Pohľadová transformácia zodpovedá vonkajším parametrom kamery.
- Vnútorné parametre kamery zodpovedajú projekcii a transformácii do súradnicového systému obrazu. Vykreslovací reťazec má špeciálny medzistupeň - orezávací súradnicový systém, ktorý dierkový model kamery nemá.
- Dierkový model kamery je jednoduchší, pretože na rozdiel od vykreslovacieho reťazca nemá prednú a zadnú orezávaciu rovinu a nerieši prekrývanie objektov. Vykreľovací reťazec ponecháva aj vo finálnych súradničiach bodu v obraze hodnotu z , teda hĺbkou, aby u prekrývajúcich sa objektov bolo možné zistiť, ktorý z nich je v popredí. Dierkový model kamery nič také nezahrňa.

Za poznámku stojí, že v oboch modeloch sa vyskytuje termín *projekčná matica*, ale ide o dve rôzne matice.

V rámci tejto práce bolo potrebné skombinovať znalosti z oboch oblastí, pretože súčasťou dodaných dát bola kalibračná matica kamery, pomocou ktorej mali byť dátá zobrazené, ale použitý framework si vyžadoval zadat projekčnú maticu (tú z vykreslovacieho reťazca).

Problém sa dá formulovať nasledovne: je známa projekčná matica (tá z dierkového modelu kamery), ktorá prevádzka body zo súradnicového systému kamery priamo do súradnicového systému obrazu, pričom nepočíta hodnotu hĺbky. Je potrebné vypočítať projekčnú maticu, ktorá bude prevádzkať body zo súradnicového systému kamery do orezávacieho súradnicového systému tak, aby následne po automatickom prevode do súradnicového systému obrazu výsledný obraz zodpovedal zadanej projekčnej matici. Rozmery $w \times h$ výsledného obrazu sú známe.

Riešenie je možné matematicky odvodiť. Známa projekčná matica, ktorá prevádzza body priamo do súradnicového systému obrazu, má tvar

$$P = \begin{pmatrix} f_x & 0 & p_x & 0 \\ 0 & f_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Potrebujeme nájsť maticu $M_{projection}$ takú, aby platila rovnosť

$$M_{viewport} \cdot M_{projection} \cdot V_{view} = P \cdot V_{view},$$

teda parametre k_1, k_2, k_3 a k_4 také, aby platila rovnosť

$$\begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} k_1 & 0 & k_2 & 0 \\ 0 & k_3 & k_4 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & p_x & 0 \\ 0 & f_y & p_y & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Otázniky pridané do matice P zastupujú výpočet súradnice z , ktorý pôvodná matica neobsahovala, ale pri vykreslovaní trojrozmernej scény v počítačovej grafike je nutný.

Rovnosť zjednodušíme

$$\begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} k_1x + k_2z \\ k_3y + k_4z \\ \frac{-(f+n)}{f-n}z + \frac{-2fn}{f-n} \\ -z \end{pmatrix} = \begin{pmatrix} f_x x + p_x z \\ f_y y + p_y z \\ ?z+? \\ z \end{pmatrix},$$

vykonáme delenie homogénnou súradnicou

$$\begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -k_1 \frac{x}{z} - k_2 \\ -k_3 \frac{y}{z} - k_4 \\ \frac{(f+n)}{f-n} + \frac{2fn}{z(f-n)} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x \frac{x}{z} + p_x \\ f_y \frac{y}{z} + p_y \\ ? + \frac{?}{z} \\ 1 \end{pmatrix}$$

a znova zjednodušíme

$$\begin{pmatrix} \frac{w}{2}(-k_1 \frac{x}{z} - k_2) + \frac{w}{2} \\ \frac{h}{2}(-k_3 \frac{y}{z} - k_4) + \frac{h}{2} \\ \frac{f-n}{2}(\frac{(f+n)}{f-n} + \frac{2fn}{z(f-n)}) + \frac{f+n}{2} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x \frac{x}{z} + p_x \\ f_y \frac{y}{z} + p_y \\ ? + \frac{?}{z} \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} -k_1 \frac{w}{2} \frac{x}{z} - k_2 \frac{w}{2} + \frac{w}{2} \\ -k_3 \frac{h}{2} \frac{y}{z} - k_4 \frac{h}{2} + \frac{h}{2} \\ f + n + \frac{f+n}{z} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x \frac{x}{z} + p_x \\ f_y \frac{y}{z} + p_y \\ ? + \frac{?}{z} \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} -k_1 \frac{w}{2} \frac{x}{z} + \frac{w}{2}(1 - k_2) \\ -k_3 \frac{h}{2} \frac{y}{z} + \frac{h}{2}(1 - k_4) \\ f + n + \frac{f+n}{z} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x \frac{x}{z} + p_x \\ f_y \frac{y}{z} + p_y \\ ? + \frac{?}{z} \\ 1 \end{pmatrix}.$$

Z toho vyplýva, že parametre n a f je možné si zvoliť ľubovoľne, pretože ich zadaná matica nedefinuje, a že musí platiť

$$\begin{aligned} -k_1 \frac{w}{2} &= f_x, & \frac{w}{2}(1 - k_2) &= p_x, \\ -k_3 \frac{h}{2} &= f_y, & \frac{h}{2}(1 - k_4) &= p_y, \end{aligned}$$

a teda sme dospeli k hľadanému vyjadreniu parametrov k_1, k_2, k_3 a k_4

$$\begin{aligned} k_1 &= -\frac{2f_x}{w}, & k_2 &= 1 - \frac{2p_x}{w}, \\ k_3 &= -\frac{2f_y}{h}, & k_4 &= 1 - \frac{2p_y}{h}. \end{aligned}$$

Hľadaná projekčná matica má teda tvar

$$M_{projection} = \begin{pmatrix} -\frac{2f_x}{w} & 0 & 1 - \frac{2p_x}{w} & 0 \\ 0 & -\frac{2f_y}{h} & 1 - \frac{2p_y}{h} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

2.4 Framework deck.gl a jeho nadstavba Pydeck

Pri vývoji webovej aplikácie pre vizualizáciu väčšieho množstva dát, u ktorej má vykreslovanie prebiehať na strane klienta, teda vo webovom prehliadači, sa hodí priamo či nepriamo použiť niektorý z frameworkov pre zobrazovanie dát v jazyku JavaScript.

Takým vhodným frameworkom je napríklad deck.gl, ktorý je určený na zobrazovanie veľkých sád dát. Je zameraný najmä na zobrazovanie geografických dát mapových podkladov, ale hodí sa aj na iné typy dát. Vyznačuje sa vysokou presnosťou a výkonnosťou. Pre akceleráciu využíva rozhrania WebGPU a WebGL2 [10].

Vizualizácia dát v deck.gl sa skladá z dvoch základných častí:

- **Vrstvy (Layers).** Do vrstiev sa ukladajú zobrazované dátá. Framework deck.gl ponúka vyše 30 preddefinovaných typov vrstiev, ktoré zodpovedajú rôznym často sa vyskytujúcim typom dát. Pre túto prácu je významná najmä vrstva **PointCloudLayer**, ktorá je určená na zobrazenie mračna bodov, a vrstva **PathLayer**, ktorá je určená na zobrazenie trás.
- **Pohľad (View).** Definuje vlastnosti kamery, napríklad zorné pole a prednú a zadnú orezávaciu rovinu (*near plane* a *far plane*). Je možné buď určiť všetky tieto vlastnosti osobitne, alebo rovno zadať vypočítanú projekčnú maticu. Časť **ViewState** určuje polohu a orientáciu kamery, pričom u orientácie je možné určiť iba dva uhly (*bearing* a *pitch*). Typ pohľadu definuje spôsob interakcie vizualizácie s používateľom, napríklad pre zobrazenie trate z pohľadu strojvedúceho je ideálny typ **FirstPersonView**.

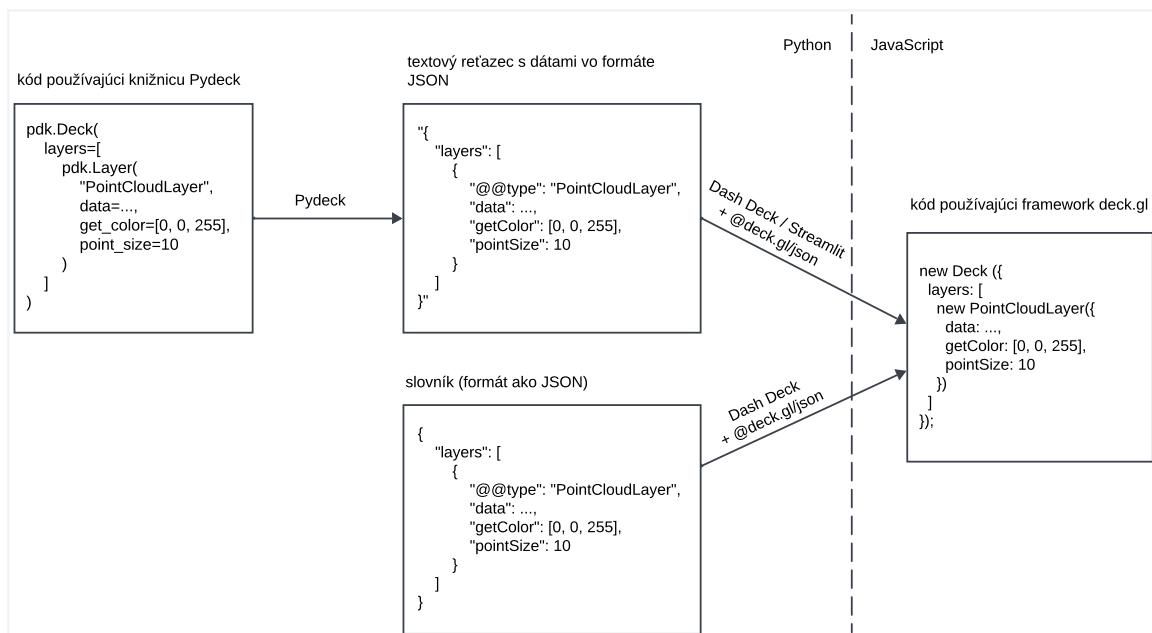
To, že sa deck.gl na vizualizáciu dát z mobilného mapovacieho systému naPozaj hodí, dokazujú ukážky na jeho webových stránkach. Použiteľnosť vrstvy **PointCloudLayer** demonštruje plynulá animácia mračna vyše 800 000 bodov.

Na webových stránkach deck.gl sa nachádza aj galéria projektov vytvorených pomocou tohto frameworku. Medzi nimi je Autonomous Visualization System, toolkit pre vývoj webových aplikácií pre vizualizáciu dát z autonómnych vozidiel [20]. Na webových stránkach

tohto projektu je ukážka takej aplikácie, ktorá zobrazuje vozidlo, dátá z lidaru, vektorové dátá, ako napríklad trasu vozidla, a záznam z kamery. To je veľmi podobné aplikácii vyvíjanej v rámci tejto práce, ale je tu niekoľko rozdielov. Poprvé, mračno bodov nie je agregované, a teda sa v každom kroku animácie zobrazuje iba v tom momente nasnímaná časť, a podruhé, kamerový záznam sa zobrazuje oddelené, a nie na pozadí mračna bodov a vektorových dát.

Hoci je framework deck.gl primárne určený pre použitie v Javascripte, je možné ho použiť aj v jazyku Python, a to pomocou knižnice **Pydeck**. Tá je pomerne jednoduchá a podstatou jej činnosti je, že prevedie kód napísaný v jazyku Python do formátu JSON. Framework deck.gl má totiž modul `@deck.gl/json`, ktorý prijíma reprezentáciu vizualizácie vo formáte JSON a transformuje ju do javascriptového kódu (na definície funkcií a `deck.gl` objektov)¹.

Knižnica Pydeck je dobrým prostriedkom na vytvorenie jednoduchých vizualizácií, s ktorými môže používateľ interagovať pohybmi myši. Jej možnosti sú však oproti pôvodnému frameworku deck.gl veľmi obmedzené. Nie je vhodná na vytváranie zložitejších animácií s veľkým množstvom dát, pretože sa aj po tej najmenšej zmene musia dátá a definícia vizualizácie nanovo prevádztať do formátu JSON a následne na javascriptový kód (obrázok 2.7), čo je veľmi časovo náročné.



Obr. 2.7: Schéma vzťahov medzi technológiami Pydeck, Dash Deck a deck.gl a transformácií, ktorými prechádza definícia zobrazenia.

2.5 Nastavenie polohy kamery vo frameworku deck.gl

Súčasťou dodaných dát z mobilného mapovacieho systému boli rotácie a translácie kamery. Pre nastavenie polohy vo frameworku deck.gl existujú dve možnosti:

¹Ukážka rozhrania modulu `@deck.gl/json` je na <https://deck.gl/playground>.

1. Aplikovať transformáciu na dátu a nechať kameru v počiatku súradnicového systému, prípadne v nejakom inom pevnom bode.
2. Nechať dátu v pôvodnom stave a aplikovať všetky transformácie iba na kameru.

Je zrejmé, že pre dosiahnutie rovnakého výsledku musí byť transformácia použitá v druhom prípade inverzná k tej, ktorá je použitá v tom prvom.

Bližším popisom a zhodnotením výhod a nevýhod týchto dvoch metód sa zaobrajú nasledujúce dve podsekcie.

Aplikovanie transformácií na dátu

Ako bolo zmienené v sekcií 2.4, dátu sa vo frameworku deck.gl členia do vrstiev. Každej vrstve je potom potrebné priradiť pole s dátami a definovať pre ňu takzvané prístupové funkcie (*data accessors*), ktoré určujú, akým spôsobom sa z pola s dátami získa poloha a farba prvku. Napríklad u vrstvy **PointCloudLayer** je potrebné definovať funkciu `getPosition` pre získanie polohy bodu a `getColor` pre výpočet farby bodu [10].

Práve vo funkcií `getPosition` je možné aplikovať na polohu bodu ľubovoľnú transformáciu.

Napríklad v prípade, že je potrebné posunúť kameru v mračne bodov o 10 jednotiek v kladnom smere po osi x, by bolo možné vykonať túto transformáciu pomocou funkcie `getPosition` tak, že by sa posunul každý bod po osi x o 10 jednotiek v zápornom smere. V prípade, že by bol v poli dát každý bod vo formáte [x, y, z, intenzita], by potom funkcia `getPosition` vyzerala takto:

```
function getPosition(d) {
  return [d[0] - 10, d[1], d[2]];
}
```

Výhodou tohto prístupu je, že je možné vykonať ľubovoľnú transformáciu, pretože do funkcie možno napísať akýkoľvek kód.

Veľkou nevýhodou je časová náročnosť, pri zmene polohy kamery sa totiž musí prepočítať poloha každého bodu. Tieto výpočty sa vykonávajú na procesore a výsledky sa následne nahrávajú na grafickú kartu. To konštatuje aj samotná dokumentácia frameworku deck.gl, kde sa píše, že kľúčom k tvorbe výkonných aplikácií je minimalizácia aktualizácií vrstiev, pri ktorých dochádza k prepočítavaniu dát a ich opäťovnému nahrávaniu na grafickú kartu. Taktiež je tam uvedené, že prístupové funkcie by mali byť čo najtriviálnejšie, pretože sa počítajú pre každú položku v dátach, t. j. pre každý bod v mračne bodov, a teda sa každé pridanie operácií naviac výrazne prejaví na výkonnosti. 99% procesorového času venovaného aktualizácií dát vrstvy sa strávi práve volaním prístupových funkcií [11].

Aplikovanie transformácií na kameru

Meniť polohu a orientáciu kamery je v deck.gl možné pomocou troch parametrov: `position`, `bearing` a `pitch` [10]. To sú rotácie iba podľa dvoch osí, a teda nie je možné dosiahnuť všeobecnú rotáciu – chýba možnosť nastaviť rotáciu okolo tretej osi, takzvaný *roll* uhol.

V dokumentácii deck.gl je popísaná aj trieda **Viewport**, u ktorej je možné namiesto parametrov `position`, `bearing` a `pitch` nastaviť maticu pohľadu `viewMatrix`, čo znamená možnosť použiť akýkoľvek rotáciu. Problém je však v tom, že sa v dokumentácii už nikde nepíše, ako túto triedu použiť. Ide teda o nezrovnalosť, pravdepodobne pozostatok

z predchádzajúcich verzií `deck.gl`, kde táto možnosť bola, ale medzičasom bola zrušená. V aktuálnej verzii je možné túto maticu už iba prečítať, ale nie nahradíť vlastnou.

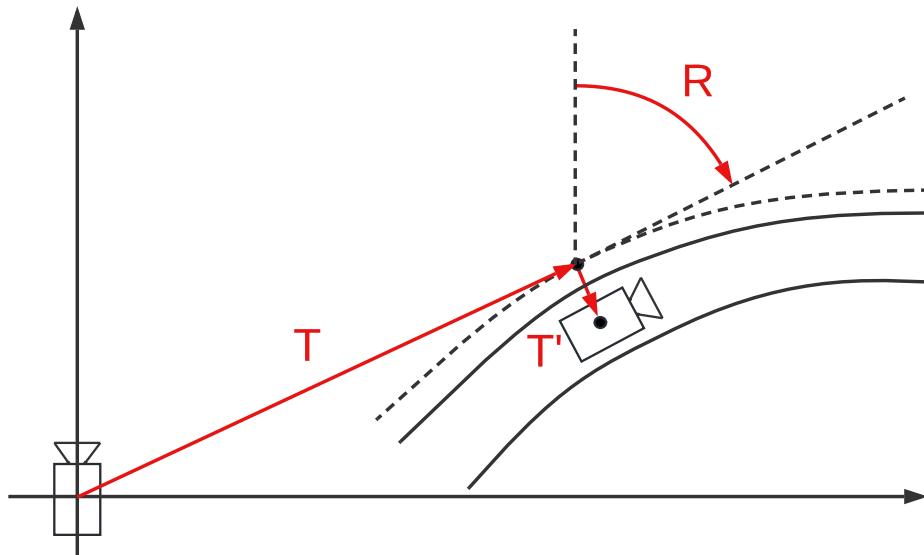
Nevýhodou tohto spôsobu nastavenia polohy kamery teda je, že sa nedá nastaviť *roll* uhol. Tento problém je však možné vyriešiť trikom, a to otočením HTML elementu `canvas`, do ktorého `deck.gl` vykresluje výsledné zobrazenie, pomocou CSS vlastnosti `transform`.

Za poznámku tiež stojí, že ak potrebujeme na kameru aplikovať nejakú zložitejšiu transformáciu než iba jednu rotáciu a posunutie, tak je potrebné odvodiť vzorce, ktorými sa táto zložitejšia transformácia prevedie na parametre `position`, `bearing` a `pitch`. Konkrétnym prípadom, ktorý bolo potrebné vyriešiť v tejto práci, sa zaoberá nasledujúca kapitola.

Velkou výhodou naopak je, že na rozdiel od predchádzajúceho spôsobu nedochádza k žiaľnym aktualizáciám vrstiev, teda nie je potrebné prepočítavať polohy bodov na procesore ani ich znova nahrávať na grafickú kartu. Tým sa výrazne zvýší výkonnosť.

Upresnenie polohy kamery

Experimentmi s dodanými dátami sa zistilo, že ak je mračno bodov zobrazené presne podľa dodaných údajov o polohe a parametroch kamery, výsledné zobrazenie nesedí presne na záber z kamery. Aby naozaj sedelo, je potrebné pridať isté posunutie kamery. Taká situácia je zjednodušene znázornená na obrázku 2.8 – je zadaná translácia T a rotácia R , ale kameru je ešte potrebné posunúť doprava transláciou T' .



Obr. 2.8: Zjednodušený príklad situácie, kedy je potrebné kombinovať viacero transformácií kamery. Je k dispozícii translácia T a rotácia R , ale kameru je ešte potrebné posunúť doprava transláciou T' , aby bola v strede koľajníč.

Ak by bolo pri implementácii v `deck.gl` možné použiť aplikovanie transformácií na dátu, mal by tento problém jednoduché riešenie: z translácie T a rotácie R sa zloží transformačná matica, ktorá sa použije v prístupovej funkcií, a translácia T' sa priamo použije ako poloha kamery.

Ak však tento prístup nie je vhodný, napríklad z dôvodu horšieho výkonu, je nutné nájsť transláciu T_V a rotáciu R_V tak, aby platil vzťah

$$TRT' = T_V R_V.$$

Transformácie T_V a R_V sa potom už totiž dajú priamo previesť na parametre `position`, `bearing` a `pitch` (na prevod rotácie na `bearing` a `pitch` sa dá použiť napríklad trieda `Rotation` z knižnice `scipy`).

Nech

$$T = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad T' = \begin{pmatrix} 1 & 0 & 0 & t'_1 \\ 0 & 1 & 0 & t'_2 \\ 0 & 0 & 1 & t'_3 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$T_V = \begin{pmatrix} 1 & 0 & 0 & t_{v1} \\ 0 & 1 & 0 & t_{v2} \\ 0 & 0 & 1 & t_{v3} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{a} \quad R_V = \begin{pmatrix} r_{v11} & r_{v12} & r_{v13} & 0 \\ r_{v21} & r_{v22} & r_{v23} & 0 \\ r_{v31} & r_{v32} & r_{v33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Potom

$$TRT' = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t'_1 r_{11} + t'_2 r_{12} + t'_3 r_{13} + t_1 \\ r_{21} & r_{22} & r_{23} & t'_1 r_{21} + t'_2 r_{22} + t'_3 r_{23} + t_2 \\ r_{31} & r_{32} & r_{33} & t'_1 r_{31} + t'_2 r_{32} + t'_3 r_{33} + t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

a

$$T_V R_V = \begin{pmatrix} r_{v11} & r_{v12} & r_{v13} & t_{v1} \\ r_{v21} & r_{v22} & r_{v23} & t_{v2} \\ r_{v31} & r_{v32} & r_{v33} & t_{v3} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Z toho je vyplýva, že musí platíť $R_V = R$ a

$$\begin{pmatrix} t_{v1} \\ t_{v2} \\ t_{v3} \end{pmatrix} = \begin{pmatrix} t'_1 r_{11} + t'_2 r_{12} + t'_3 r_{13} + t_1 \\ t'_1 r_{21} + t'_2 r_{22} + t'_3 r_{23} + t_2 \\ t'_1 r_{31} + t'_2 r_{32} + t'_3 r_{33} + t_3 \end{pmatrix} = \begin{pmatrix} r_{v11} & r_{v12} & r_{v13} \\ r_{v21} & r_{v22} & r_{v23} \\ r_{v31} & r_{v32} & r_{v33} \end{pmatrix} \begin{pmatrix} t'_1 \\ t'_2 \\ t'_3 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}.$$

Tým sme našli hľadanú transláciu T_V a rotáciu R_V .

2.6 Frameworky pre tvorbu webových aplikácií

Kedže hlavným cieľom práce bolo vytvoriť aplikáciu v jazyku Python, a to ideálne webovú, bolo potrebné preskúmať existujúce technológie, ktoré to umožňujú.

Porovnanie frameworkov Streamlit a Dash

Streamlit a Dash sú frameworky, ktoré majú rovnaké zameranie: oba slúžia na tvorbu webových aplikácií pre prácu s dátami (*data apps*) v jazyku Python. Dash je oproti Streamlitu na nižšej úrovni abstrakcie, pretože sám o sebe nemá žiadnen vizuálny štýl a mnohé jeho komponenty sa priamo mapujú na HTML elementy, napríklad `dash.html.Div` a `dash.html.H1` [13, 14].

Oba frameworky majú podporu pre Pydeck, u Streamlitu je priamo k dispozícii element `st.pydeck_chart` a Dash má na tento účel vytvorenú prídavnú knižnicu **Dash Deck**.

Ukázalo sa však, že `st.pydeck_chart` podporuje iba pohľad `MapView`, ktorý je určený na zobrazenie dát na mape a nedá sa použiť na perspektívne zobrazenie bodov v trojrozmernom priestore. Preto je pre účely tejto práce element `st.pydeck_chart` prakticky nepoužiteľný.

Dash Deck má navyše tú výhodu, že umožňuje vynechať Pydeck a definovať zobrazenie iba pomocou slovníkov so štruktúrou zodpovedajúcou tej, ktorú vyžaduje modul `@deck.gl/json`, čo je tiež znázornené na obrázku 2.7. To trochu zefektívni vykonávanie zmien vo vizualizácii, keďže taká reprezentácia umožní jednoduchšie vykonávanie úprav.

Z týchto dôvodov bol pre implementáciu zvolený framework Dash, ktorý je podrobnejšie popísaný v nasledujúcej podsekcii.

Popis frameworku Dash

Dash je framework, ktorý umožňuje tvorbu webových aplikácií v jazyku Python. Vytvorený kód aplikácie sa spúšta v rámci HTTP servera a generuje HTML dokument a skripty v jazyku JavaScript, ktoré server odošle klientovi. Tieto skripty následne komunikujú sa serverom, čím sa zabezpečuje funkcionality aplikácie. Server je bezstavový a neuchováva si žiadne informácie o klientoch.

Kód webovej aplikácie napísanej s využitím frameworku Dash sa skladá z dvoch základných častí: komponentov a callbackov.

Komponenty sú prvky používateľského rozhrania, ktoré sa skladajú do stromovej štruktúry. Je možné použiť komponenty priamo zodpovedajúce HTML elementom (Dash HTML Components), komponenty z knižnice Dash Core Components (napríklad `Graph`, `Input`, `Tabs`, `Upload`) a ďalšie špeciálne komponenty [13].

Pre túto prácu je významný komponent sklad (`Store`) z knižnice Dash Core Components, ktorý umožňuje uložiť dátá na strane klienta. Hodí sa na uloženie dát, ktoré sa následne zobrazujú pomocou frameworku deck.gl.

Ďalej je možné použiť knižnicu **Dash Bootstrap Components** (DBC), ktorá poskytuje ďalšie komponenty, ikony, lepšie možnosti pre rozloženie stránky, a v neposlednom rade vizuálne štýly [6].

Tam, kde nestačia štýly a možnosti rozloženia stránky z DBC, je možné u komponentov dodefinovať vlastné pravidlá v jazyku CSS.

Callbacky vytvárajú funkcionality aplikácie. Každý callback je funkcia, ktorá má vstupy a výstupy. Vstupy a výstupy sú vždy atribúty konkrétnych komponentov. Callback sa zavolá na začiatku behu aplikácie (táto vlastnosť sa dá zrušiť) a potom vždy, keď sa zmení niektorý z jeho vstupov. Callback môže mať aj špeciálne vstupy typu `State`, ktorých zmena callback nespustí. Platí obmedzenie, že každý atribút komponentu môže byť výstupom maximálne jedného callbacku [13]. U vstupov žiadne obmedzenia nie sú.

Existujú dva typy callbackov.

- Obyčajné callbacky. Sú napísané v jazyku Python. Klient má iba informáciu o ich vstupoch a výstupoch. Keď sa zmení niektorý zo vstupov, klient pošle na server požiadavok obsahujúci hodnoty všetkých vstupov. Server spustí kód callbacku a pošle klientovi odpoved, ktorá obsahuje hodnoty všetkých výstupov.
- Klientske callbacky. Sú napísané v jazyku JavaScript. Klient má k dispozícii celý kód callbacku. Keď sa zmení niektorý zo vstupov, klient vykoná kód callbacku bez akejkoľvek komunikácie so serverom.

Je zrejmé, že klientske callbacky sú efektívnejšie než tie obyčajné, pretože ich nespomaľuje rézia komunikácie so serverom. Najvýraznejšie sa to prejaví vtedy, keď vstupy alebo výstupy obsahujú veľké množstvo dát.

K aplikácii vytvorenej pomocou frameworku Dash je možné pridať vlastné štýlové predpisy a skripty alebo moduly v jazyku JavaScript. Je potrebné uložiť ich do priečinka s názvom **assets** v koreňovom priečinku aplikácie. Rovnako je nutné postupovať aj pri vkladaní obrázkov a videí. Všetky štýlové predpisy, skripty a moduly z priečinka **assets** Dash automaticky načítava a pripája k aplikácii [13].

Kapitola 3

Návrh aplikácie

Cieľom tejto práce bolo vytvoriť používateľskú aplikáciu. Výsledná aplikácia mala byť webová, aby ju bolo možné spustiť jednoducho pomocou webového prehliadača. Čo sa týka funkcionality, mala by splňať nasledujúce body:

- Zobrazenie dát z mobilného mapovacieho systému. Tieto dátá sú tvorené mračnom bodov z lidaru, kamerovým záznamom, údajmi o pohybe vlaku a ďalšími vektorovými dátami a mali by byť zobrazené z pohľadu strojvedúceho vlaku.
- Umožniť používateľovi vybrať si konkrétnu pozíciu vlaku alebo prehrať si animáciu pohybu vlaku s nastaviteľnou rýchlosťou.
- Umožniť používateľovi nahrať súbory s dátami na zobrazenie: súbor s mračnom bodov, video z kamery na čele vlaku, súbor s vektorovými dátami, textové súbory s údajmi o pohybe vlaku – pole translácií, pole rotácií a pole zodpovedajúcich časových razítok.
- Zobrazovať dva typy dát z lidaru:
 - *real-time* – malé, postupne nasnímané kusy mračna bodov s časovými razítkami,
 - *postprocess* – jedno celistvé mračno bodov, ktoré vzniklo ich spojením.

(U oboch typov ide o agregované dátá – všetky body sú v jednej spoločnej súradnicovej sústave). Umožniť používateľovi prepínať medzi týmito dvoma typmi.

- Poskytnúť používateľovi možnosti prispôsobenia zobrazenia, ako napríklad zmeny viditeľnosti jednotlivých vrstiev (mračno bodov, vektorové dátá, kamerový záznam) a základné nastavenia zobrazenia mračna bodov (rôzne farebné škály podla intenzity, veľkosť a priehľadnosť bodov, zobrazenie bodov len do určitej vzdialenosťi) aj vektorových dát (farba a hrúbka čiar).
- Mať prepínač na zapnutie a vypnutie skreslenia mračna bodov a vektorových dát podľa parametrov skreslenia kamery.
- Zobrazovať prejazdný profil vlaku v predikovanej polohe v rôznych vzdialenosťach pred vlakom – 25, 50, 75 a 100m. Zobrazovať aj čiaru spájajúcu predikované polohy. (Súbory s predikovanými polohami sú súčasťou dát, ktoré aplikácia načítava.)

3.1 Návrh používateľského rozhrania

Pri návrhu používateľského rozhrania aplikácie bolo prioritou zobrazenie vizualizácie ako hlavnej časti aplikácie na čo najväčšej ploche a tiež tak, aby bola viditeľná vo všetkých stavoch.

Z ovládacích prvkov sú za najdôležitejšie považované prvky pre zmenu polohy vlaku, ktoré sú umiestnené v spodnom paneli. Všetky ostatné ovládacie prvky sú skryté v bočnom paneli, ktorý sa dá vysunúť tlačítkom v ľavom hornom rohu, a sú rozdelené do troch záložiek – nahrávanie dát, prispôsobenie zobrazenia a samostatná záložka pre prejazdný profil. Celkový návrh vzhľadu je na obrázku 3.1.

Hlavnou ideou návrhu je používanie čo najviac štandardných prvkov, aby sa v aplikácii používateľia ľahko zorientovali. Tomu napomáha aj použitie knižnice Bootstrap, ktorá je u webových aplikácií veľmi často používaná, a preto budú jej prvky pravdepodobne pre používateľov intuitívne.

Pre používateľskú prívetivosť by mala aplikácia splňať nasledujúce body:

1. všeobecné vlastnosti:

- intuitívnosť,
- responzívny design,
- hladký beh animácií,
- možnosť exportu a importu nastavení zobrazenia,
- svetlý aj tmavý režim,
- kompatibilita s rôznymi prehliadačmi, prípadnú nekompatibilitu používateľovi ohlásit,

2. v záložke „Dáta“:

- prehľadné zobrazenie, aké dátá boli nahrané, možnosť ich nahradenia inými dátami,
- zobrazenie indikátoru, že práve prebieha nahrávanie dát.

3.2 Problém nahrávania dát do aplikácie

Dáta, ktoré má vyvíjaná webová aplikácia zobrazovať, sú rozdelené do väčšieho množstva súborov rôznych typov. Nahrávať ich do aplikácie klasickým spôsobom po jednom či po nejakých skupinách by bolo súčasťou možné, ale pre používateľov časovo náročné a veľmi nepraktické.

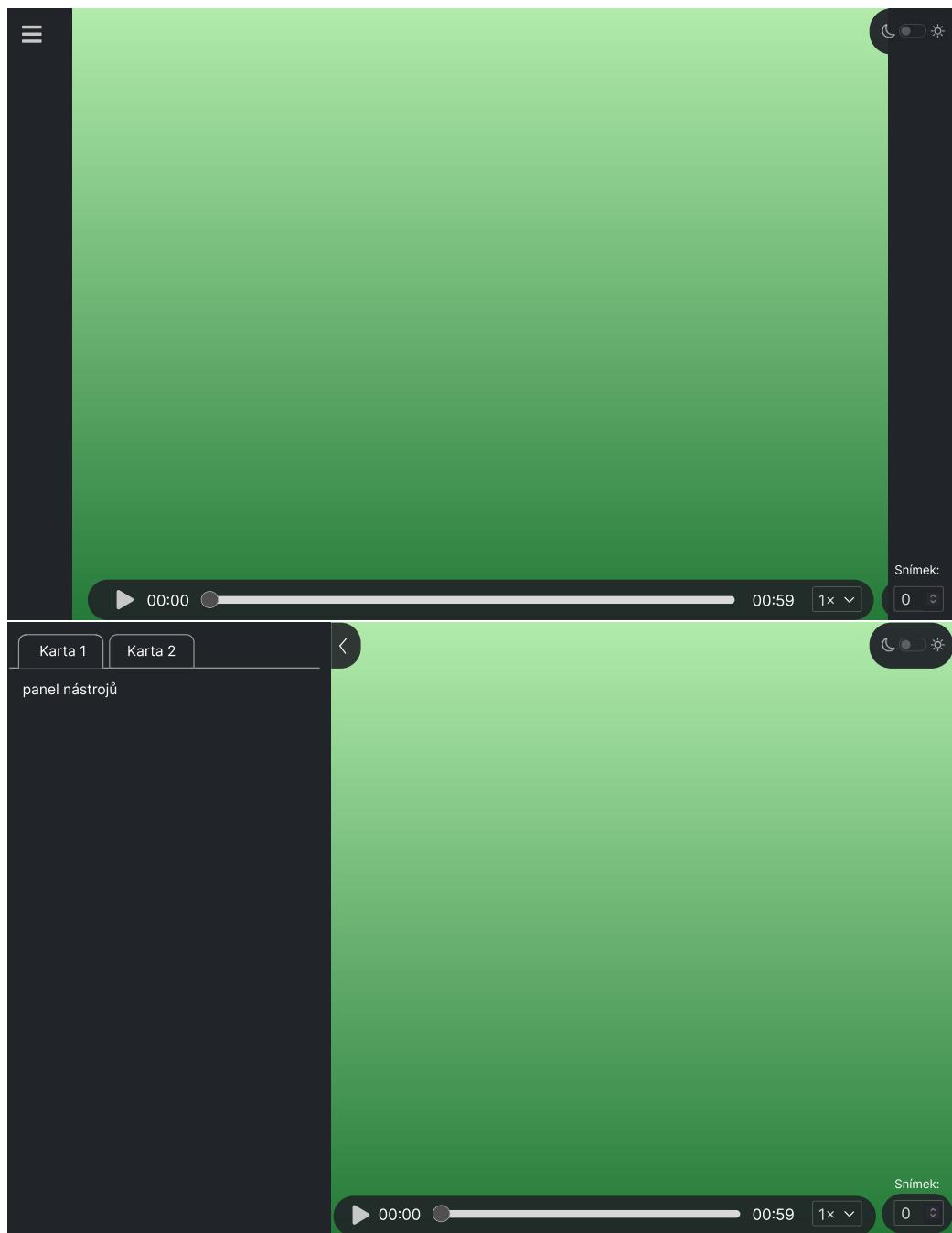
Kedže je aplikácia určená úzkej cieľovej skupine používateľov, môžeme si dovoliť predpokladať, že používateľ bude buď spúšťať server webovej aplikácie lokálne, alebo bude mať k serveru aspoň prístup. Za tohto predpokladu má problém s nahrávaním dát veľmi elegantné riešenie, a tým je **projektový súbor**, teda textový súbor, v ktorom sú špecifikované cesty k dátovým súborom uloženým na serveri. Používateľovi stačí nahrať tento súbor a aplikácia podľa neho automaticky nahrá všetky potrebné dátové súbory.

Presný formát projektového súboru bolo potrebné navrhnúť. Kompletný návrh je uvedený v prílohe [DOPLNIT]. Bol zvolený formát TOML, ktorý je jednoduchý, praktický a dobre čitateľný [19]. U dátových súborov, ktorých môže byť rôzny počet, napríklad súbory s dátami z lidaru, sa do projektového súboru uvedie, v akom priečinku sa nachádzajú, ako

sú pomenované a koľko ich je. Predpokladá sa, že sú tieto súbory jednotne pomenované a očíslované (napríklad `pcd_0.pcd`, `pcd_1.pcd`, ...).

Keby však bol projektový súbor jediným možným spôsobom nahrávania dát do aplikácie, malo by to isté nevýhody – napríklad pre vyskúšanie zmeny v jednom dátovom súbore by bolo potrebné meniť projektový súbor a znova ho nahrávať.

Preto by mala výsledná aplikácia kombinovať oba prístupy, projektový súbor aj klasické nahrávanie dát. Pre jednoduchosť a prístupnosť by mala umožniť nahranie základných súborov klasickým spôsobom. Naopak pre pokročilejšie použitie by mala mať možnosť nahrania projektového súboru, podľa ktorého by mala nahrať všetky špecifikované dátové súbory. Potom by mal používateľ prehľadne vidieť všetky nahrané súbory a mať možnosť tieto súbory jednotlivo lubovoľne nahradzovať inými súbormi.



Obr. 3.1: Návrh používateľského rozhrania aplikácie vytvorený pomocou nástroja Figma.

Kapitola 4

Implementácia navrhnutej aplikácie

Navrhnutá webová aplikácia je implementovaná v jazyku Python s využitím frameworku Dash. Pre optimalizáciu obsahuje aplikácia aj skript napísaný v jazyku Javascript, ktorý používa pre vykreslovanie mračna bodov a vektorových dát hardvérovo akcelerovaný framework deck.gl a na rozdiel od zvyšku kódu aplikácie sa vykonáva priamo na klientovi. Táto štruktúra je schematicky znázornená na obrázku 4.1.

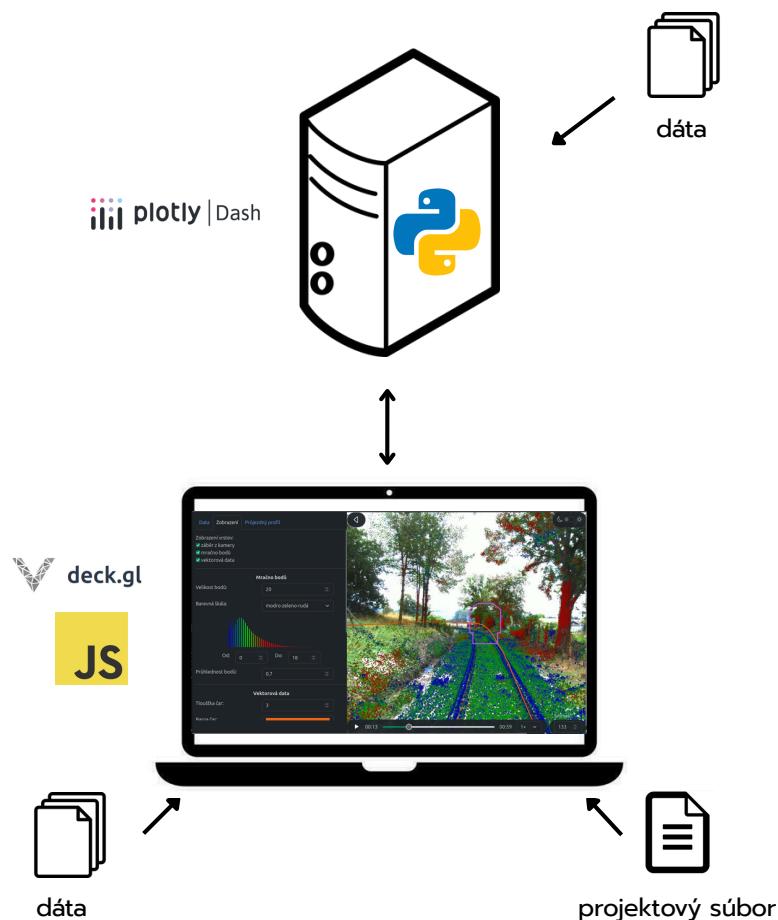
Vizuálny štýl používateľského rozhrania čiastočne zabezpečuje knižnica Dash Bootstrap Components, čiastočne je dodefinovaný pomocou CSS.

Zdrojový kód je rozdelený do nasledujúcich súborov:

- `app.py` – Hlavný modul, inicializuje a spustí aplikáciu.
- `params.py` – Počiatočné parametre.
- `loading_functions.py` – Pomocné funkcie pre načítanie dát zo súborov.
- `general_functions.py` – Pomocné funkcie pre výpočet projekčnej matice, transformačných matíc a operácií s rotáciami.
- Moduly s komponentmi a callbackmi rozdelené podľa štruktúry používateľského rozhrania. Sú to štyri dvojice súborov. V jednom súbore z dvojice sú vždy definované komponenty používateľského rozhrania vrátane ich vizuálneho štýlu, v tom druhom k nim patriace callbacky.
- `visualization.js` – Vizualizácia mračna bodov a vektorových dát pomocou frameworku deck.gl, riadenie animácie phybu vlaku. Činnosť tohto skriptu je podrobnejšie popísaná v sekciách 4.2 a 4.3.

4.1 Popis modulov

Táto sekcia sa venuje podrobnejšiemu popisu niektorých modulov.



Obr. 4.1: Schéma finálnej implementácie.

Hlavný modul (súbor `app.py`)

Načíta zo súborov ukažkové dátá, ktoré aplikácia zobrazí po spustení. Tieto dátá spolu s počiatočnými parametrami uloží do skladov, z ktorých ich následne callbacky definované v module `data` kopírujú do objektu `window`, aby boli prístupné skriptu `visualization.js`.

Definuje základné rozloženie a vzhľad aplikácie – vizualizácia na celú výšku okna, spodný panel, bočný panel s troma záložkami a prvok na prepínanie svetlého a tmavého režimu. Vizualizácia je zložená z troch HTML elementov: `Video`, `Canvas`, do ktorého vykresluje dátá framework `deck.gl`, a druhý `Canvas` pre implementáciu skreslenia. Toto základné rozloženie je definované väčšinou pomocou CSS pravidiel. Záložka `Data` je obalená komponentom `Loading` z knižnice Dash Core Components, ktorý sa používateľovi zobrazuje, keď prebieha aktualizácia niektorého prvku z tejto záložky, teda keď aplikácia nahráva dátá.

Modul načíta všetky potrebné komponenty a callbacky z iných modulov, inicializuje a spustí aplikáciu. Obsahuje tri klientske callbacky - jeden pre inicializáciu vizualizácie, jeden pre vysúvanie bočného panelu a jeden pre prepínanie svetlého a tmavého režimu.

Modul *animation control*

(súbory `animation_control_components.py`, `animation_control_callbacks.py`)

Obsahuje prvky pre ovládanie animácie – spustenie a zastavenie, posunutie, zmena rýchlosťi, výber konkrétneho snímku, – ktoré sa nachádzajú v spodnom paneli. Prvky sú rozmiestnené s využitím CSS mechanizmov *grid* a *flexbox*.

Kedže riadenie animácie prebieha na strane klienta, všetky callbacky v tomto module sú klientske a väčšinou volajú funkcie definované v skripte `visualization.js`. Mechanizmus riadenia animácie je popísaný v sekcií 4.3.

Zmena rýchlosťi animácie prebieha tak, že sa u videa zmení vlastnosť `playbackRate`. Tým sa zmení rýchlosť videa, a keďže vykreslovanie mračna bodov a vektorových dát je naviazané na video, zrýchli sa tým celá animácia.

Modul *data*

(súbory `data_tab_components.py`, `data_tab_callbacks.py`)

Tvorí záložku *Data* v bočnom paneli a riadi nahrávanie dát do aplikácie. Obsahuje komponenty dvoch typov – `*_upload` a `*_uploaded_file[s]`. Komponenty typu `*_upload` umožňujú manuálne nahranie súborov do aplikácie. Keď také nahranie prebehne, komponent `*_upload` je skrytý a namiesto neho sa zobrazí komponent `*_uploaded_file[s]`, ktorý používateľovi pre kontrolu zobrazuje názov nahraného súboru a obsahuje aj tlačítko v tvare x, ktoré skryje komponent `*_uploaded_file[s]` a znova zobrazí komponent `*_upload`, čím umožní používateľovi nahrať iný súbor.

Špeciálnym typom nahrávaného súboru je projektový súbor, ktorý vyvolá automatické nahranie všetkých v ním definovaných súborov.

Niektoré typy dát nemajú komponent `*_upload` a dajú sa nahrať iba pomocou projektového súboru. Sú to tie typy dát, ktoré sa skladajú z viacerých súborov, a to konkrétnie mračno bodov typu real-time, vektorové dáta a polohy prejazdného profilu.

Väčšina callbackov je napísaná v Pythone, spracúvajú nahrané súbory a spracované dátá ukladajú do skladov (komponenty *Store*).

Isté špecifické sú u nahrávania súboru s mračnom bodov a videa. Framework Dash totiž načítané súbory odovzdáva callbackom ako pole bytov, ale použitá knižnica `pypcd4` dokáže načítať súbor vo formáte `pcd` iba podľa cesty k súboru. Preto je nahraný súbor s mračnom bodov uložený ako dočasný súbor do priečinka `assets/temp`, odtiaľ nahraný a potom hned zmasaný.

Podobne je to u videa, ktoré je tiež potrebné uložiť ako dočasný súbor a potom nastaviť cestu k nemu do atribútu `src` komponentu `html.Video`. Na rozdiel od súboru s mračnom bodov ho však nie je možné zmazať hned v tom istom callbacku, musí v priečinku `assets/temp` po istú dobu zostať, pretože jeho nahranie do aplikácie nejakú dobu trvá. Preto je definovaný špeciálny callback, ktorý sa spustí každé dve minúty a vymaže všetky dočasné súbory staršie než dve minúty. Okrem toho novému videu po nahraní musí byť nastavený rovnaký čas, ako malo to predchádzajúce, čo je zabezpečené ďalším callbackom.

Ďalej sú v tomto module ešte klientske callbacky, ktoré kopírujú nové dátá zo skladov do objektu `window`, aby boli prístupné skriptu `visualization.js`.

Modul *visualization*

(súbory `visualization_tab_components.py`, `visualization_tab_callbacks.py`)

Tvorí záložku *Zobrazení* v bočnom paneli a obsahuje všeobecné nastavenia týkajúce sa zobrazenia. Dôležitou funkciou je voľba typu zobrazovaného mračna bodov – postprocess alebo real-time.

Najzložitejším komponentom je graf, ktorý zobrazuje rozloženie bodov podľa intenzity a zároveň aj farebnú škálu (ukážka je na obrázku 4.3). Ďalej táto záložka obsahuje aj posuvníky pre upresnenie polohy kamery a tlačidlá pre export a import nastavení zobrazenia. Nastavenia sa exportujú vo formáte `toml`. Táto možnosť je užitočná najmä pre uloženie nastavenia posuvníkov, ale ukladá aj všetky ostatné nastavenia zo záložiek *Zobrazení* a *Prújezdny profil*.

Nachádza sa tu aj možnosť zapnutia a vypnutia skreslenia vizualizácie podľa parametrov skreslenia kamery.

Modul obsahuje niekoľko klienskych callbackov, ktoré volajú funkcie definované v skripte `visualization.js`.

Ďalej je tu viacero callbackov patriacich ku grafu zobrazujúcemu farebnú škálu. Kedže je to zložitý komponent, obsahuje štyri sklady, do ktorých sa ukladá typ farebnej škály, jej rozsah a agregácia mračna bodov oboch typov, ktoré aplikácia zobrazuje. Graf sa aktualizuje, keď sa zmení ktorýkoľvek z týchto údajov, alebo keď sa zmení typ zobrazovaného mračna bodov.

V module sa ešte nachádzajú callbacky pre počítanie skreslenia, ktoré sú bližšie popísané v sekcií 4.4.

Modul *profile*

(súbory `profile_tab_components.py`, `profile_tab_callbacks.py`)

Tvorí záložku *Prújezdny profil* v bočnom paneli a obsahuje iba nastavenia týkajúce sa zobrazenia prejazdného profilu a čiary cez polohy prejazdného profilu. Umožnuje používateľovi vybrať si, ktorú predikciu chce vidieť – na 25, 50, 75 alebo 100 metrov.

Callbacky sú klienske a volajú funkcie definované v skripte `visualization.js`.

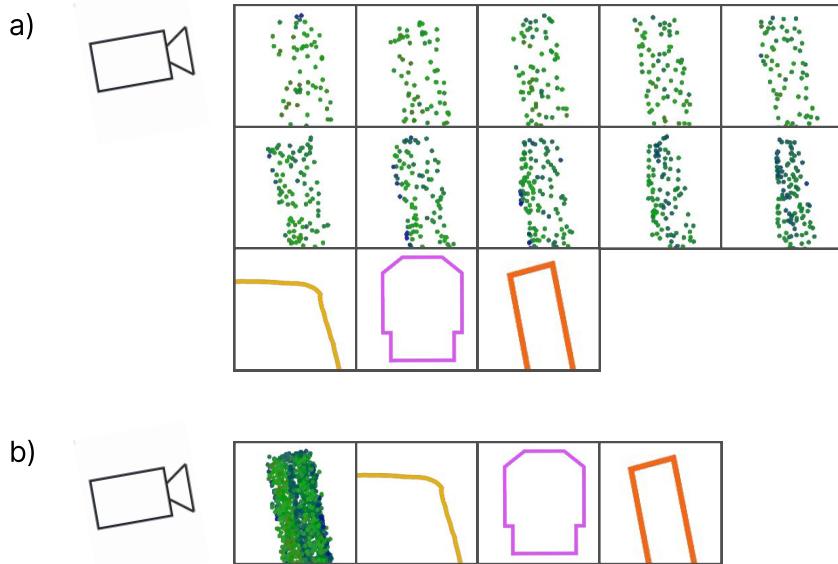
4.2 Rozdelenie dát do vrstiev

Pri vizualizácii dát pomocou frameworku deck.gl sú použité dva typy vrstiev:

- `PointCloudLayer` pre mračno bodov,
- `PathLayer` pre vektorové dátá.

Rozdelenie vizualizácie na vrstvy je znázornené na obrázku 4.2.

Na začiatku tvorby práce bola na zobrazenie vektorových dát použitá vrstva `LineLayer`. Ukázalo sa však, že vstva `LineLayer` je prispôsobená iba pre zobrazovanie čiar na mapách a pri použití nemapového pohľadu ako `FirstPersonView` nefunguje dobre – niektoré čiary sa otáčajú kolmo ku kamere a tým pádom sa ich šírka zmenšuje, niekedy až tak, že úplne prestanú byť viditeľné. Toto správanie u vrstvy `LineLayer` nie je možné zmeniť. Naopak u vrstvy `PathLayer` je možné nastaviť vlastnosť `billboard`, ktorá znamená otočenie čiar smerom ku kamere, preto bola nakoniec táto vrstva použitá namiesto vrstvy `LineLayer`. `PathLayer` sa od `LineLayer` líši ešte tým, že čiary majú pevnú šírku v pixeloch a nezužujú sa podľa toho, ako ďaleko sú od kamery.



Obr. 4.2: Schéma rozdelenia dát do vrstiev. Počet vrstiev s mračnom bodov závisí od typu zobrazovaného mračna bodov.

Vizualizácia obsahuje nasledujúce vrstvy:

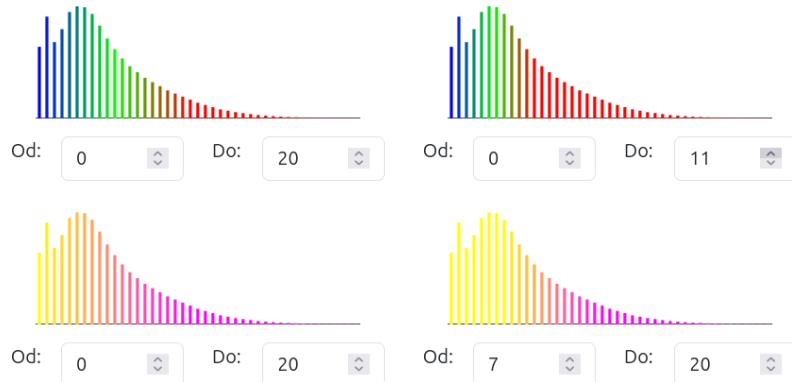
- Mračno bodov. Ak aplikácia zobrazuje *rozdelené* mračno bodov, teda malé, postupne nasnímané časti s časovými razítkami, tak je vrstiev desať a zobrazujú desať z týchto častí – jednu, ktorá zodpovedá danej pozícii, a deväť predchádzajúcich (obrázok 4.2, časť a)). Ak aplikácia zobrazuje mračno bodov typu postprocess, tak je celé v jednej vrstve (obrázok 4.2, časť b)).
- Čiara spájajúca predikované polohy prejazdného profilu vlaku.
- Prejazdný profil vlaku v predikovanej polohe. Od ostatných vektorových dát sa lísi tým, že sa jeho poloha mení pri zmene polohy kamery. To je v implementácii dosiahnuté pomocou špeciálnej prístupovej funkcie `profileGetPath` (prístupové funkcie boli popísané v sekcií 2.5, podsekcia Aplikovanie transformácií na dátu).
- Ďalšie vektorové dáta.

Farebné škály

Okrem polohy má každý bod v mračne bodov v dodanej sade dát určenú ešte jednu vlastnosť – intenzitu v rozmedzí od 0 do 42, ktorú je vhodné farebne rozlíšiť.

Boli implementované dve farebné škály, klasická modro-zeleno-červená a žltlo-fialová, pričom u oboch si môže používateľ zvoliť, od akej intenzity začína a pri akej končí. Ak je napríklad nastavená žltlo-fialová farebná škala od 0 do 20, tak to znamená, že body s intenzitou 0 budú mať žltú farbu, body s intenzitou 20 a vyššou fialovú farbu a body s intenzitou od 1 do 19 kombinovanú farbu.

Pre prehľadnosť bol tiež implementovaný graf, ktorý používateľovi ukazuje relatívne počty bodov s jednotlivými intenzitami a aké majú nastavené farby. Ukážky možných nastavení sú na obrázku 4.3.



Obr. 4.3: Nastaviteľná farebná škála pre mračno bodov. Vodorovná os predstavuje intenzitu, vertikálna os predstavuje počet bodov.

4.3 Implementácia animácie pohybu vlaku

Animácia pohybu vlaku sa vykonáva výhradne na strane klienta, bez komunikácie so serverom. Zabezpečujú ju tri javascriptové funkcie, ktorých definícia sa nachádza v súbore `visualization.js`: `runDeckAnimation`, `stopDeckAnimation` a `animationStep`.

V tejto sekcii je popísaný princíp fungovania animácie, pre prehľadnosť boli vynechané niektoré menej dôležité implementačné detaily.

Ked' používateľ klikne na tlačítko pre prehranie animácie, spustí sa klientsky callback, ktorý vykoná nasledujúci kód:

```
const video = document.getElementById('background-video');
window.runDeckAnimation();
video.play();
video.onPause = function(){ window.stopDeckAnimation() };
```

Funkcia `runDeckAnimation` reinicializuje niektoré parametre, ak je to potrebné, a potom zavolá funkciu `animationStep`.

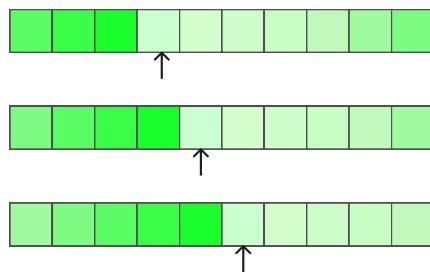
Funkcia `animationStep` vykonáva tieto kroky:

1. Najprv zabezpečí, aby sa zavolala znova pri najbližšom vykreslení nového snímku vo videu:

```
const video = document.getElementById('background-video');
video.requestVideoFrameCallback(animationStep);
```

2. Potom na základe časových razítok polôh kamery v mračne bodov a aktuálneho času videa vypočíta aktualizovanú polohu kamery. Ak je to posledná poloha, zastaví video metódou `pause`.

3. Ak aplikácia zobrazuje mračno bodov typu real-time, tak následne na základe časových razítok mračna bodov zistí, či by sa nemali zobraziť nejaké nové časti. Vymieňanie častí prebieha efektívnym cyklickým spôsobom, ktorý je znázornený na obrázku [4.4](#).
4. Potom prebehne samotná aktualizácia vizualizácie (v skripte funkcia `updateDeck`), kedy sa kamera posunie na novú polohu a aktualizujú sa vrstvy (v skripte funkcia `createLayers`), čím sa posunie prejazdný profil a prípadne sa vymenia časti mračna bodov.
5. Ako posledné sa aktualizujú prvky v grafickom používateľskom rozhraní, ktoré ukažujú stav animácie – posuvník, čas a vstupné pole s číslom snímky.



Obr. 4.4: Schéma cyklického prepisovania deck.gl vrstiev obsahujúcich mračno bodov typu real-time. Vizualizácia obsahuje vždy desať vrstiev, ktoré zobrazujú desať častí mračna bodov. Ak podla časových razítok má byť zobrazená nejaká nová časť, tak sa prepíše vždy tá najstaršia.

Animácia sa končí buď tak, že používateľ klikne na tlačítko pre zastavenie animácie, čím sa spustí klientsky callback, ktorý zastaví video metódou `pause`, alebo tak, že skončí video, alebo tak, že je dosiahnutá posledná poloha kamery. V každom prípade sa spustí funkcia `stopDeckAnimation`, pretože bola na začiatku animácie nastavená ako callback pre udalosť zastavenia videa.

Funkcia `stopDeckAnimation` upovedomí framework Dash o novej hodnote posuvníka, času a vstupného pola s číslom snímku. Toto sa nevykonáva počas animácie, pretože je to výkonovo náročné.

4.4 Implementácia skreslenia

Pri zobrazovaní mračna bodov a vektorových dát tak, aby zobrazenie čo najpresnejšie sedelo na záznam z kamery, je potrebné vziať do úvahy aj skreslenie kamery, ktoré bolo popísané v sekcií [2.2](#).

Parametre skreslenia boli súčasťou dodaných dát. Existuje niekoľko možností, ako ich použiť:

1. Odstrániť skreslenie zo záznamu z kamery.
2. Aplikovať skreslenie na mračno bodov, a to:
 - (a) meniť súradnice bodov v rámci niektornej transformácie vo vykresľovacom retazci,

(b) aplikovať skreslenie až na finálny obraz.

Možnosti 1 a 2b) sú výkonovo približne rovnako náročné a výkonnosť závisí od rozmerov vizualizácie v pixeloch.

Možnosť 2a) by bola v rámci frameworku realizovateľná dvoma spôsobmi: buď výpočtom skreslenia v prístupových funkciách, alebo implementáciou vlastných vrstiev a napísaním vlastného kódu pre vertex shader. Akékoľvek výpočty v prístupových funkciách však majú výrazný vplyv na výkon, výpočet skreslenia je pomerne zložitý (oproti napríklad násobeniu maticou, ktoré sa u bežných transformácií používa) a výkonnosť by sa naviac zhoršovala s počtom vykreslovaných bodov. Druhý spôsob je implementačne náročný a realizácia by zabrala veľa času.

Preto bola zvolená a implementovaná možnosť 2b).

Pri hľadaní existujúcich riešení podobných problémov bol nájdený článok od J. Balochu, ktorý sa zaobrá aplikáciou rôznych efektov na obrázky vo webových stránkach pomocou Javascriptu [3].

Toto riešenie funguje tak, že vytvorí element **Canvas** rovnakej veľkosti ako pôvodný obrázok, cyklom prechádza všetky pixely obrázku a pre každý pixel vypočíta jeho novú polohu a vykreslí ho do elementu **Canvas**.

Myšlienka tohto riešenia bola využitá aj pri implementácii skreslenia v tejto práci, ale s niekolkými rozdielmi.

Poprvé, pixely sa nečítajú z obrázku, ale z elementu **Canvas**, do ktorého vykresluje zobrazenie framework deck.gl. Keďže je tento element naviazaný na framework deck.gl, nie je možné z neho prečítať pixel jednoducho príkazmi

```
ctx = canvas.getContext("2d");
pixelData = ctx.getImageData(...);
```

Namiesto toho je nutné použiť rozhranie WebGL:

```
const gl_ctx = canvas.getContext("webgl2");
const pixelData = new Uint8Array(width * height * 4);
gl_ctx.readPixels(0, 0, width, height, gl_ctx.RGBA, gl_ctx.UNSIGNED_BYTE,
    pixelData);
```

Podruhé, keďže vo vyvíjanej aplikácii je potrebné počítať skreslenie mnohokrát s tými istými parametrami a rozmermi obrazu, bolo by neefektívne pri každom výpočte počítať novú polohu pre každý pixel, pretože tieto hodnoty sa nemenia.

Preto je skreslenie implementované dvoma klientskymi callbackmi. Ten prvý vypočíta pre každý pixel jeho novú polohu v skreslenom obraze a výsledky uloží do poľa v objekte **window**. Tento výpočet je nutné zopakovať iba vtedy, keď sa zmenia rozmery vizualizácie. Ten druhý reaguje na vypnutie a zapnutie skreslenia používateľom. Keď je skreslenie zapnuté, skryje element **visualization-canvas**, do ktorého vykresluje zobrazenie deck.gl, zviditeľní namiesto neho element **distorted-visualization-canvas**, ktorý má rovnakú veľkosť a pozíciu, a zaregistruje javascriptový callback, ktorý po každom vykreslení zobrazenia frameworkom deck.gl prečíta hodnotu všetkých pixelov a prekreslí ich na predpočítané pozície do elementu **distorted-visualization-canvas**.

Kapitola 5

Testovanie a evaluácia

Pri vývoji aplikácie bola k dispozícii iba jedna sada dát, ktorú bolo možné použiť pre testovanie.

Už pri prvých pokusoch o zobrazenie mračna bodov sa ukázalo, že údaje o polohách kamery sú v inom súradnicovom systéme ako mračno bodov, pretože sa líši poradie os, čomu bola implementácia prispôsobená.

Ďalej sa ukázalo, že aj pri zobrazení mračna bodov presne podľa dodanej polohy kamery a kalibračnej matice zobrazenie nesedí presne na záznam z kamery a je potrebné pridať isté posunutie a rotáciu. Preto boli do používateľského rozhrania aplikácie pridané posuvníky, ktorými si používateľ môže prispôsobiť polohu virtuálnej kamery (posunutím a otočením v istom rozsahu), a tým opraviť nepresnosti v dátach.

5.1 Optimalizácia vykreslovania mračna bodov

V priebehu implementácie boli vyskúšané štyri rôzne spôsoby vykreslovania mračna bodov:

1. a
2. a
3. a
4. a

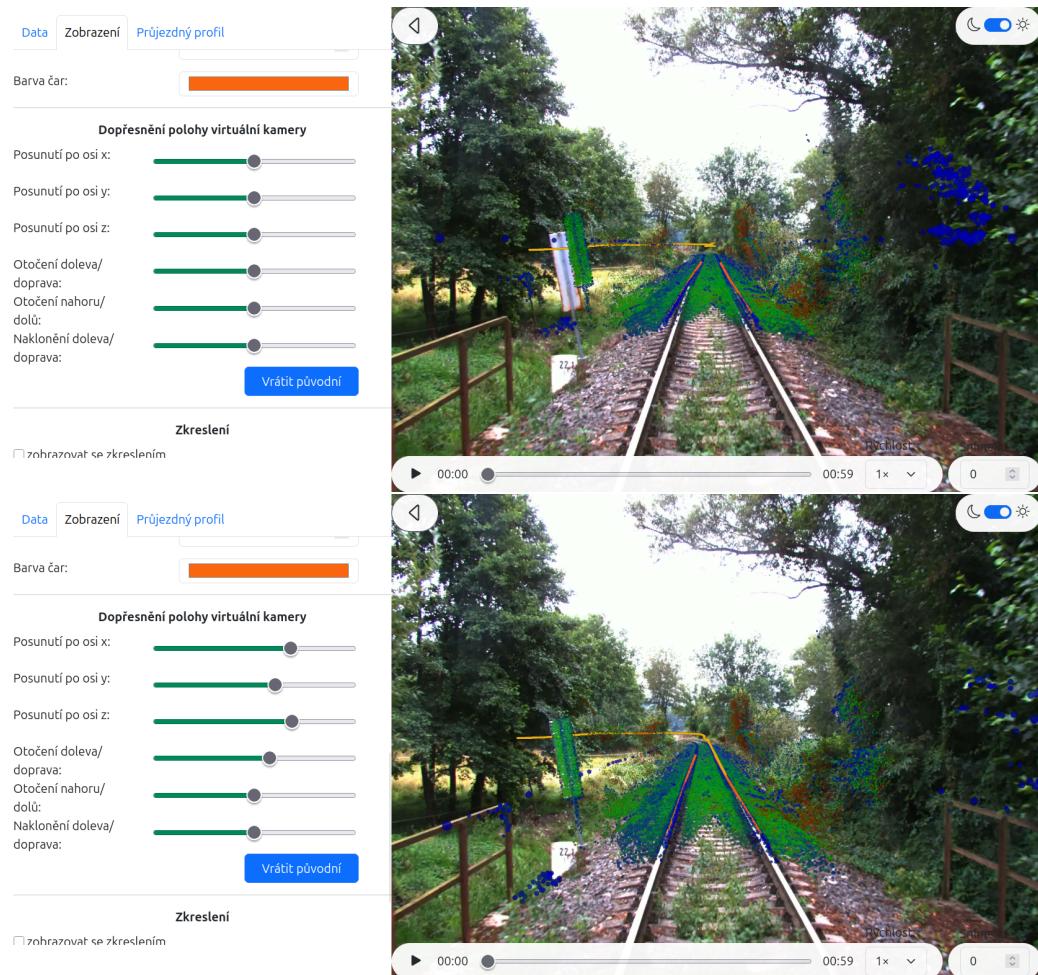
Počas vývoja aplikácia prešla niekoľkými zmenami spôsobu vykreslovania mračna bodov a vektorových dát z dôvodu optimalizácie výkonu, ktoré sú popísané v nasledujúcich podsekciách.

Všetky v nich uvedené merania prebiehali na notebooku s operačným systémom Ubuntu 24.04.1, procesorom Intel Core i5-9300H × 8, grafickou kartou Intel UHD Graphics 630 (CFL GT2) a veľkosťou operačnej pamäte 16 GiB.

Profiling prebiehal v prehliadači Google Chrome 133.0.6943.126 s rozlíšením obrazovky 1480 × 832. Bol použitý profiler vstavaný v tomto prehliadači.

Optimalizácia priamym použitím frameworku deck.gl

Aby bolo možné vyniechať Dash Deck a použiť deck.gl priamo, bolo nutné nájsť spôsob, ako do aplikácie napísanej pomocou pythonového frameworku Dash zahrnúť kód v jazyku JavaScript, a zabezpečiť komunikáciu medzi týmto kódom a zvyškom aplikácie.



Obr. 5.1: Funkcionalita posuvníkov pre prispôsobenie polohy virtuálnej kamery. Hore je zobrazenie bez prispôsobenia, dole s prispôsobením.

Dashboard interne používa na vytváranie používateľského rozhrania JavaScript, konkrétnie framework React [13]. Jeho tvorcovia rátali s tým, že vývojári, ktorí ho budú používať, budú niekedy chcieť priamo použiť javascriptový kód. Preto to tento framework umožňuje viacerými spôsobmi:

- Klientske callbacky. Podobajú sa tým štandardným, ale na rozdiel od nich bežia iba na klientovi a písu sa v jazyku JavaScript. Štandardné callbacky bežia na serveri a písu sa v Pythone.
- Skripty v jazyku JavaScript, ktoré je možné pripojiť k generovanej HTML stránke. Spustia sa automaticky pri otvorení stránky v prehliadači.
- Vytváranie vlastných komponentov používateľského rozhrania pomocou frameworku React.

V tejto práci bola pre jednoduchosť využitá prvá a druhá možnosť. Bol napísaný skript `visualisation.js`, v ktorom sú definované funkcie pre zobrazenie a zmeny zobrazenia dát pomocou `deck.gl`. Odkazy na tieto funkcie sú potom priradené do objektu `window`, aby

bolo možné volať ich z klientskych callbackov. Tento objekt teda tvorí rozhranie skriptu `visualisation.js`, ktoré je využívané zvyškom kódu aplikácie.

Pre správne pripojenie zdrojových kódov frameworku `deck.gl` k javascriptovému kódu tak, aby mohol fungovať v rámci aplikácie napísanej vo frameworku Dash, bolo nutné použiť bundler. V rámci tejto práce bol použitý bundler Webpack. Výstupnému skriptu bolo potrebné nastaviť príponu `.mjs`, aby ho Dash spustil ako modul. Mimo modulu totiž nie je v jazyku JavaScript možné použiť deklaráciu `import`, ktorá je nutná pre pripojenie zdrojových kódov frameworku `deck.gl`.

Nastavenie parametra <code>interval</code> [ms]	Čas, za ktorý prebehla celá animácia (100 snímok) [s]	Priemerný čas vykreslenia jednej snímky [ms]
1000	101	1010
800	80	800
750	76	760
700	72	720
650	71	710
600	70	700

Tabuľka 5.1: Výkonnosť vykreslovania mračna bodov obsahujúceho 201 880 bodov *pred prou optimalizáciou*, teda iba pomocou kódu v jazyku Python. Použité technológie: Python, Dash, Dash Deck. Z nameraných údajov vyplýva, že minimálny čas potrebný vykreslenie jednej snímky je asi 750 ms, čo znamená, že rýchlosť nedosahuje ani 2 snímky za sekundu.

Nastavenie parametra <code>interval</code> [ms]	Čas, za ktorý prebehla celá animácia (500 snímok) [s]	Priemerný čas vykreslenia jednej snímky [ms]
250	126	252
200	101	202
150	76	152
125	68	136
100	63	126
75	62	124

Tabuľka 5.2: Výkonnosť vykreslovania mračna bodov obsahujúceho 201 880 bodov *po prvej optimalizácii*, teda s vykreslovaním dát v jazyku Javascript. Použité technológie: Python, Dash, Javascript, `deck.gl`. Z nameraných údajov vyplýva, že minimálny čas potrebný vykreslenie jednej snímky je asi 150 ms, čo zodpovedá rýchlosťi asi 6 snímok za sekundu.

Meraná bola výkonnosť vykreslovania mračna bodov počas animácie pohybu vlaku. Animácia bola riadená pomocou komponentu `Interval` frameworku Dash, ktorého funkcionálita spočíva v tom, že periodicky spúšta callback, pričom periódu určuje parameter `interval`.

Meranie výkonnosti vykreslovania mračna bodov pred a po optimalizácii je popísané v tabuľkách 5.1 a 5.2.

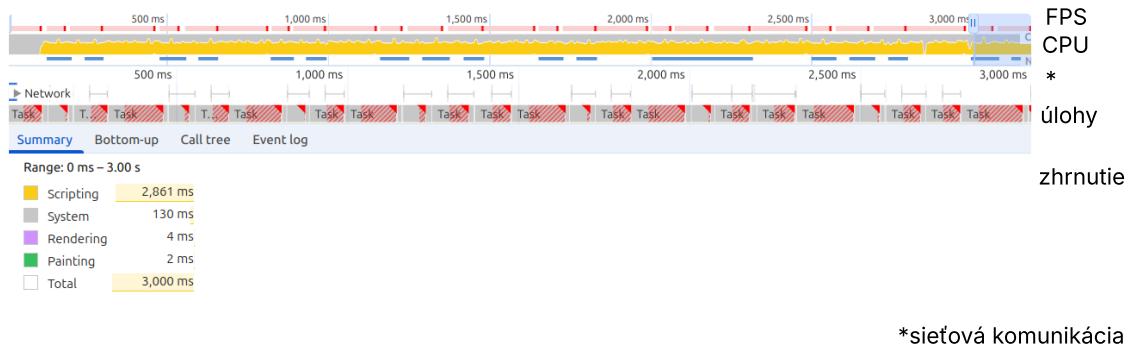
Z dát v tabuľkách vyplýva, že optimalizácia výkon výrazne zlepšila.

Optimalizácia riadením animácie výhradne na strane klienta

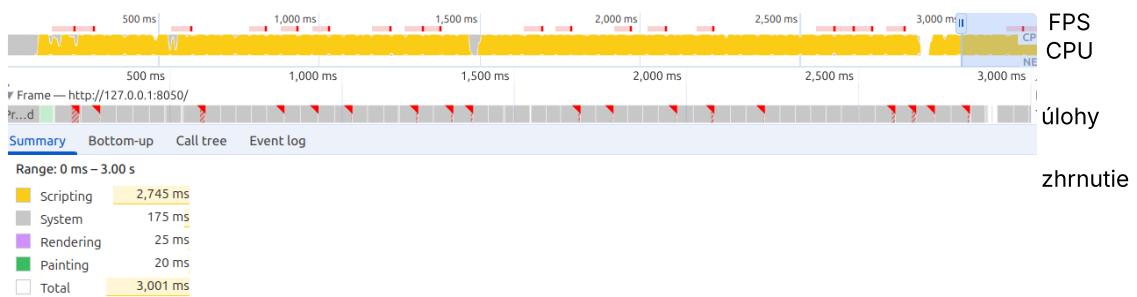
Optimalizácia popísaná v predchádzajúcej podsekcii síce priniesla z hľadiska výkonu zlepšenie, ale výsledok stále neboli dostačujúci.

Riadenie animácie pomocou komponentu **Interval** frameworku Dash je totiž pomerne ťažkopádne, pretože funguje na základe komunikácie medzi klientom a serverom a tým animáciu spomaľuje. Preto bolo vhodné nahradíť ho riadením animácie v jazyku JavaScript výhradne na strane klienta.

Podrobnejší popis tohto spôsobu riadenia animácie je uvedený v sekcií 4.3.



Obr. 5.2: Profiling animácie pohybu vlaku, ktorá je riadená pomocou komponentu **Interval**.



Obr. 5.3: Profiling animácie pohybu vlaku, ktorá je riadená iba pomocou kódu v jazyku JavaScript.

Oba spôsoby boli porovnané pomocou profilingu. V aplikácii sa prehrávala animácia pohybu kamery spojená s videom. Vizualizované dátá tvorila jedna vrstva **PointCloudLayer**, ktorá obsahovala 201 880 bodov. Výsledky profilingu sú na obrázkoch 5.2 a 5.3.

Z grafov je možné vyčítať, že hoci sa zvýšilo zataženie procesora, výrazne sa znížil čas potrebný na vykreslenie jednej snímky animácie, čo umožnilo vykreslenie väčšieho počtu snímok za sekundu (v grafoch sekcia „úlohy“).

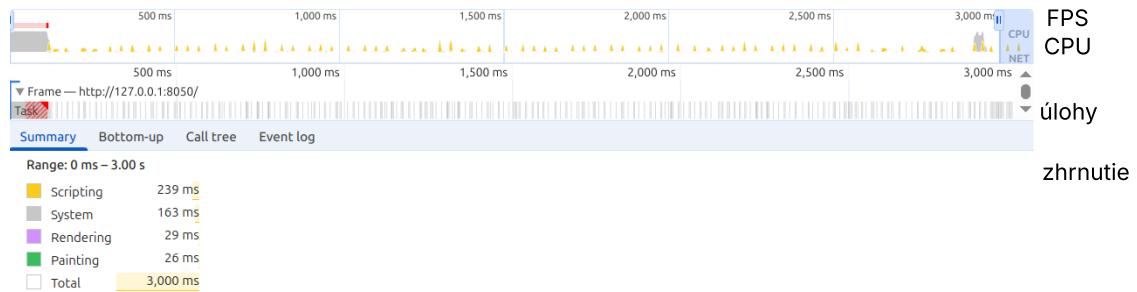
Animácia teda bežala hladšie, ale výkon aplikácie stále neboli výhovujúci, pretože animácia výrazne zatažovala procesor a hodnota snímkovej frekvencie stále miestami dosahovala príliš nízke hodnoty (v grafoch červené čiary v sekcií „FPS“).

Optimalizácia použitím iného spôsobu nastavenia polohy kamery

V sekcií 2.5 boli popísané dva rôzne spôsoby nastavenia polohy kamery vo frameworku **deck.gl**. V implementácii aplikácie bol použitý najprv prvý spôsob, teda aplikovanie transformácií na dátá, pretože bol jednoduchší a priamo vyplynul z prvej fázy implementácie

(vykreslenie mračna bodov v jazyku Python bez použitia frameworku). Ten bol následne zmenený na druhý spôsob, teda aplikovanie transformácií na kameru, pretože sa ukázalo, že tak dôjde k významnej optimalizácii.

Nový spôsob nastavenia polohy kamery bol analyzovaný pomocou profilingu rovnakej animácie ako v prechádzajúcej podsekcii.



Obr. 5.4: Profiling animácie pohybu vlaku s novým spôsobom nastavenia polohy kamery.

Z nameraných údajov vyplýva, že táto optimalizácia veľmi výrazne znížila zafaženie procesora. Zatiaľ čo pred optimalizáciou (obr. 5.3) sa počas trojsekudového intervalu vykonával javascriptový kód 2,745 sekúnd, po optimalizácii (obr. 5.4) to bolo iba 239 milisekúnd, čo predstavuje viac než desaťnásobné zníženie.

Dôležitou metrikou pri posudzovaní výkonnosti animácií je snímková frekvencia meraná v FPS (*frames per second*, počet vykreslených snímok za sekundu). V ideálnom prípade by sa snímková frekvencia mala blížiť 60 FPS, pretože vtedy sa animácia ľudskému oku zdá hladká [4].

Podľa vstavaného ukazovateľa snímkovej frekvencie v prehliadači Google Chrome dosahovala skúmaná animácia pred optimalizáciu zhruba 28 až 34 FPS, zatiaľ čo po nej 51 až 55 FPS. Aj v tomto ohľade teda nastalo významné zlepšenie.

5.2 Výkonnosť

5.3 Vyhodnotenie používateľskej prívetivosti

Kapitola 6

Záver

Literatúra

- [1] AHN, S. H. OpenGL Projection Matrix. *Songho.ca* online. Dostupné z: <https://www.songho.ca/>. [cit. 2025-05-01]. Path: OpenGL.
- [2] AHN, S. H. OpenGL Viewport Transform. *Songho.ca* online. Dostupné z: <https://www.songho.ca/>. [cit. 2025-05-01]. Path: OpenGL.
- [3] BALOCH, J. Image Perspective Distortion in JavaScript. *Medium* online. 10. marca 2023. Dostupné z: <https://blog.bitsrc.io/image-perspective-distortion-with-javascript-and-html-canvas-14627233623d>. [cit. 2025-05-03].
- [4] BASQUES, K. Analyze runtime performance. *Chrome for developers* online. Dostupné z: <https://developer.chrome.com/>. [cit. 2025-04-14]. Path: Docs; DevTools; Performance.
- [5] DE VRIES, J. *Coordinate Systems* online. Dostupné z: <https://learnopengl.com/>. [cit. 2025-05-01]. Path: Getting started.
- [6] FACULTY SCIENCE LTD. *Dash Bootstrap Components* online. Dostupné z: <https://dash-bootstrap-components.opensource.faculty.ai/>. [cit. 2025-04-24].
- [7] HARTLEY, R. a ZISSEMAN, A. *Multiple View Geometry in Computer Vision*. 2. vyd. Cambridge: Cambridge University Press, 2003. 152-156 s. ISBN 978-0-521-54051-3.
- [8] OPENCV. *Camera Calibration* online. Dostupné z: <https://docs.opencv.org/4.x/>. [cit. 2025-04-29]. Path: OpenCV-Python Tutorials; Camera Calibration and 3D Reconstruction.
- [9] OPENCV. Camera Calibration and 3D Reconstruction. *OpenCV 2.4.13.7 documentation* online. Dostupné z: <https://docs.opencv.org/2.4/>. [cit. 2025-04-29]. Path: OpenCV API Reference; calib3d. Camera Calibration and 3D Reconstruction.
- [10] OPENJS FOUNDATION. *Deck.gl* online. 2024. Dostupné z: <https://deck.gl/>. [cit. 2025-01-14].
- [11] OPENJS FOUNDATION. Performance Optimization. *Deck.gl* online. 2025. Dostupné z: <https://deck.gl/>. [cit. 2025-04-12]. Path: Docs; Developer Guide; Using the API.
- [12] OPENSFM. Coordinate Systems. *OpenSfM* online. Dostupné z: <https://opensfm.readthedocs.io/>. [cit. 2025-04-30]. Path: Geometric Models.
- [13] PLOTLY. *Dash Python User Guide* online. 2025. Dostupné z: <https://dash.plotly.com/>. [cit. 2025-01-17].

- [14] SNOWFLAKE INC. *Streamlit* online. 2024. Dostupné z: <https://streamlit.io/>. [cit. 2025-01-17].
- [15] STEMKOSKI, L. a PASCALE, M. *Developing Graphics Frameworks with Python and OpenGL* online. 1. vyd. Boca Raton: CRC Press, júl 2021. 344 s. ISBN 9781003181378. Dostupné z: <https://doi.org/10.1201/9781003181378>. [cit. 2025-05-01].
- [16] SUN, W. a COOPERSTOCK, J. Requirements for Camera Calibration: Must Accuracy Come with a High Price? In: IEEE. *2005 Seventh IEEE Workshops on Applications of Computer Vision (WACV/MOTION'05)* online. Los Alamitos, CA, USA: IEEE, Január 2005, sv. 1, s. 356–361. Dostupné z: <https://doi.org/10.1109/ACVMOT.2005.102>. [cit. 2025-04-29].
- [17] THE KHRONOS GROUP INC. WebGL Overview. *The Khronos Group* online. Dostupné z: <https://www.khronos.org/>. [cit. 2025-05-01]. Path: WebGL.
- [18] THE MATHWORKS, INC. What Is Camera Calibration? *MATLAB Help Center* online. Dostupné z: <https://www.mathworks.com/help/vision/ug/camera-calibration.html>. [cit. 2025-04-29]. Path: Image Processing and Computer Vision; Computer Vision Toolbox; Camera Calibration.
- [19] PRESTON-WERNER, T.; GEDAM, P. et al. *TOML v1.0.0* online. 1. novembra 2021. Dostupné z: <https://toml.io/en/v1.0.0>. [cit. 2025-04-11].
- [20] UBER ATG. *Autonomous Visualization System* online. Dostupné z: <https://avs.auto/>. [cit. 2025-04-11].