Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

3D rozhranie pre interaktívnu vizualizáciu histogramov

Diplomová práca

2026 **Bc. Daniel Chovanec**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

3D rozhranie pre interaktívnu vizualizáciu histogramov

Diplomová práca

Študijný program:	Informatika (2508T00)
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	doc. Ing. Štefan Korečko PhD.

Košice 2026 **Bc. Daniel Chovanec**

Abstrakt v SJ

Diplomová práca sa zaoberá knižnicou Ndmvr-core, ako časťou projektu NdmSpc. Práca je rozdelená do viacerých častí. V analytickej časti je rozobraný modul NdmVr so zameraním na výkon a hierarchiu použitých technológií. Analyzovaná je zároveň knižnica Jsroot z hľadiska kompatibility s inými riešeniami. V závere analytickej časti sú preskúmané prístupy k vykresľovaniu, opätovne s dôrazom na výkonnosť. Druhá časť práce sa venuje základnej myšlienke vizualizácie dát prostredníctvom konceptu hyperkocky a následnému návrhu komunikácie a samotnej vizualizácie. V nasledujúcej kapitole je popísaná implementácia navrhnutého riešenia, kde prepájame zavedené princípy s konkrétnym vypracovaním. Dôraz je kladený na vysvetlenie štruktúry riešenia na úrovniach komponentov, tried, dátových štruktúr a ich indexovania. V záverečnej kapitole sú prezentované výkonnostné testy riešenia. Súčasťou je aj spätná väzba od členov ROOT tímu, ako aj od ďalších respondentov, ktorým bolo riešenie predstavené.

Kľúčové slová v SJ

vizualizácia, Three.js, histogram, virtuálna realita, jsroot

Abstrakt v AJ

The master's thesis describes an Ndmvr-core library as part of the NdmSpc project. The thesis is divided into several chapters. An analysis of the NdmVr module, with an emphasis on performance and the hierarchy of technologies used. We also analyze the JSROOT library in terms of compatibility with other solutions. The final part of the analysis explores rendering approaches and techniques, with the same focus on performance. The second part of the thesis addresses the core idea of data visualization through the concept of a hypercube, followed by the design of communication and the visualization itself. The subsequent chapter describes the implementation of the designed solution, linking the established principles with their actual realization. Emphasis is placed on explaining the structure of the solution at the level of components, classes, data structures, and their indexing. The final chapter presents performance testing of the solution. It also includes feedback from members of the ROOT team, as well as from other respondents to whom the solution was presented.

Kľúčové slová v AJ

visualization, Three.js, histogram, virtual reality, jsroot

Bibliografická citácia

CHOVANEC, Daniel. *3D rozhranie pre interaktívnu vizualizáciu histogramov*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2026. 63s. Vedúci práce: doc. Ing. Štefan Korečko PhD.

79298

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE
DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

3D rozhranie pre interaktívnu vizualizáciu histogramov

3D Interface for Interactive Histogram Visualization

Študent: **Bc. Daniel Chovanec**
Školiteľ: **doc. Ing. Štefan Korečko, PhD.**
Školiace pracovisko: **Katedra počítačov a informatiky**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

1. Analyzovať aktuálnu implementáciu vizualizácie histogramov v softvéri NDMVR a podobné riešenia.
2. Navrhnúť a implementovať inovované riešenie pre interaktívnu vizualizáciu histogramov.
3. Riešenie realizovať so zreteľom na použitie v imersívnej virtuálnej realite a prepojenie so softvérovým rámcom JSROOT.
4. Návrh a implementáciu koordinovať s ďalšími riešiteľmi prác na softvéri NDMVR.
5. Implementované riešenie overiť a vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 24.04.2026
Dátum zadania diplomovej práce: 31.10.2025

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -12-

v.z.

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry, ako aj s využitím nástrojov generatívnej umelej inteligencie v súlade s Metodickým pokynom o záverečných a kvalifikačných prácach a Etickým kódexom študenta Technickej univerzity v Košiciach. Umelá inteligencia bola použitá ako podporný nástroj bez porušenia zásad akademickej etiky a autorskej integrity.

Poďakovanie

Na tomto mieste by som rád poďakoval svojmu vedúcemu práce za jeho čas a odborné vedenie počas riešenia mojej záverečnej práce. Rovnako by som sa rád poďakoval svojim rodičom a priateľom za ich podporu a povzbudzovanie počas celého môjho štúdia.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov

1 Úvod

Vizualizácia dát je dôležitou časťou práce s údajmi. Údaje môžu v závislosti od svojej nátury či účelu prezentácie nadobúdať rôzne formy. Jednou z nich sú **histogramy**; ich primárnym účelom je zobrazenie objemných dátových súborov v prípadoch, kedy by štandardný bodový graf pôsobil neprehľadne. Táto práca sa zaoberá prípadom, kedy súbor neobsahuje len veľké množstvo bodov, ale samotné body sú zložené z väčšieho počtu parametrov. Pre lepšiu predstavu o tom, ako rýchlo môže dátový súbor nadobudnúť vysokú dimenzionalitu, si predstavíme príklad monitorovania jázd taxíkov. V tomto scenári môže každý dátový bod nabrať množstvo dimenzií (parametrov), akými sú napríklad: počet pasažierov, dĺžka a čas trvania cesty, dátum vykonania jazdy, celková zaplatená suma či výška obslužného.

Hlavným nedostatkom štandardných vizualizácií je ich obmedzenie na tri zobrazované dimenzie, čo je predovšetkým dôsledkom praktických limitov ľudského vnímania priestoru. Ak chceme vizualizovať komplexnejšie dáta, sme prevažne odkázaní na ich rozdelenie do viacerých samostatných zobrazení. V takom prípade môže nastať u čitateľa strata súvislosti informácií, z dôvodu zvýšenej kognitívnej záťaže. Táto záťaž vzniká pri opätovnom prepojení vzťahov oddelených vizualizácií a procese vytvárania si vnútorného obrazu pri snahe o odvodenie požadovaného záveru z prezentovaných dát. Hoci je pomerne jednoduché zaviesť do vizualizácie dodatočné parametre (kde okrem veľkosti binov môžu hodnoty reprezentovať napríklad farba, tvar či ohraničenie), tento prístup nie je dobre škálovateľný. Tieto aditívne metódy sú totiž konečné a narážajú na svoje limity v momentoch, kedy nepoznáme vopred daný počet parametrov. To znamená, že by sme používateľovi nedokázali zaručiť plnú n-dimenzionalitu.

V práci nadväzujeme na existujúce riešenie **NdmVr**[1] (súčasť projektu NdmSpc) a analytický rámec **ROOT** [1], ktorý vyvíja tím pod záštitom inštitútu CERN. Napriek tomu, že sa tieto nástroje zaoberajú problematikou zobrazovania údajov, v oboch doposiaľ absentuje zadefinovaný štandardný prístup pre prácu s n-di-

[1] `https://root.cern/`

menzionálnymi dátami vo forme histogramov. Používateľ spracovávajúci veľkú dátovú sadu s koreláciami medzi jednotlivými parametrami je väčšinou neustále nútený prechádzať medzi rôznymi zobrazeniami. Často musí vyčerpávajúco hľadať v oddelených súboroch či dokonca čeliť potrebe manuálnej úpravy kódov makier, ktoré vizualizácie generujú.

Formulácia úlohy

Cieľom práce je vytvorenie vizualizačného jadra s rozhraním, ktoré by integrovalo všetky požadované parametre, čím by sme dokázali výrazne uľahčiť a zrýchliť analýzu dát aj hľadanie vzájomných korelácií. Pri vizualizácii viacerých parametrov súčasne je možné efektívnejšie a rýchlejšie rozpoznať celkovú distribúciu hodnôt naprieč celou dátovou sadou. To je dôležité najmä pri začiatku práce s novými dátami, kedy si používateľ chce najprv vytvoriť počiatočný obraz o získaných údajoch.

Zásadným rozdielom pri predkladanom riešení je preto silné rozhranie a s ním spojené praktické interakcie nad objektom histogramu. Práve tie prinášajú používateľom úsporu času a zbavujú ich rutinného listovania medzi súbormi či prepisovania kódov. Napriek základnej myšlienke sme stále viazaní na ľudské vnímanie troch dimenzií. Preto aj obdobné riešenia pristupujú k problému analogicky: rozdelením zobrazenia do segmentov, ktoré sú však navzájom prepojené a vytvárajú jeden koherentný n-rozmerný obraz.

Takto koncipovaná n-dimenzionálna vizualizácia poskytuje vyššiu mieru flexibility. V prípade, ak potrebujeme porovnávať konkrétne parametre, znamená to, že ich dokážeme systematicky rozdeliť do segmentov a získať tak požadované porovnania v jednom čase. Ak sa vrátime k príkladu jázd taxíkov, môže nás zaujímať porovnanie diaľky cesty voči cene a súčasne čas trvania cesty voči obslužnému. Navrhované riešenie umožňuje vytvoriť segmenty, kde porovnávame vnútorné hodnoty nezávislých párov parametrov, ktoré však zostávajú prepojené s hodnotami párov iných typov.

Dôležitou vlastnosťou tejto práce je zároveň transparentnosť riešenia, keďže v týchto prípadoch je pre používateľov nadmerne kritická dôveryhodnosť spojená s korektnosťou vizualizácie. Je preto podstatné, aby bolo riešenie voľne dostupné v plnej verzii vrátane zdrojových kódov a podrobne zdokumentované pre používateľov, ktorí si ho chcú preveriť či pochopiť základy jeho fungovania a používania. V neposlednom rade musí byť systém čo najvýkonnejší, keďže dnešné dátové sady ľahko dosahujú rozmery rádovo v gigabajtoch. Nevyhnutnou súčas-

ťou práce je preto preskúmanie možností vykresľovania s cieľom využiť tie, ktoré ponúkajú najlepšiu syntézu výkonnosti, praktickosti a stability.

2 Analýza

Táto kapitola sa zaoberá analýzou zadaného problému podľa pokynov školiteľa a zadávacieho listu. V prvom kroku sa zameriame na existujúce riešenie knižnice NdmVr. Druhá podkapitola sa zameriava na použitie knižnice **JSROOT**[2] ako nástroja na vytvorenie objektov vizualizácie. Téma použitia tejto knižnice pokračuje v kapitole implementácie riešenia, pričom táto podkapitola sa priamo zameriava na experimentálne overenie prvotnej myšlienky vykreslenia objektov cez túto knižnicu. Posledná podkapitola je zameraná na vykreslenie histogramu, konkrétne na bázové vykreslenie kvádrov v scéne, s dôrazom na získanie, čo najlepšieho výkonu, naprieč rôznymi riešeniami. Stanovené tvrdenia sú v druhej a tretej podkapitole taktiež podložené experimentálnou implementáciou.

2.1 Knižnica NdmVr

V tejto kapitole sa zameriame na predchádzajúcu implementáciu knižnice NdmVr [1]. Postavená je na rámci React, pričom je dodatočne manažovaná Vite vývojárskými nástrojmi. Týmto spôsobom je možné vytvárať demo aplikácie. React komponenty sú taktiež exportované do npm (Node Package Manager) modulu, prostredníctvom ktorého ich je možné využiť v separátnom riešení. Podrobný opis všetkých komponentov knižnice je možné nájsť v prácach[3, 4], preto sa ďalej zameriame len na časti jadra spojené s vykresľovaním, ktoré je dôležité analyzovať pre vytvorenie nového riešenia.

2.1.1 Komunikácia

Primárnym komponentom vykresľovania je NdmVrHistogram. Objekty konfigurácie sú tomuto komponentu posielané prostrednictvom RxJs subjektov, tie prenášajú informácie o veľkosti, pozícii a téme histogramu. Taktiež obsahujú definované rozmedzie zobrazenia histogramu a funkcie pre spracovanie interakcií s binmi.

Ďalšou technikou predania dát predstavuje React Context. Cez Context sa distribuujú informácie o aktuálne označených binoch, o zobrazovanej časti histogramu a funkcie, ktoré reagujú na zmenu stavu. Tretím spôsobom komunikácie predstavuje Hook `useNdmVrRedirect` a Hook `useNdmVrLocal`, čo sú separátne úložiská využívajúce lokálne úložisko prehliadača, vďaka čomu je možné informácie distribuovane predávať do všetkých modulov aplikácie. Dôvodom použitia tohto úložiska je možnosť uložiť stav aplikácie medzi reláciami.

Takto implementovaný komunikačný model spĺňa funkcionálne požiadavky, no takéto kombinovanie vzájomne neprepojených komunikačných kanálov prináša množstvo nevyžiadaných vedľajších účinkov. Hlavným problémom je rozdielny životný cyklus jednotlivých hodnôt. Pri použití RxJs subjektov sú hodnoty distribuované odoberateľom okamžite. Naopak, Hook useState prepojený na React Context aktualizuje stav až v nasledujúcom snímku vykreslenia, čo riadi React. Hook-y spojené s lokálnym úložiskom prehliadača React priamo neriadi, len na ne reaguje.

Takéto rozdiely v životných cykloch spôsobovali chyby časovania pri aktualizáciách stavu. V dôsledku toho na ne React v mnohých prípadoch reagoval nesprávne alebo dochádzalo k zbytočnému opätovnému prekresľovaniu komponentov. Navyše je takýto dizajn komunikácie často krát ťažko čitateľný a mätúci, primárne pre vývojárov, ktorí s týmto riešením začínajú pracovať.

2.1.2 Vykresľovanie

Komponent NdmVrHistogram je zložený z viacerých A-Frame entít. Medzi ne patria elementy `a-entity` a `a-text` slúžiace pre vykreslenie osí histogramu a plátna pre 2d obsah. Primárne sú však `a-box` elementy, ktoré reprezentujú objekty binov histogramu.

Hodnoty pozície a veľkosti jednotlivých binov sú prepočítané v pomocnej triede HistogramReactFactory. Na základe dimenzionality histogramu je prepočítané pole elements, obsahujúce údaje jednotlivých binov. Hodnoty pozície a veľkosti sú vypočítané za použitia metód na **JSROOT** objekte histogramu.

Jedným z dizajnových nedostatkov je existencia troch separátnych metód pre rozdielne dimenzionality histogramu. Toto rozdelenie síce reflektuje prístup použitý v knižnici **JSROOT**, no tam sú tieto typy oddelené z dôvodu rozdielnej povahy vizualizácie týchto histogramov, čo je sprevádzané inými možnosťami vizualizácie. V tomto prípade sa však vizualizácia na základe dimensionality nelíši, preto takáto implementácia zavádza zbytočnú duplicitnu logiku.

Ďalším problémom je vysoká neefektivita využitia pamäte. Pre každý bin

sa do poľa elements ukladajú informácie identifikátora, pozícia na osi a ďalšie atribúty. Celkovo to predstavuje 50 atribútov rozdelených do 4 objektov. Tieto informácie sú navyše uložené aj ako atribúty HTML elementu, čo znamená, že musia byť serializované do reťazca znakov. Dodatočné údaje pre každý bin tak predstavujú približne 2,2 kB pamäte. Tieto dáta sú navyše odovzdané do komponentu `binth`, ktorý je prepojený s elementom binu. To znamená, že tieto dáta a s nimi spojená priestorová náročnosť je zdvojnásobená.

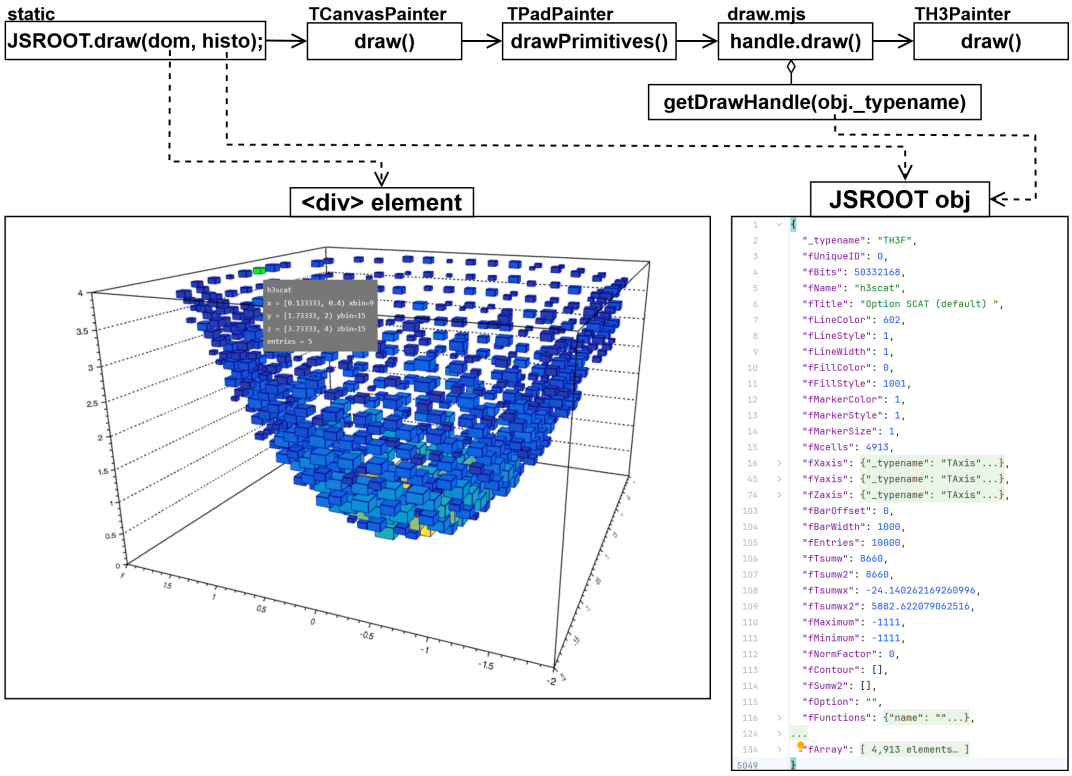
Bin reprezentovaný elementom `a-box`, je doplnený o element `draw-box`, ktorého úlohou je vykreslenie ohraničenia binu. Element `draw-box` vytvára skupinu čiar, kde každá z čiar predstavuje separátny objekt v scéne. Vo výsledku, objekt binu spolu s jeho ohraničeniami sa skladá z 13 objektov. To znamená, že počet objektov v scéne je 13-násobkom počtu binov v histograme, čo zavádza nadmernú záťaž pre systém.

2.2 Vykreslenie objektov cez JSROOT

Pre získanie funkcionality vykreslenia ROOT objektov cez knižnicu **JSROOT** sme najprv museli nájsť dôkaz, že 3D Three.js objekty sú abstraktovateľné z internej architektúry **JSROOT**-u. Na to sme z github repozitára naklonovali lokálnu verziu, ktorú sme dokázali upravovať podľa potreby. Samotná architektúra a hierarchia komponentov sú pomerne komplexné. Základnou triedou vykresľovaných objektov je `ObjectPainter`, to bol teda náš začiatočný bod. Postupným prechádzaním cez dedičské triedy `BasePainter`, `ObjectPainter` a `THistPainter` sme sa dopracovali k triede `TH3Painter`, ktorej úlohou je vykresliť 3-dimenzionálny histogram. V tejto triede bol pre nás začiatočný bod metóda `redraw`, tá je volaná používateľom, ak chce objekt vykresliť na div element. V tejto metóde je volaná metóda `create3DFrame`, ktorej úlohou je vytvorenie a pridelenie Three.js scény, taktiež pridelenie pomocných metód pre navigáciu a interakcie. Podstatná pre nás však je metóda `draw3DBins`, ktorej úlohou je predovšetkým vytvorenie 3D objektu histogramu. Samotný objekt je potom pridelený do scény volaním zdedenej metódy `add3DMesh` z triedy `FramePainter`. Pre jednoduchosť dôkazu sme metódu `draw3DBins` doplnili o extrahovanie 3D objektu cez náš RxJs subject. Tento subject už je pod našou kontrolou, vieme ho teda začať odoberať. Získaný 3D objekt následne stačí jednoducho pridať do základnej Three.js scény pristúpením cez jej asociovanú A-Frame entitu. Takto upravenú knižnicu **JSROOT** ďalej bolo potrebné len skompilovať a prelinkovať na naše riešenie príkazom `npm link`. Tým sme docielili dôležitý dôkaz, že 3D objekty **JSROOT**-u

sú extrahovateľné a prepoužiteľné pre naše riešenie. Toto riešenie však fungovalo len s lokálnou, upravenou kópiou **JSROOT**-u, čo nie je korektný prístup, no pre dôkaz to plne stačilo.

Hierarchiu modulov sme skúmali ďalej, tu bolo dôležité pochopenie významu jednotlivých modulov a komplexné prepojenia medzi nimi. Tieto poznatky sme zúžitkovali v druhom prístupe k vizualizovaniu objektov cez **JSROOT**. Globálna funkcia `redraw` slúži pre inicializáciu, vytvorenie a vykreslenie objektu do div elementu, pričom jej návratová hodnota je inštancia `FramePaintera`. Z vytvorenej závislosti, ktorú sme videli v implementácii metódy `redraw`, kde inštancii `FramePainter` boli pridelené metódy pre interakciu a nimi vytvorené samotné objekty scény a podobne. Asynchrónnou metódou `redraw` sme mali prístup k inštancii `FramePaintera`, pričom naše poznatky boli správne, keďže cez túto inštanciu sme sa dokázali dopracovať k základnej scéne. Iteratívnym filtrovaním grafu scény sme sa dopracovali k objektu `InstancedMesh`, ktorý obsahoval biny histogramu. Taktiež sa nám podarilo odfiltrovať objekty `LineSegments`, do ktorých boli zabalené objekty `InstancedBufferGeometry`, ktoré vizualizovali osi histogramu. Tieto vyfiltrované objekty sme jednoducho presunuli zo scény **JSROOT**-u do našej. Týmto sme získali možnosť vykresliť ROOT objekty bázovou neupravenou **JSROOT** knižnicou v našej scéne. Opísanú hierarchiu volania `draw` je možné vidieť na obrázku 2.1.



Obrázok 2.1: Simplifikovaný diagram volania draw z knižnice JSROOT

Osobitným problémom však bolo pridanie interakcie. To z dôvodu, že extrahovať sa nám podarilo čistý 3D objekt, bez pomocných metód a interakcií ktoré sme extrahovaním stratili. Na objekte `InstancedMesh` sme však ešte stále dokázali získať index inštancie z poľa. Dôležité však je, že **JSROOT** implementácia výsledné pole inštancí komprimuje na pole binov, ktoré majú byť zobrazené[5]. Táto implementácia musí explicitne vypísať a pamätať si prevedenie indexov z poľa inštancií do indexov používaných pre jednoznačnú identifikáciu binov, tie sú uložené v poli `bins`. Toto pole má dĺžku počtu viditeľných binov, kde index vstupov zodpovedá indexu instanceId z poľa `InstancedMesh`, hodnotou vstupu je index binu v histograme podľa ROOT štandardu. Toto pole dokážeme využiť tak, že index z poľa inštancií získaného raycasterom prevedieme na index binu histogramu, s ktorým vieme ďalej pracovať podľa prípadu interakcie s binom.

2.3 Vykreslenie histogramu

Ako základ analýzy vykresľovania bola použitá implementácia, ktorú vypracoval školiteľ. Tá obsahovala dva nové komponenty, histogram-skor komponent a pseudo-histogram komponent. Rozdiel medzi nimi je, že histogram-skor komponent vykresľuje histogram na základe url adresy, z ktorej načíta objekt histogramu, naopak pseudo-histogram vykresľuje falošný histogram, čo znamená, že

je vykreslený na základe vstupných parametrov počtu binov na každej z osí. Tieto histogramy využívajú balík aframe-instanced-mesh.[2] Tento balík používa techniku inštancovania, pomocou ktorej procesor dokáže viacero zdanlivo oddelených objektov poslať grafickej karte ako jeden objekt s rozličnými parametrami pre jednotlivé inštancie základného objektu. Takouto "batch" skupinou podstatne znížime limitujúci faktor komunikácie medzi procesnými jednotkami. [6] Balík bol nainštalovaný a použitý takto. Vo vnútri entity histogramu je pridaný objekt `a-box` s atribútom `instanced-mesh` a označený identifikátorom. Vo vnútri histogramu sú následne jednotlivé biny označené atribútom `instanced-mesh-member`, čo zabezpečí, že všetky biny histogramu budú v jednej `instanced-mesh` skupine. Týmto balíkom sme docielili signifikantný nárast z pohľadu FPS (Frames Per Second), no naopak sme stratili možnosť interagovať s jednotlivými binmi. Z tohto dôvodu bola pre každý bin pridaná jeho neviditeľná kópia tzv. hitbox. Týmto spôsobom sme dokázali interagovať s jednotlivými binmi histogramu a udržali sme nárast FPS, aj keď už nie tak veľký. Výsledkom zlepšenia však ešte stále nebol ten očakávaný.

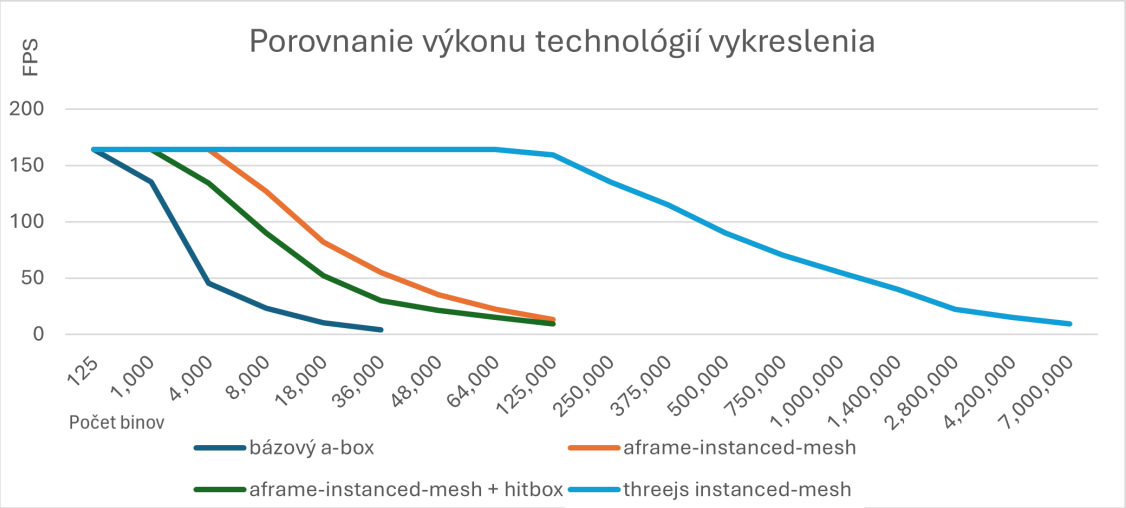
Pre zlepšenie výkonu sme skúšali meniť materiál binov histogramu. Výmenou materiálu za jednoduchší sme dosiahli zlepšenie, no jednalo sa len o zopár snímok za sekundu. Razantný pokles bol viditeľný po pridaní neviditeľných hitboxov, preto sme ich skúsili nahradiť nami vytvorenými boxmi, tým pádom sme ich dokázali mať pod kontrolou. Vďaka tomu sme všetky hitboxy vedeli zaradiť do BVH (Bounding Volume Hierarchy) stromovej štruktúry[7], s použitím knižnice three-mesh-bvh[3]. Nakoľko sme uvažovali nad myšlienkou, že problémom pri hitboxoch nemusí byť vykresľovanie, nakoľko sú neviditeľné. Problémom naopak môže byť raycaster, ktorý musí kontrolovať kolíziu so všetkými objektmi v scéne. Preto sme skúsili použiť hierarchickú štruktúru objektov, známu ako BVH, aj napriek tomu, že táto technika je používaná pre tzv. ray tracing, napriek tomu, samotnú myšlienku dokážeme prepoužiť pre ray casting. Ak sú tieto objekty v stromovej štruktúre BVH, sú rozdelené do skupín, raycaster tak kontroluje kolíziu so skupinou [8], nie priamo s objektom. Týmto spôsobom raycaster zistí, s ktorým objektom sa zrazil, nie v n, ale v log2(n) počte iterácií. Po implementácii BVH nás výsledok prekvapil, keďže výkon z pohľadu FPS bol o niečo horší, ako bez použitia BVH. Tento výsledok mohol byť zapríčinený vzniknutým tzv. overhead-om, ktorý mohol byť príliš veľký. Príčina, ktorá však bola pravdepodobnejšia je, že lúč posielaný z raycastera sa nezastavuje na prvom objekte, ale

[2] `https://www.npmjs.com/package/aframe-instanced-mesh`
[3] `https://www.npmjs.com/package/three-mesh-bvh`

pokračuje ďalej a taktiež kontroluje objekty, ktoré môžu byť za objektom prvého stretu. Kvôli tomu sa algoritmus hľadania v strome musí rozvetviť do viacerých iterácií, čo v konečnom dôsledku môže zapríčiniť pokles výkonu oproti prípadu, kedy sme BVH nepoužili. Rovnako tak, prideľovanie objektov do skupín bolo automatické, čo znamená, že rozdelenie nebolo optimálne. Po tomto pokuse sme objavili ukážku v Three.js, ktorá využívala `InstancedMesh`. Čo nás prekvapilo, bol výkon, kde pri 10 000 objektoch sa FPS držalo hodnoty 130.[4] Na základe tejto ukážky sme zhodnotili, že balík aframe-instanced-mesh musí byť napísaný neoptimálne, keďže v našom prípade použitia sme nedokázali získať požadované zrýchlenie. Preto sme sa rozhodli použiť bázový `InstancedMesh` z knižnice Three.js, to bolo možné, nakoľko knižnica A-Frame je postavená na Three.js.

Pre implementáciu prechodu z balíka aframe-instanced-mesh na bázový `InstancedMesh` sme museli zmeniť algoritmus vykreslenia histogramu. Na začiatku vykresľovania histogramu sme zadefinovali geometriu BoxGeometry z Three.js. Rovnako tak sme zadefinovali materiál MeshPhongMaterial z Three.js a vytvorili si základný objekt Object3D, ktorý A-Frame rovnako používa ako základný typ pre zobrazované objekty. Tento objekt bude slúžiť ako šablóna pre všetky biny histogramu. Následne sme vytvorili inštanciu `InstancedMesh` z Three.js, ktorej sme ako parametre predali geometriu, materiál a počet inštancií ktoré zobrazí, tento počet sme vyrátali z objektu histogramu. Po predpríprave sme mohli prejsť do hlavného cyklu, v ktorom sú jednotlivé biny pridané do objektu `InstancedMesh`. Podľa pozícii na osiach, je pozícia a veľkosť binu vypočítaná na základe ohraničenia binu na každej z osí. Vo výsledku sú teda súradnice v A-Frame scéne rovnaké, ako sú hodnoty súradníc na osiach histogramu, samozrejme relatívne k začiatočnému bodu histogramu. Pozíciu a veľkosť sme následne mohli priradiť šablónovému 3D objektu a tento objekt pridať do inštancie instanced-mesh. Po zmene algoritmu vykresľovania sme dosiahli výsledok, s ktorým sme boli nadmieru spokojný, zodpovedajúci výkonu, ktorý sme videli v ukážke instanced-mesh komponentu. S takto signifikantne zvýšeným výkonom sme vytvorili test, kde sme skúmali maximálny počet zobrazovaných binov. Meranie bolo vykonané na zariadení Lenovo disponujúcim operačným systémom Windows 10, 16 GB operačnej pamäte, procesorom AMD Ryzen 5 5600H a grafickou kartou Nvidia RTX 3060 Laptop. Výsledky tohto merania môžete vidieť na grafe 2.2.

[4] `https://threejs.org/examples/?q=inst#webgl_instancing_performance`



Obrázok 2.2: Graf porovnania výkonu

2.4 Výsledok analýzy

Z vypracovanej analýzy, ktorá zahŕňala rozbor predchádzajúceho riešenia **NdmVr**, preskúmanie knižnice **JSROOT** a testovanie pokročilých metód vykresľovania, vyplývajú tieto zásady:

- **Komunikácia** - Musí pozostávať z jednotného a dostatočne flexibilného riešenia na zachovanie čistoty kódu a podporu univerzálnosti.
- **Štruktúra objektov** - Vykresľované entity by mali pozostávať z minimálneho počtu samostatných objektov. Ich atribúty nie je vhodné uchovávať ako HTML atribúty, kvôli efektívnosti využitia systémových prostriedkov.
- **Integrácia JSROOT** - Vykreslenie histogramu do separátnej 3D scény prostredníctvom knižnice JSROOT je realizovateľné, čo ponúka možnosť zahrnutia do finálneho riešenia.
- **Optimalizácia** - Technika inštancovania je mimoriadne efektívna, jej použitie je nevyhnutné.
- **Detekcia kolízií** - Techniku BVH na kontrolu priesečníkov lúča je prospešné podrobiť ďalšiemu prevereniu, vzhľadom na preukázanie náročnosti na systémové zdroje pri týchto kontrolách.

Vytvorené experimentálne implementácie je taktiež možné použiť vo finálnom riešení, čomu sa ďalej venujú podkapitoly 5.2 a 5.7.

3 Návrh riešenia

Po evaluácii softvérového riešenia NDMVR pre interaktívne zobrazenie histogramov, zhodnotení nedostatkov, nevyhnutných vlastností a obmedzení, sme prešli k návrhu nového riešenia. Primárna požiadavka spočívala v odstránení React-u z techologického základu (tech-stack) riešenia, ten je teda z hierarchie technológií odstránený. Rovnako bol prehodnotený prínos rámca A-Frame do nášho riešnia kde alternatívou, by bolo použitie čistej knižnice Three.js, bez väzby na iný rámec, ktorý ju abstraktuje. Pre jednoduchší a kontinuálny prechod, sme rámec A-Frame v technologickom základe ponechali. Použitie je však limitované pre testovanie implementácie spojené s vytvorením minimalistického dema. Je to z priameho dôvodu, že takto máme viaceré esenciálne funkcionality tzv. „out of the box“. Dôležitou vlastnosťou však je, že komponenty v jadre sú písané výlučne použitím knižnice Three.js. Takéto komponenty tak máme plne pod kontrolou. Tento predpoklad tak dodatočne podporuje potenciál integrácie do iných riešení. Základ riešenia teda pozostáva z čistého HTML a JavaScriptu s použitím knižnice Three.js.

3.1 Hyperkocka

Základnou podmienkou nášho riešenia je vytvorenie vizualizácie n-dimenzionálneho histogramu. Ako základ, sme zvolili koncept hyperkocky, tiež nazývaná, ako n-dimenzionálna kocka. Základom vizualizácie histogramu v 3D, je umiestnenie jednotlivých binov vyobrazených ako kvádre do priestoru, podľa hodnôt jeho parametrov. V našom prípade celá myšlienka spočíva v umiestnení kvádrov dovnútra iných kvádrov, kde obe majú zadané hodnoty pre iné parametre. Takto dokážeme vytvoriť hierarchickú štruktúru, kde vnútorný kváder má hodnoty pre svoje vnútorné parametre, zároveň dedí hodnoty parametrov kvádra, v ktorom je umiestnený, no taktiež na základe jeho hodnôt definuje svet kvádru na nižšej vrstve v tejto hierarchii[9].

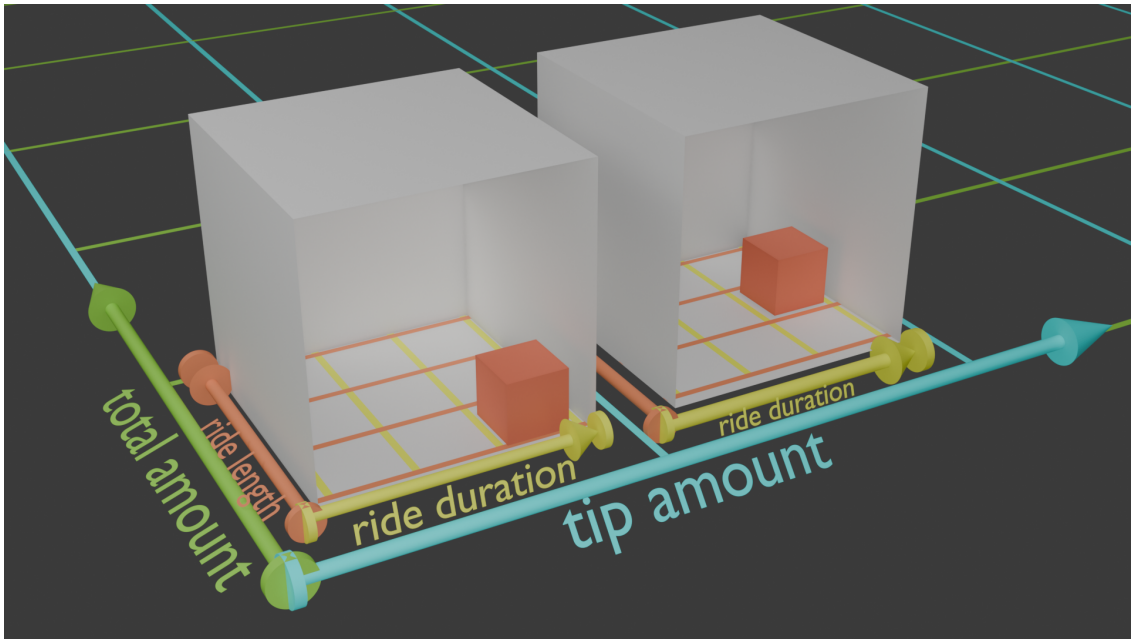
Pre objasnenie tohto systému sa vrátime k príkladu z úvodu. Na obrázku 3.1

je ukážka, ako môže vyzerať vizualizácia s pridaním hierarchie hyperkocky. Histogram je rozdelený do dvoch vrstiev, z ktorých každá obsahuje dva parametre:

1. **Prvý bin:** cestujúci zaplatil celkovo 0–10 €, z čoho 0–1 € bolo obslužné.
2. **Druhý bin:** cestujúci zaplatil celkovo 0–10 €, z čoho 1–2 € bolo obslužné.

Pri druhej (nižšej) vrstve postupujeme tak, že bin dedí hodnoty parametrov z vyššej vrstvy:

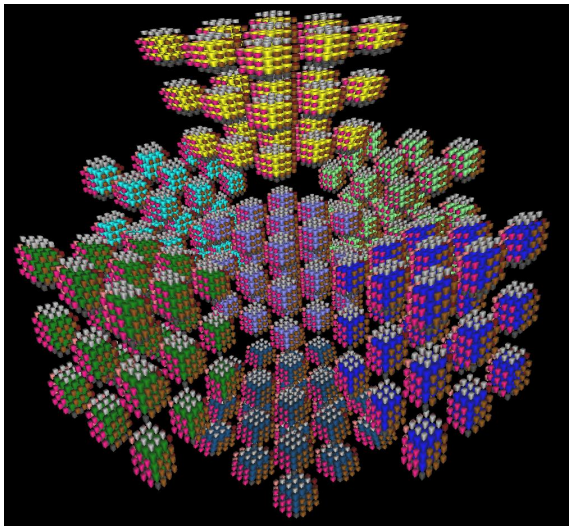
- **Prvý bin nižšej vrstvy:** je zložený z cesty s trvaním 2–3 minúty a dĺžke **0–1 km**, pričom vďaka dedičnosti vieme, že ma priradené **0–1 euro obslužného** a 0–10 € celkovej sumy.
- **Druhý bin nižšej vrstvy:** analogicky tvorí jazda v trvaní 2–3 minút a dĺžke **1–2 km**, ktorá dedí hodnoty **1–2 eur obslužného** a 0–10 € celkovej sumy.



Obrázok 3.1: Vizualizácia hierarchického histogramu s 4 parametrami

Takéto vrstvenie dimenzií môžeme nájsť napríklad pri experimentálnych vizualizačných nástrojoch pre dopyty na databázu[10], v tomto prípade sa však jedná o 2D vizualizáciu. Druhým, pokročilejším príkladom je vizualizácia n-rozmerných funkcií do 3D priestoru prostredníctvom tzv. hyperbuniek.[11] Práve prechod do 3D priestoru umožňuje využiť efektívnosť paralelného spracovania obrazu na základe rýchleho rozpoznávania vzorov podľa proximity, farby či textúry[12]. Týmto spôsobom taktiež dokážeme prepoužiť algoritmus vykreslenia

základného, maximálne 3-dimenzionálneho histogramu cez rekurziu tak, aby vykreslil takto zadefinovaný n-dimenzionálny histogram. Vstupný n-dimenzionálny histogram, teda rozdelíme do „vrstiev“ po 3 parametroch. Výsledne zobrazený vstup z dát teda tvorí hierarchia týchto vrstiev, z ktorých je možné si vizuálne jednoducho vyskladať pôvodný vstup. Markantnou výhodou takejto vizualizácie je systematické uloženie výsledných objektov. Takéto uloženie je podľa Gestaltových princípov[13], vhodné pre používateľa, nakoľko sa dokáže jednoducho navigovať v usporiadaných dátach vďaka tomu, že väčšina používaných parametrov pozostáva zo spojitých hodnôt. Používateľ taktiež dokáže vidieť distribúciu všetkých dát v jednom čase. To znamená, že dokáže rýchlo a intuitívne prejsť k požadovaným hodnotám naprieč všetkými parametrami. Zároveň, ak postupuje opačným spôsobom a sleduje najprv distribúciu vstupov, vidí ich pre všetky parametre naraz, bez potreby preklikávania sa medzi inými parametrami, vytváraním animácie, či zmeny tvaru objektov pre niektorý z ďalších parameterov, čo by viedlo k vizuálnej únave, keďže tieto zmeny vyžadujú intenzívne sústredenie, vo výsledku by to mohlo prejsť až do tzv. slepoty voči zmenám, kde používateľ by sa snažil zachytiť jednotlivé zmeny.[14] Na obrázku 3.2 je priložená možná vizualizácia histogramu v systéme n-parametrovej vizualizácie.



Obrázok 3.2: Vizualizácia hyperkocky. [15]

3.2 Komunikácia

Jednou z požiadaviek je uskutočnenie prepoužiteľného riešenia, ktoré je prispôsobiteľné viacerým koncovým aplikáciám. Na tento účel je potrebné silné rozhranie výsledných komponentov so zvyškom koncovej aplikácie. Samotný A-Frame

má však nepraktické rozhranie pre dynamické prostredie. Toto rozhranie funguje cez HTML atribúty elementov, kvôli tomu je obmedzený na určité typy hodnôt, rovnako tak na samotné objekty, ktoré sa nedajú jednoducho predať do vnútra A-Frame entity[16]. Z tohto dôvodu, je základná a väčšinová časť komunikácie postavená, na knižnici RxJs. Tá ponúka perfektnú zhodu s požiadavkami na naše minimalistické prostredie. Preto sme sa rozhodli komunikáciu medzi jednotlivými komponentmi, rovnako tak komunikáciu s klientom, riešiť cez rozhranie na vytvorených RxJs subjectoch. Pri všetkých subjectoch budeme používať tzv. singleton návrhový vzor[17], pre zaručenie stálej správnej referencie na rovnaký kanál správ, navyše bude pre používateľa jednoduchšie získať túto referenciu z ktorejkoľvek časti kódu prostredníctvom exportovanej "getter" metódy.

3.2.1 Získanie objektu vizualizácie

Pre získanie objektu sme zadefinovali 2 hlavné spôsoby pre ich rozdielne použitia. Prvým je využitie protokolu HTTP. Získanie objektu týmto spôsobom je pre používateľa jednoduché, keďže vie vytvoriť dopyt na server, ktorý vráti požadovaný objekt. Druhým spôsobom je získanie objektu pomocou websocket spojenia. Toto bola taktiež jedna z kľúčových požiadaviek na to, aby riešenie dokázalo byť vysoko dynamické. Websocket spojením taktiež získame mnoho pokročilých možností, ako je rýchle aktualizovanie objektu pre vizualizáciu, čo vedie k použitiam pre strímovanie objektov z natívnej ROOT aplikácie do nášho vizualizačného nástroja, vytvorenie animácií, či dokonca vytvorenie zdieľaného virtuálneho prostredia.

3.3 Vizualizácia objektu

Základným objektom pre vizualizáciu je objekt histogramu. Z pohľadu implementácie sme sa ho rozhodli napísať ako JS triedu, používajúcu len Three.js. Takúto triedu môžeme neskôr zabaliť do A-Frame entity, alebo do entity ľubovoľného rámca postaveného na Three.js. Implementáciu sme rozdelili do viacerých krokov. Prvým je základná kocka, ktorou dokážeme potvrdiť korektnosť vytvoreného technologického základu, čo znamená, že kocku má tvoriť Three.js objekt zabalený do A-Frame entity a má byť schopný komunikovať prostredníctvom RxJs. Ďalším krokom je vytvorenie algoritmu pre vykreslenie samotného histogramu, pričom tento histogram spočiatku limitujeme na 3 dimenzie. Na vizualizovanom základnom histograme dokážeme vykonať predbežné testovanie, na zhodnotenie použiteľnosti a výkonu riešenia. Za predpokladu solídneho, spoľahlivého

výsledku prejdeme na transformáciu algoritmu na rekurzívny, schopný vykresliť n-dimenzionálny histogram.

Pre priblíženie sa k vizualizácii ktorú ponúka **JSROOT** sme zhodnotili, že je potrebné vykreslenie ohraničenia binov. Tieto ohraničenia slúžia na ľahšie vizuálne separovanie jednotlivých binov, keďže ak majú rovnakú farbu, tak navzájom splývajú[18]. Ohraničenie je taktiež vykreslené aj z binu vrchného histogramu, čím získame pomôcku pre separáciu jednotlivých binov, ak zobrazujeme vnútorné histogramy nižších parametrov.

3.3.1 Inštancovanie

Počas testovania sme hľadali iné riešenia a technológie vykresľovania. To nás doviedlo k technike inštancovania objektov. Táto technika je implementovaná v separátnom balíku aframe-instanced-mesh. Jeho účelom je samotné inštancovanie základných A-Frame objektov. Jeho použitím je možné získať razantné zrýchlenie vo vykresľovaní, no samotný balík nie je napísaný korektne. To sa nám potvrdilo, keďže pri jeho použití sme získali nárast výkonu, no inicializačné vykreslenie trvalo dlho. Naopak pri použití inštancovania z balíka Three.js[5] v demách pre túto techniku, bol výkon lepší, taktiež inicializačné vykreslenie bolo priam instantné. Preto sme sa rozhodli komponent histogramu napísať s pôvodným algoritmom a použitím inštancovania z balíka Three.js. Týmto spôsobom taktiež zabezpečíme kontinuálnu integritu s triedou, ktorá používa čistý Three.js.

3.4 Vizualizácia prostredníctvom rámca JSROOT

V počiatku projektu NDMSPC bola jednou zo základných požiadaviek vizualizácia dát prostredníctvom knižnice **JSROOT**. Táto požiadavka bola pôvodne považovaná za jedinú potrebnú pre vizualizáciu objektov, čo vychádzalo z predpokladu, že **JSROOT** je, rovnako ako A-Frame, postavený na knižnici Three.js, ktorá tvorila základ projektu NDMVR. Uvedená požiadavka však nebola počas celého obdobia implementácie NDMVR naplnená a postupne sa od jej realizácie upustilo.

Počas analýzy riešenia **JSROOT** v kontexte dostupných prístupov a technológií tejto práce sme sa rozhodli k tejto požiadavke opätovne vrátiť. Umožnila nám to dizajnová zásada navrhovaného riešenia, ktorá spočívala v prechode z rámca A-Frame na bázovú knižnicu Three.js[5]. Tento prechod nám poskytol významné poznatky o princípoch a fungovaní knižnice Three.js, ktoré bolo možné priamo uplatniť pri implementácii požadovanej funkcionality.

Významným predpokladom úspešnej realizácie bolo aj užšie prepojenie autorského tímu s vedeckým inštitútom CERN, ktorého členmi sa v danom období stali autor práce spolu s Ing. Natanaelom Prokopom a školiteľom tejto práce.

3.5 Interakcia s objektom

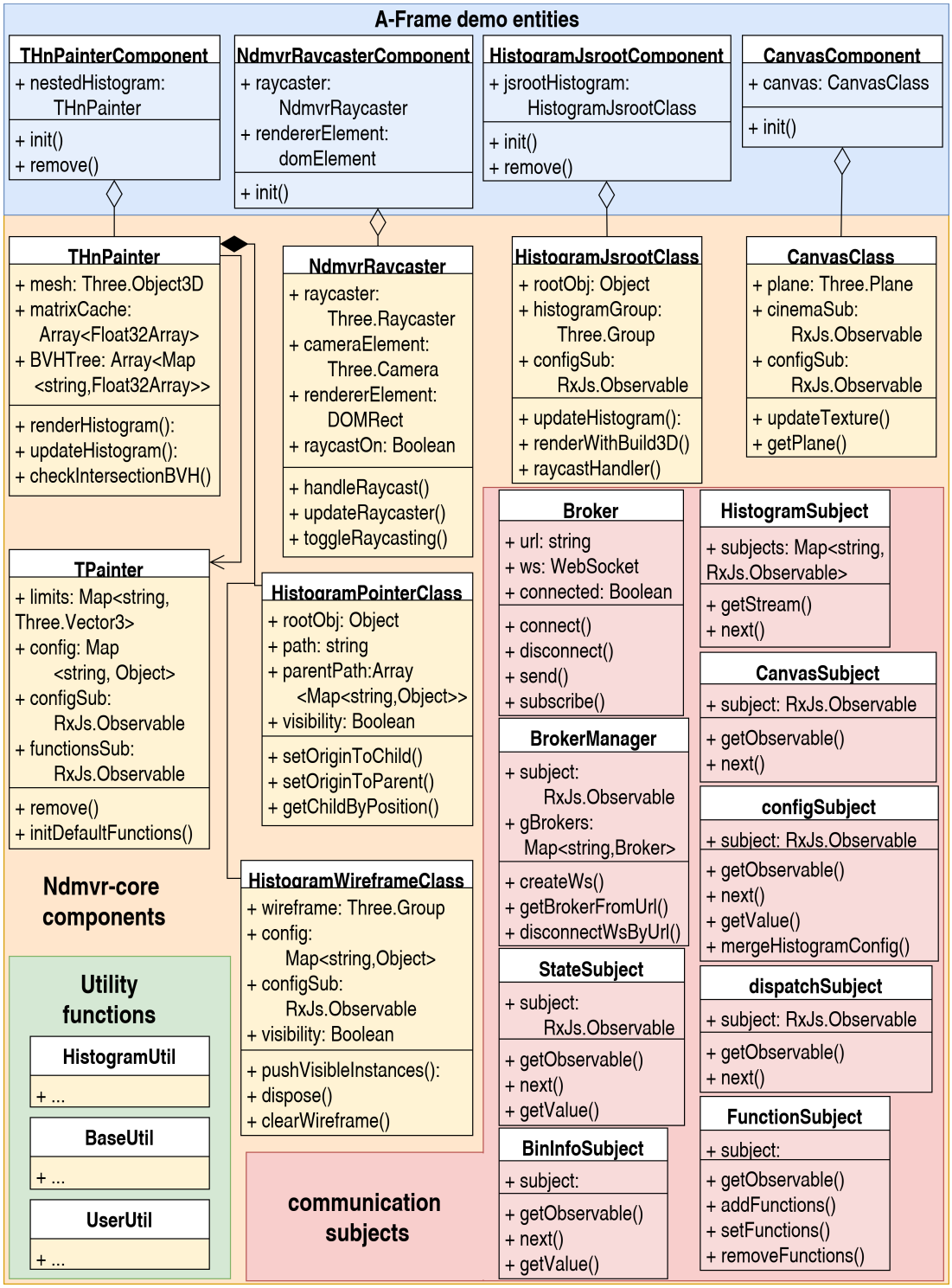
Posledným aspektom je interakcia s objektmi v scéne. Základná interakcia prebieha s objektom histogramu. Nakoľko sme sa rozhodli tento objekt inštancovať, interakcia je o to náročnejšia na správne fungovanie. Je to z dôvodu, že všetky zdanlivo oddelené objekty sú zoskupené pod jedným objektom `InstancedMesh`. V tomto prípade sme navrhli, že inštancované objekty je možné uložiť v špecifickom poradí. Pre interakciu s konkrétnym objektom je možné pristúpiť prostredníctvom identifikátora v poli, ktorého poradie sme určili.

Dôležitým faktom tu je, že tieto objekty musia byť pripravené pre podporu virtuálnej reality, preto ich základom musia byť objekty priamo umiestnené v scéne. Kvôli tomu odpadlo veľa rozličných riešení, pri ktorých sa 2D elementy s absolútnou pozíciou umiestnia na obrazovku, nakoľko tento prístup nefunguje pri integrovaní objektov do prostredia virtuálnej reality.

3.5.1 Diagram tried

Na obrázku 3.3 je priložený výsledný diagram tried, ktorý opisuje hlavné komponenty a ich vzťahy. Z dôvodu udržania kompaktnosti, diagram neobsahuje všetky metódy; zobrazené metódy taktiež neobsahujú parametre a typy návratových hodnôt. Celé signatúry sú dohľadateľné v systémovej príručke. Jednotlivé triedy a súbory sú farebne rozlíšené podľa ich logického rozčlenenia do nasledujúcich kategórií:

- **A-Frame demo entity** - Určené na testovanie (používajúce Ndmvr-core objekty)
- **Ndmvr-core komponenty** - Priame implementácie Three.js objektov.
- **RxJs subjekty** - Triedy obsahujúce RxJs subjekty určené pre komunikáciu
- **Pomocné funkcie (utils)** - Súbory podporných funkcií, ktoré slúžia na modularizáciu a skrátenie kódu komponentov.



Obrázok 3.3: Diagram tried výsledného riešenia

4 Prerekvizity

Kapitola prerekvizít sa skladá z častí, ktoré boli vypracované v rámci tejto práce, no priamo nesúvisia so stanovenou problematikou. Opísané časti tejto kapitoly teda priamo neriešia tématiku vykresľovania, ale sú nevyhnutné pre výslednú implementáciu a jej distribúciu.

V nasledujúcich podkapitolách je opísané vytvorenie dátového servera, jeho kontajnerizácia a nasadenie do prostredia Kubernetes. Druhou podkapitolou, je nastavenie CI/CD procesov, zabezpečujúce kontinuálne nasadzovanie demo aplikácie a publikáciu výsledného npm balíka.

4.1 Dátový server

Histogramy, ktoré chceme zobrazovať, je potrebné ukladať na niektoré úložisko. Pre túto potrebu, no rovnako aj pre potrebu ukladať ďalšie informácie, ako napríklad výsledky testov či prieskumov, sme sa rozhodli vytvoriť verejne dostupné REST (Representational State Transfer) programové rozhranie, pomocou ktorého vieme všetky dáta ukladať na úložisko. Toto úložisko pozostáva z MongoDB databázy[5] a NodeJS servera[6], ktorý sa pripojí na databázu a vytvára dopyty pre CRUD (Create, Read, Update, Delete) operácie. Pre ľahšie nasadenie a vývoj sme pre databázu využili oficiálny Docker[7] obraz MongoDB.

4.1.1 Implementácia servera

Na inicializáciu sme využili základnú šablónu, ktorú ponúka NodeJS cez príkaz `npm init`, ním sme taktiež nastavili všetky potrebné metadáta. Pre server bolo ďalej nutné nainštalovať závislosti.

- **MongoDB** - Vytvorenie klienta, ktorý sa pripojí na databázu.

[5] `https://www.mongodb.com/`
[6] `https://nodejs.org/en`
[7] `https://www.docker.com/`

- **Express** - Naštartovanie servera a vytvorenie endpointov, čo bude predstavovať naše rozhranie.
- **Dotenv** - Získanie premenných prostredia.

Tieto závislosti sú vo forme npm balíkov a následne sú zadefinované v súbore `package.json`. V tomto súbore je taktiež zadefinovaný vstupný bod, ktorý sme nastavili na `index.js`. Skript sa skladá z inicializácie servera, ktorý bude počúvať na požiadavky, a zo zadefinovania endpointov rozhrania. Štruktúru ďalej tvorí skript `MongoClient.js`, v ňom je asynchrónna funkcia pre vytvorenie klienta pre MongoDB databázu. Poslednou časťou servera je služba, ktorá je zadefinovaná v skripte `storageService.js`. Ten pozostáva z asynchrónnych funkcií, ktoré cez klienta MongoDB uskutočnia CRUD operácie. Jednotlivé dáta môžu byť filtrované na základe názvu kolekcie, databázy, typu a rozsahu času, kedy boli tieto dáta vložené. Tieto filtre sú v požiadavke predané vo forme parametrov v URL adrese. Pri spúšťaní tejto aplikácie je potrebné zadefinovať premenné prostredia, kde sú zadefinované parametre pre pripojenie MongoDB klienta a portu, na ktorom bude server počúvať. Tento server bol následne lokálne otestovaný. Po vyladení všetkých chýb sme prešli na nasadenie.

Na začiatku sme vytvorili definíciu pre vytvorenie Docker obrazu pomocou `Dockerfile` súboru. Ako základ bol použitý obraz `node:18-alpine`, následne je skopírovaný obsah priečinku, sú nainštalované závislosti a odhalený port, na ktorom bude server počúvať. Takto vytvorený obraz musí byť verejne dostupný, pre tento účel sme využili Gitlab register kontajnerov. V tomto registri sú verejne dostupné obrazy, ktoré je možné ďalej použiť pri nasadení. Pre automatizáciu celého procesu sme vytvorili Gitlab CI/CD pipeline. Tá je zadefinovaná vo forme `.yml` súboru. Skladá sa z dvoch etáp. V build etape sú vypísané informácie aktuálneho času, z ktorej vetvy je tento obraz vytvorený a kto je autorom posledného Git commit-u v tejto vetve. V deploy etape je vytvorený Docker obraz servera príkazom `docker build` a sú mu pridelené tagy `latest`, krátky SHA hash commitu a verzia. Tento obraz je následne prenesený do Gitlab registra, z ktorého je následne možné dostať sa k najnovšiemu obrazu tagom `latest`, no taktiež k špecifickým verziám servera.

Ďalším krokom bolo vytvorenie základných deklarácií pre Kubernetes[8]. Tie pozostávajú z deklarácie objektu Pod (najmenšia výpočtová jednotka, ktorú je možné v Kubernetes vytvoriť a spravovať) typu Deployment pre databázu a jej Service (abstraktný spôsob, ako vystaviť aplikáciu bežiacu na sade Podov ako sie-

[8] `https://kubernetes.io/`

ťovú službu.), ako aj z deklarácie podu typu Deployment pre server a jeho Service. Posledná deklarácia bola pre vytvorenie objektu Secret (objekt, ktorý umožňuje ukladať a spravovať citlivé informácie), ktorý použijú databáza a server pre nastavenie hodnôt autentifikácie. Deklarácia typu deploy sa skladá zo základných údajov, ako je počet replík a label-u (v tejto podkapitole označujeme **label** za pár kľúča a hodnoty z prostredia kubernetes [9]), s ktorým bude tento Pod pracovať. Ďalej je zadefinovaný kontajner pre MongoDB s hodnotami pre autorizáciu klienta. Deployment dopĺňa deklarácia objektu Service, ktorý prepája Pod databázy podľa label-u tak, aby bol dostupný mimo vnútornej siete klastra. Rovnako tak bola vytvorená deklarácia deploymentu pre Node.js server. V nej je taktiež špecifikácia základných údajov a vytvorenie kontajnera, ktorý je vytiahnutý z Gitlab registra. Následne mu sú nastavené premenné prostredia. Poslednou definíciou bolo vytvorenie objektu Secret, v ktorom sú zadané hodnoty pre nastavenie a následné autentifikovanie sa do databázy.

Pre ľahšie výsledné nasadenie na klaster sme následne vytvorené deklarácie dosadili do prostredia NdmSpc. Prvým krokom bolo nainštalovanie klastra cez skript. Skript sa spúšťa s prepínačom `CONTAINER_TOOL` pre nástroj na prácu s kontajnermi, kde možnosti sú Docker alebo Podman. Ďalším dôležitým prepínačom je `CLUSTER_TYPE`, ktorý definuje typ klastra a jeho možnosti sú minikube, alebo kind. Ďalšími prepínačmi je možné nastaviť verziu Kubernetesu, OLM (Operator Lifecycle Manager), prípadne niektorých z doplnkových služieb, ktoré sa skladajú z nástroja na vyvažovanie záťaže (loadbalancer), vstupnej brány (Ingress), grafického rozhrania (dashboard) či služby Keycloak[10]. Po tom, čo sa klaster spustil a nainštalovali sa všetky služby, mohli sme klaster doplniť o NdmSpc operátor prostredníctvom OLM. Operátor obsahuje všetky CRD (Custom Resource Definition), ktoré rožšíria klaster o špecifické typy zdrojov, ktoré sa budú správať, ako natívne kubernetes entity. Tým, že sme klaster rozšírili o definície špecifických zdrojov, sme mohli prejsť na naštartovanie všetkých entít prostredia NdmSpc. Celý tento proces je automatizovaný pomocou Ansible[11] skriptu, ktorý je verejne dostupný cez konfiguráciu. V nej sú zadefinované predvolené hodnoty základných údajov pre Ingress, nastavenie portov, hodnoty pre nastavenie a autorizáciu do databázy a odkaz na Docker obrazy. Následne je spustený skript, ktorý naštartuje všetky objekty z prostredia NdmSpc. Do tohto skriptu sme pridali aj deklarácie vytvoreného servera spolu s úložiskom. Pred

[9] `https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/`

[10] `https://www.keycloak.org/`

[11] `https://docs.ansible.com/`

tým sme, ale museli pôvodné kubernetes deklarácie zmeniť tak, aby sme využili šablónovanie, ktoré nám ponúka Ansible. To znamená, že statické hodnoty sme zamenili za premenné, ktoré sú načítané z predvolených, či používateľom zadaných hodnôt z konfigurácie. Posledným krokom bolo doplnenie Nginx špecifikácie o nový endpoint služby nášho servera.

4.2 CI/CD projektu

Naše riešenie sme sa rozhodli nasadiť dvomi spôsobmi. Pre používateľov, ktorí sa s naším riešením prvotne oboznamujú, alebo im postačuje bázovo nastavená aplikácia, pripravíme dostupné demo. Pre prípad integrácie nášho riešenia do existujúcej architektúry, alebo jeho úpravu pre špecifický prípad, pripravíme npm balík. Prvou časťou bolo nasadenie dema, pre tento účel sme využili službu Gitlab `Pages`. Tento krok pozostával zo zapnutia služby `Pages` priamo na Gitlabe. Druhým krokom bolo vytvorenie CI/CD pipeline-y, ktorá použitím Node.js Docker obrazu vytvorí Bundle z obsahu Git commit-u prostredníctvom príkazu `vite build`. Tento Bundle následne skopíruje do public adresára, z ktorého číta služba `Pages`.

Pre vytvorenie npm balíka boli potrebné viaceré zmeny. Prvou časťou bolo vytvorenie skriptu `index.js`, v ňom sú exportované všetky triedy a metódy, ktoré sú určené pre použitie používateľom. Ďalej sme upravili súbor `package.json`, kde sme zmenili typ na module, nastavili vstupný bod na vytvorený `index.js` a zadefinovali export na súbor `index.es.js` z priečinka `dist`, v ktorom bude vybudovaný modul. Upravili sme súbor `vite.config.js`, v ktorom je konfigurácia pre vybudovanie modulu. Táto konfigurácia pozostáva z funkcie `defineConfig`, do ktorej sme pridali parameter `mode`. Ak je tento parameter nastavený na `pages`, ako vstupné body sú nastavené html súbory jednotlivých demo aplikácií, v ktorých sú ukážky balíka. Ak má parameter inú hodnotu ako `pages`, vstupným bodom je `index.js`. Posledným krokom bolo upravenie Gitlab CI/CD pipeline.

Použitý je rovnaký node Docker obraz, pridali sme cache, do ktorého sa ukladá priečinok `node_modules`, nakoľko tento priečinok obsahuje skripty jednotlivých závislostí, nie je potrebné ho budovať opakovane. Ďalším blokom je `build-pages`, v ňom je spustený príkaz `vite build` s príznakom `mode` nastavený na `pages`, čiže vybudovaný Bundle pozostáva zo stránok ukážok balíka. Tento Bundle je v ďalšom bloku `pages`, skopírovaný do priečinka bundle pre službu `pages`, rovnako ako v predchádzajúcom kroku. Je dôležité, že blo-

ky `build-pages` a `pages` sú spustené, len ak aktivita, ktorá spustila pipeline, nepozostávala z vytvorenia tagu. Naopak posledný blok `deploy-npm`, je spustený len po vytvorení nového tagu. V tomto bloku je na začiatku upravený súbor `package.json`, v ktorom shell príkazom `sed` je upravená verzia na tú, ktorá bola definovaná v tagu. Po tejto úprave je balík skompilovaný do balíka príkazom `vite build`, bez nastavenia príznaku `mode`. V node kontajneri je npm nastavený na verejný register, aby tento balík bol neskôr verejne dostupný. Pre autentifikáciu do tohto registra je použitý token. Ten je vytiahnutý z CI/CD premenných, ktoré boli zadefinované v predchádzajúcich projektoch skupiny NdmSpc a tento projekt zdedil tieto premenné, keďže patrí taktiež do skupiny NdmSpc na Gitlabe. Skompilovaný balík je následne poslaný do npm registra príkazom `npm publish` s príznakom `access` nastaveným na public, aby bol verejne dostupný. Týmito zmenami sme docielili automatizáciu, ktorá vo výsledku po každom Git push-i do repozitára skompiluje balík. Z nového obsahu je zobrazená nová verzia na Gitlab Pages. Po vytvorení nového tagu, sa skompiluje nový balík s zodpovedajúcou verziou a je publikovaný v npm registri. Týmto spôsobom sme taktiež docielili funkcionalitu, kde vieme na hlavnej vetve repozitára mať stabilnú verziu a z vývojových vetiev dokážeme publikovať release candidates.

5 Implementácia vizualizácie

Vizualizátor sa skladá z aplikácie, ktorá ako buildovací nástroj využíva Vite[12] v kombinácii s čistým JavaScriptom. Na začiatku sme využili šablónu pre inicializáciu projektu, ktorú ponúka Vite. Prvým krokom bolo vytvorenie základnej A-Frame scény s jednou kockou v strede, ktorá bude symbolizovať histogram o veľkosti jedného binu. Na tento účel sme nainštalovali balík A-Frame pomocou npm. Následne sme `index.html` súbor doplnili o zavolanie skriptu `main.js`. V tomto skripte je do hlavného div elementu stránky pridaná základná A-Fame scéna spolu s kockou v strede. Ďalším krokom bolo pridanie služby, ktorá zabezpečí získanie histogramu pomocou http dopytu. Službu sme vytvorili v `histogramService.js`, tu sme pridali asynchrónnu metódu, ktorá vykoná dopyt metódou fetch. Tá v prípade úspešného dopytu, vráti histogram vo forme json objektu.

5.1 Komunikácia

Táto kapitola sa zaoberá implementáciou služieb a subjektov pre komunikáciu. Najprv zabezpečíme získanie histogramu pomocou http dopytu, túto službu sme vytvorili v `histogramService.js`, tu sme pridali asynchrónnu metódu, ktorá vykoná dopyt metódou fetch, Tá v prípade úspešného dopytu, vráti histogram vo forme json objektu.

V ďalšej fáze bola pridaná podpora pre získavanie histogramov cez websocket spojenie. Základom je trieda `Broker`, tá má v konštruktore parametre url, `autoConnect` a channel, týmito parametrami definujeme adresu, na ktorú sa websocket klient pokúsi nadviazať pripojenie. Parameter `autoConnect` zadáva, či má pripojenie nadviazať pri inicializácii. Parameter channel predstavuje RxJs subjekt, do ktorého sa posielajú prijaté informácie. Trieda ďalej disponuje metódami connect, tá sa pokúsi nadviazať spojenie na zadanej adrese, prijaté správy následne posiela do zadaného kanála. Metódou send dokážeme poslať informá-

[12] `https://vite.dev/`

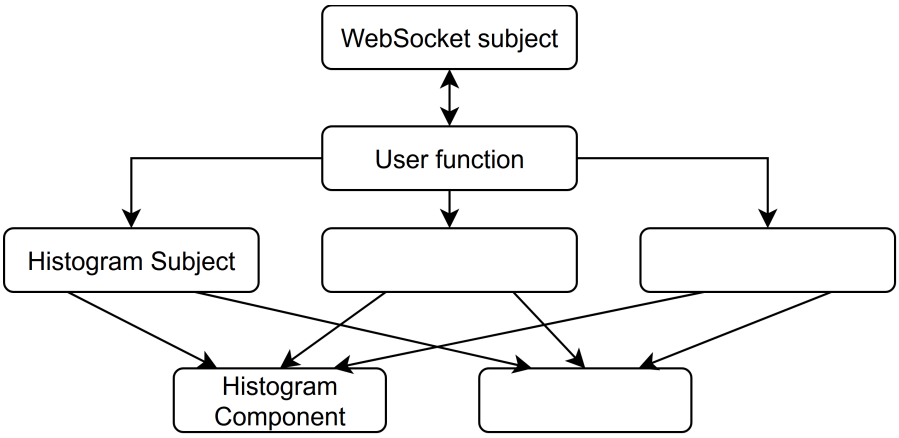
ciu naspäť na server, keďže websocket spojenie je plne duplexné.

Ďalšou triedou je `BrokerManager`. V nej je na začiatku inicializovaný RxJs `ReplaySubject` s hodnotou 1, čo znamená, že po pripojení na tento subjekt, odoberateľ vždy dostane poslednú hodnotu. Inicializovaná je taktiež mapa `gBrokers`, ktorá bude držať vstupy, kde kľúčom je adresa a hodnotou je broker spojený s touto adresou. V triede je zadefinovaná metóda `createWs`, tá vytvorí nový broker podľa vstupných parametrov. Druhou metódou v tejto triede je `createWsFromParams`. Tá berie ako parameter reťazec získaný z parametrov URL adresy. Z tohto reťazca extrahuje všetky adresy, pre ktoré následne vytvorí brokerov spojených s týmito adresami. Medzi ďalšie metódy patrí `getBrokerbyUrl`, tá vráti broker podľa URL adresy. Metóda `connectWsByUrl` vytvorí a pripojí broker podľa adresy. Metóda `getSubject` vráti RxJs subjekt, do ktorého sú posielané dáta z WebSocketu. Tento subjekt je následne možné začať odoberať a ďalej narábať s prijatými dátami.

Do rozhrania komunikácie bol taktiež pridaný nový RxJs subjekt, ten je zadefinovaný v triede `FunctionSubject`. Pri inicializácii je vytvorený Replay subjekt so žiadnou hodnotou, čo znamená, že po začatí odoberania dostane objekt všetky dáta, ktoré boli do tohto subjektu poslané. Trieda obsahuje metódy `addFunctions` a `removeFunctions`. Obe prijímajú list objektov, v ktorom je definovaná entita, identifikátor a udalosť, pre ktorú je funkcia určená. Každý takýto objekt je následne poslaný do subjektu s príznakom add, alebo delete, podľa volanej metódy. Poslednou vytvorenou triedou je `DispatchSubject`. Ten obsahuje najnižší RxJs Subject, čo znamená, že odber dostáva dáta, ktoré sú poslané až po začatí odoberania. Tento subjekt bol vytvorený pre doplnenie, či substitúciu, základných udalostí, ktoré sú vysielané napríklad pri kliku a hoveri nad objektom, sprostredkované napríklad raycaster-om. Pomocou tohto subjektu, však dokážeme poskytnúť používateľovi funkcionalitu, kde dokáže vyslať ním definovanú udalosť, na ktorýkoľvek objekt, ktorý odoberá tento subjekt.

Našu prvú verziu histogram komponentu sme prepojili s novovytvorenými subjektmi. Pri inicializácii objekt histogramu začne odoberať správy zo subjektu brokerManagera, čím vieme dynamicky dostávať histogramy. Ďalší odber správ je na functionSubject, z tohto subjektu histogram získa definované funkcie, ktoré sa majú vykonať pri zadanej udalosti. Posledný odber je vytvorený na `DispatchSubject`, ten počúva na používateľom vytvorené udalosti. Takto rozšírená entita histogramu bola otestovaná, kde v hlavnom skripte bol vytvorený objekt histogramu a cez functionSubject boli zadefinované funkcie pre klik a hover.

Výsledný tok údajov s WebSocket spojením je zobrazený na obrázku 5.1. Tok začína WebSocket Subjectom, ktorý prijíma údaje zo spojenia. Tieto údaje môžu byť dodatočne spracované používateľskou funkciou, po čom sú poslané do Histogram Subjectu, ktorý odoberá samotný komponent histogramu.



Obrázok 5.1: Graf komunikácie

5.2 Vykreslenie histogramu

Pre vykreslenie histogramu sme nadviazali na poznatky a skúšobnú implementáciu nadobudnutú počas analýzy riešenia. Pre splnenie požiadavky z návrhu o používaní bázového Three.js sme presunuli všetku funkcionalitu nenaviazanú na A-Frame do separátnej JavaScript triedy `histogram.js`. Jej úlohou je vytvoriť vizualizovaný objekt histogramu. Objekt `instancedMesh` v tejto triede je teda aj výstupom triedy, ktorý je použitý v pôvodnom A-Frame komponente vo forme zabalenia Three.js komponentu do A-Frame entity pre účely testovania.

V tejto etape sme sa opäť zamerali na vykresľovanie histogramu. Požiadavkou bolo pridanie škálovania binov na základe ich obsahu, t.j. počtu výskytov (entries) v histograme pre jednotlivé biny. Druhou požiadavkou bolo pridanie parametra size, ktorým bude možné nastaviť veľkosť histogramu v A-Frame scéne. Prvá požiadavka bola vyriešená takto. Na začiatku vykresľovania histogramu nájdeme najvyššiu hodnotu počtu vstupov, následne v cykle pri prepočte, veľkosť binu je vynásobená faktorom škálovania. Tento faktor je dopočítaný, ako podiel hodnoty počtu vstupov konkrétneho binu, voči tomu maximálnemu. Tým sme docielili, že každý bin v histograme veľkosťou zodpovedá jeho počtu vstupov, kde ak je táto hodnota 0, bin nie je zobrazený. K tejto funkcionalite bol taktiež pridaný vstupný parameter pre entitu histogramu content_min. Ním môže používateľ nastaviť minimálny počet entries pre bin, aby bol zobrazený. Druhou

časťou bolo pridanie parametra size. Ten je taktiež predaný ako parameter entity histogramu v podobe `vec3`. Hodnota je predaná do funkcie, ktorá prepočítava veľkosť a pozíciu jednotlivých binov. Táto hodnota je použitá konkrétne vo funkcii `getRootBinSizePosByAxis`. Jej úlohou je výpočet pozície a veľkosti binu na určenej osi. V pôvodnej verzii bola táto hodnota vypočítaná ako rozdiel medzi hornou a spodnou hranicou binu na osi pre veľkosť. Pozícia bola stredná hodnota medzi týmito hranicami. Túto funkciu sme doplnili o prepočet, kde ak dostane parameter size pre konkrétnu os, pozíciu vynásobi o podiel medzi cielenou veľkosťou histogramu na danej osi a absolútnou hodnotou rozdielu medzi hranicami binu. Hodnota veľkosti je vynásobená rovnakým spôsobom, čím sme docielili, že pozícia a veľkosť sú naškálované korektne podľa hodnôt histogramu a parametra veľkosti. V tejto etape sme taktiež ošetrili situáciu, keď hodnoty na osiach histogramu obsahujú záporné hodnoty. To je ošetrené takým spôsobom, že k pozícii binu je prirátaná hodnota podielu medzi rozdielom nuly a najnižšej hodnoty na osi a absolútnou hodnotou rozdielu najnižšej a najvyššej hodnoty na osi. Týmito zlepšeniami sme zabezpečili, že hodnota pozície entity histogramu vždy začína od hodnoty 0, 0, 0 na osiach histogramu, pričom taktiež môžeme zadať veľkosť histogramu v A-Frame scéne.

Ďalším problémom bolo dlhé vykresľovanie pri inicializácii, kde pri použití bázových `a-box` objektov, alebo pri aframe-instanced-mesh počiatočné vykreslenie trvalo 60 sekúnd pre počet binov 225 000, počas ktorých aplikácia nebola použiteľná. Tento problém sa použitím bázového objektu instanced-mesh z Three.js taktiež vyriešil. Aby sme sa o tom uistili, vytvorili sme tzv. stress test. Tento test pozostáva z websocket servera, ktorý pravidelne posiela histogramy aplikácii zobrazovača. Tento websocket server bol vytvorený tímom NdmSpc. Server má podobu Docker obrazu, kde po vytvorení kontajnera a zapnutí začne pravidelne posielať histogramy podľa zadanej frekvencie a veľkosti histogramu, ktoré sme kontajneru predali ako premenné prostredia príznakmi t pre frekvenciu a f pre fill. Na tento server sme v teste vytvorili websocket spojenie pridaním parametra do URL adresy aplikácie. Získané histogramy boli cez broker posielané do RxJs subjektu. Pre tento subjekt sme vytvorili v komponente histogramu odoberanie, kde sme získané histogramy vykresľovali funkciou render. Týmto testom sa nám potvrdilo tvrdenie, že problém dlhého prvotného vykresľovania bol vyriešený. Postupným zvyšovaním frekvencie sme sa dostali na hodnotu 5 ms, kedy aplikácia ešte stále zvládala vykresľovať prijímané histogramy.

5.3 Interakcia s prostredím

V tejto kapitole sa zameriame na pridanie interakcie s jednotlivými binmi, keďže túto funkcionalitu sme stratili pri použití Three.js `InstancedMesh`. Pre pridanie podpory interakcie s jednotlivými binmi sme využili Three.js Raycaster. Výhodou použitia bázového Three.js raycastera je, že z udalosti intersekcie lúča s objektom `InstancedMesh` je možné extrahovať údaj `instanceId`, čím dokážeme identifikovať konkrétny bin, ktorý lúč presekol. Hodnota `instanceId` je priradená pri vložení objektu do skupiny a keďže sme jednotlivé biny sériovo do tejto skupiny vkladali, dokázali sme spätným prepočtom získať index binu na každej z osí. Hodnotou `instanceId` je taktiež možné meniť farbu a tzv. matrix jednotlivých objektov v skupine metódami `setColorAt` a `setMatrixAt` na objekte `instancedMesh`. Implementácia Three.js raycastera do nášho balíka teda vyzerá následne. Vytvorený je A-Frame komponent `ndmvr-raycaster`. Ten pri inicializácii vytvorí objekty Raycaster z knižnice Three.js a `Vector2`, čo je dvoj-dimenzionálny vektor, ktorý bude slúžiť ako súradnica polohy myši na obrazovke. Po inicializácii je zavolaná metóda `setupRaycasting`. V tejto metóde sú vytvorené tzv. EventListenery, ktoré sú pridané na window komponent stránky a počúvajú na udalosti pohybu myši a kliku. Aby sme predišli nadmernej kontrole pretínania, pridaná je funkcionalita, v ktorej je zabezpečené, že pri pohybe myšou je kontrola pretínania vykonávaná maximálne raz, každých 100ms. Pri oboch udalostiach sa kontrola pretínania vykonáva rovnako. Najprv aktualizujeme hodnotu 2-dimenzionálneho vektora podľa polohy myši, následne aktualizujeme hodnotu pozície a rotácie raycastera. Táto hodnota je nastavená tak, že smeruje od kamery v scéne cez bod definovaný 2-dimenzionálnym vektorom podľa pozície myši na obrazovke, tým pádom z pohľadu používateľa lúč smeruje od pozície kamery k myši. Ak niektorý objekt bol pretnutý, vyberieme najbližší a ak sa jedná o objekt zo skupiny instanced-mesh, vyberieme jeho `instanceId`. Následne vytvoríme novú udalosť `instance-click` alebo `instance-hover`, podľa toho, ktorým EventListenerom bol tento stret odchytený. Tejto udalosti pridáme hodnoty `instanceId`, `instancedMesh` a funkcie `getBinContent` a `getBinPosition`. Tieto funkcie prepočítajú hodnotu tzv. contentu binu v histograme, alebo pozíciu indexu binu na jednotlivých osiach. Túto udalosť následne pošleme RxJs subjektu `DispatchSubject`, čím docielime že používateľ, alebo niektorý z komponentov dokáže túto udalosť odchytiť a ďalej s touto informáciou pracovať. Túto funkcionalitu sme otestovali vytvorením ukážky, kde sme entite histogramu poslali funkcie, v ktorých pri udalosti `instance-click` zmení veľ-

kosť binu a pri udalosti `instance-hover` zmení farbu binu. Vytvorenie tejto ukážky bolo úspešné, keďže cielený tok udalostí fungoval správne. Vo výsledku sme teda priradili EventListenery spolu s ich funkciami na komponent histogramu a pri priesečníku lúča s objektom sme histogramu posielali udalosti. Tie boli objektom histogramu správne odchytené, keďže reagoval menením farieb a veľkostí binov.

V nasledujúcej fáze vývoja bola pridaná podpora pre VR okuliare a mobilné zariadenia. Pre podporu mobilných zariadení sme prebrali funkcionalitu z našej predchádzajúcej bakalárskej práce.[4] Prvým krokom bolo pridanie komponentu `registerVrModeDetector`, ktorý deteguje na akom zariadení je aplikácia spustená. Táto informácia je poslaná do nového RxJs subjektu `StateSubject`. Pre tento subjekt bol vybraný typ `BehaviourSubject`, nakoľko vždy obsahuje práve jednu hodnotu, ktorá bude predstavovať náš stav, v tomto prípade typ zariadenia, na ktorom je aplikácia spustená. Ďalším krokom bolo pridanie elementov na obrazovku, ktoré sú opísané v našej predchádzajúcej bakalárskej práci. Posledným krokom bolo pridanie komponentu `screen-controls`, ktorý bol taktiež prebratý z predchádzajúcej práce a jeho účelom je prepojenie elementu joysticku s funkcionalitou pohybu. Podporu pre VR okuliare má A-Frame natívne. Pre pohyb a interakcie so scénou sme prebrali komponenty, ktoré boli vytvorené v aplikácii NdmVr. Tieto komponenty sme prepísali do bázového Javascriptu a pridali ich do nášho balíka. Ďalšou nutnou úpravou bolo upravenie raycastera pri používaní VR okuliarov. Na pravom ovládači je pri inicializácii vybratý objekt Raycaster z entity `ndmvr-raycaster`. Následne sú vytvorené metódy `checkHover`, ktorá je opakovane volaná v tick metóde na objekte ovládača. Druhou metódou je `triggerDownClick`, tá je spustená pri stlačení trigger tlačidla na ovládači. V oboch metódach sú kontrolované prieniky lúčov s objektmi rovnako. Vytvorený je `Vector3`, čo je 3-dimenzionálny vektor, ktorý bude slúžiť na určenie smeru lúča. Počiatočná hodnota je nastavená na x: 0, y: 0, z: -1, čo je smer dopredu od počiatočnej súradnice kamery. Tento vektor je následne otočený o hodnotu rotácie objektu ovládača, ktorý je v scéne. Ďalším krokom je získanie pozície ovládača, tú získame vo forme `Matrix4` metódou setFromMatrixPosition na objekte `Vector3`, ktorý bude obsahovať pozíciu ovládača. Teraz môžeme raycasteru upraviť pozíciu a rotáciu z extrahovaných hodnôt, čím zabezpečíme, že raycaster bude smerovať od ovládača smerom dopredu, kde je namierený. V tejto fáze sme taktiež pridali podporu vertikálneho pohybu a tzv. lietajúceho módu, ktoré sme prebrali z implementácie NdmVr.

5.4 Definícia objektu n-dimenzionálneho histogramu

Pred začatím implementácie vizualízacie n-dimenzionálneo histogramu sme museli najprv zadefinovať najvhodnejší formát, pre tento nový typ histogramu. Na začiatku sme skúšali pracovať s formátom, kde prvý histogram bude mapovací, to znamená, že jeho biny budú predstavovať ďalšie vnorené histogramy, ktoré sú ďalej všetky zadefinované v poli. Spočiatku to bol dobrý nápad na formát, no pomerne rýchlo sme narazili na problémy pri implementácii, vytváraní JSON súborov v tomto formáte a všeobecnej nejasnosti pri prechádzaní súboru, ak počet dimenzií bol väčší ako 6. Preto bol zvolený nový formát, ten mal napodobňovať stromovú štruktúru, ktorú taktiež používa trieda TTree v ROOT-e. Táto trieda vychádza zo stromovej dátovej štruktúry známej v informatike, kde samotný strom sa skladá z koreňa, ktorý predstavuje vetvu 0 v tomto prípade. Zo základnej vetvy sa strom rozvetvuje do ďalších vetiev a ich listov, čo sú konečné objekty na vetve. Takúto stromovú štruktúru sme prispôsobili pre náš prípad n-dimenzionálneho histogramu. Koreňom stromu je prvý histogram, podľa **JSROOT** špecifikácie. Rozdielom v našom formáte je pridanie parametra children do tohto histogramu. Tento parameter obsahuje ďalšie vetvy vychádzajúce z koreňa stromu. Každá vetva môže mať prívlastok content, alebo jeden zo setov. Táto vetva obsahuje pole histogramov, kde pozícia v poli určuje pozíciu binu v histogramu nad ním, podľa indexovania binov v ROOT formáte. To znamená, že ak pôvodný histogram má rozmery 2x2x2 binov, pole na vetve má 64 prvkov, kde prvý platný bin, ktorý nie je mimo rozsahu, je na indexe 22. Každý z týchto histogramov teda obsahuje hodnoty do 6 dimenzií, kde prvé až 3 dimenzie sú z prierezu rozsahu binu na koreňovom histograme a ďalšie až 3 dimenzie sú z prierezu z histogramu z children atribútu. Histogram na tejto vetve môže ďalej byť ďalšou vetvou, alebo listom v závislosti od toho, či obsahuje children atribút. Ak teda obsahuje children atribút, ďalej sa rozvetvuje, kde pole opäť predstavuje histogramy prepojené s prierezom binu na indexe pozície v poli. Takýmto spôsobom sme dokázali zadefinovať formát n-dimenzionálneho histogramu, ktorý je prívetivý pre používateľa a taktiež dobre škálovateľný pre n počet dimenzií.

```
{
  "_typename": "TH1D",
  "children": {
    "content": [
      null,
      {
        "_typename": "TH1D",
        "children": {
          "content": [
            null,
            {
              "_typename": "TH1D",
              "children": {
                "likemm": [
                  null,
                  {
                    "_typename": "TH1D",
                    "fArray": [ 132 elements…], (133 lines)
                     … (255 lines)
                  },
                   2 elements… (391 lines)
                ],
                "likepp": [ 4 elements…] (789 lines)
                "mixingpm": [ 4 elements…] (789 lines)
                "unlikepm": [ 4 elements…] (789 lines)
              },
              "fArray": [ 4 elements…] (5 lines)
               … (128 lines)
            },
             2 elements… (3299 lines)
          ]
        },
        "fArray": [ 4 elements…], (5 lines)
         … (126 lines)
      },
       4 elements… (20215 lines)
    ]
  },
  "fArray": [ 6 elements…], (7 lines)
   … (128 lines)
  }
}
```

5.5 Indexovanie

Pre potreby vizualizácie histogramu a samotného algoritmu vykresľovania si potrebujeme zadefinovať rôzne typy indexovania, ktoré budeme používať. Prvým z typov je indexovanie podľa pozície binu v histograme, podľa ROOT špecifikácie indexovania[19]. Jeho úlohou je poskytnúť používateľovi čo najlepšie čitateľnú pozíciu binu. Pre túto potrebu si zadefinujeme pole vector3 objektov, v tomto objekte každé z čísel predstavuje pozíciu binu na osi v jednom histograme. Pridelením týchto vektorov do poľa vytvárame vrstvy. Ak teda chceme určiť pozíciu niektorého binu napríklad v šiestich dimenziách, vytvoríme pole dvoch vector3 objektov, kde prvý vektor prideľuje pozíciu v prvých troch parametroch a druhý vektor prideľuje pozíciu druhej trojici parametrov. Tento typ indexovania navyše vylepšíme o vyčistený výstup, ak túto informáciu posielame používateľovi. Nakoľko, jednotlivé histogramy nemusia byť všetky troj-dimenzionálne, preto ak si používateľ vyžiada pozíciu binu v histograme pošleme mu dvoj-dimenzionálne pole objektov. Prvé pole predstavuje vrstvu histogramu, druhé pole pozícií. Konečnými vstupmi sú objekty, kde každý z nich obsahuje názov a index na osi, rozsah binu na tejto osi, chybu a hodnotu. Takto príde používateľovi dobre čitateľná informácia, v ktorej je jasné, koľko parametrov je na jednotlivých osiach a aké sú hodnoty asociované s binom.

Druhým typom je lineárne indexovanie n-dimenzionálneho histogramu, ktoré budeme používať pre algoritmus vykresľovania binov a ich uloženie v objekte `InstancedMesh`, kde samotné lineárne pole bude obsahovať instanced mesh. Tomuto poľu musíme najprv zadefinovať rozsah. Ten vypočítame tak, že iterujeme cez vetvy stromu histogramu, čím zistíme maximálny počet binov na jednej vetve pre každú vrstvu stromu. Vynásobením týchto čísel získame maximálny počet binov, ktoré môžu byť zobrazené v jednej chvíli. Toto číslo pridelíme parametru count pre instanced mesh. Nie je to teda celkový počet binov, čím ušetríme miesto v pamäti. Vychádza to z princípu, že biny zobrazujeme len z aktuálne najspodnejšej viditeľnej vrstvy, čo znamená že tento počet môže byť maximálne toľko, koľko je súčin maximalného počtu binov vetvy, na jednej vrstve. Z tohto princípu vychádza aj ďalšie špecifikum tohto indexovania. Ak teda zobrazujeme najspodnejšiu vrstvu, indexy binov pokrývajú celé pole, kde jednotlivé biny sú odsadené o jednu pozíciu. Ak však chceme zobraziť celú vrstvu n - 1, kde n je posledná vrstva stromu, pozície v poli rovnako tak pokrývajú rozsah celého poľa, no jednotlivé biny sú od seba odsadené o 1 + maximálny počet binov na vetve na vrstve n. Pre uvedenie príkladu, si predstavme že na koreni máme histogram

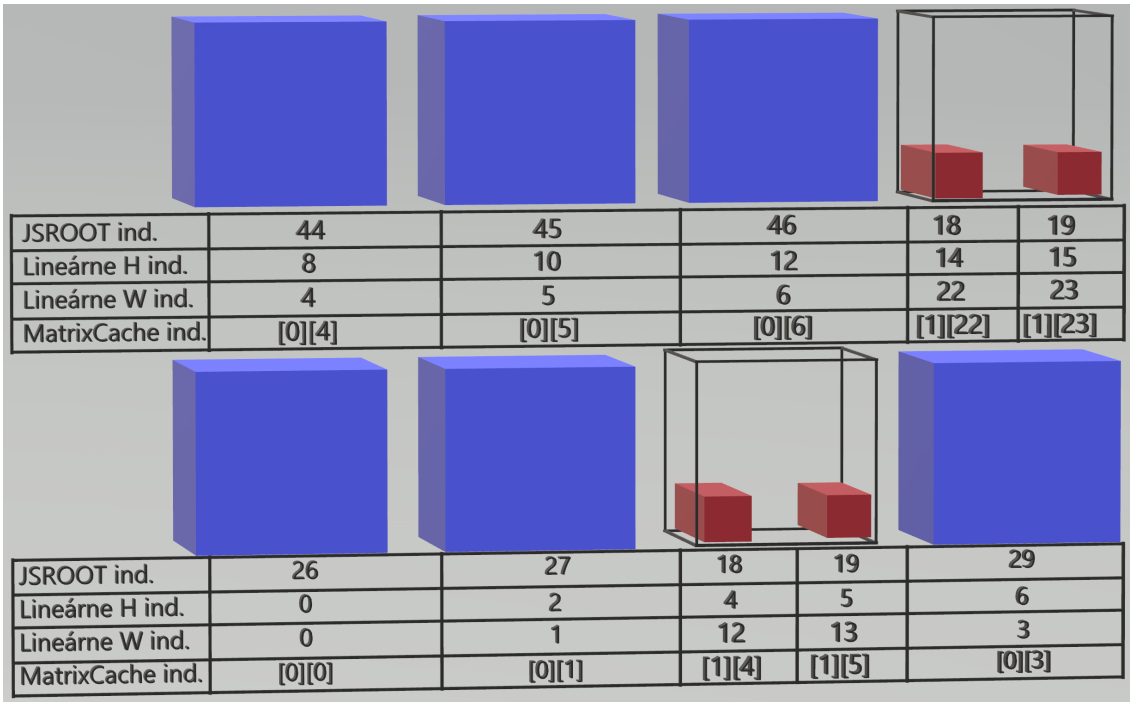
ktorý má 4 biny a vetvové histogramy vychádzajúce z koreňa majú počet binov 4 a 6. Rozsah poľa teda bude 24, keďže do súčinu berieme maximálny počet binov vetvy pre každú vrstvu. Na začiatku, ak budeme chcieť zobraziť koreňový histogram, biny v poli budú na pozíciách 0 + x * 6. Prvý bin teda bude na pozícii 0, druhý na pozícii 6 atď. Ak by sme chceli zobraziť prvú vetvu druhej vrstvy, biny by boli na pozíciách 1, 2, 3, 4, kde by pozície 5 a 6 ostali prázdne a ďalší histogram druhej úrovne by pokračoval na pozíciách 7 až 10. Pri zobrazení druhej vetvy druhej vrstvy by pozície binov teda boli súbežne od 1 do 24.

Ďalším typom je rozdelenie do binov do dvoj-dimenzionálneho poľa, prípadne troj-dimenzionálneho poľa, ak sa vetva bude rozdeľovať do dátových setov pre rôzne parametre. Tie budeme používať pre objekt matrix cache. Tento objekt sa skladá zo samostatného poľa pre každú vrstu histogramu, v prípade dátových setov do poľa pre každý dátový set. Pristupovať k jednotlivým objektom teda môžeme následovne: prvý koordinát určuje vrstvu v strome, v ktorej sa bin nachádza. Druhý koordinát označuje dátovú sadu na tejto vrstve, ak sa na nej nachádza. Posledný koordinát určuje samotnú pozíciu binu na histograme v lineárnom indexovaní. Týmto spôsobom si dokážeme zapamätať pozície a škály všetkých binov histogramu, na rozdiel od predchádzajúceho typu indexovania, čo bude dôležité neskôr pri implementácii viacerých častí.

Posledným typom je lineárne indexovanie, ktoré budeme používať pre ohraničenia binov. Toto lineárne indexovanie je špecifické tým, že pole má rozsah počtu binov v celom histograme. Vie teda uložiť ohraničenie každého binu v histograme v jeden moment. Je to z dôvodu, že biny zobrazujeme len na poslednej viditeľnej vrstve, no ohraničenia jednotlivých binov chceme ponechať aj z binov na predchádzajúcich vetvách. To zlepší prehľadnosť n-dimenzionálneho histogramu a taktiež navigovanie v ňom. Samotné indexovanie je prebraté z predchádzajúceho typu pre matrix cache, len je tzv. flattened. To znamená, že indexovanie sa začína na koreňovej vetve 0, kde jednotlivé ohraničenia sú odsadené o 1. Následne v poli pokračujú ďalšie vetvy v poradí vrstiev a vetiev, kde indexy sú odsadené o 1 + n, kde n je počet binov predchádzajúcich vetiev.

Rôzne typy indexovania je možné vidieť na obrázku 5.2. V scéne sa nachádza histogram o 3 parametroch, tie sú rozdelené tak, že na prvá vrstva obsahuje 2 parametre, ktoré sú rozdelené do osí x a z. Tretí parameter je uložený na druhej vrstve na osi x. Pre vysvetlenie, indexovanie pre **JSROOT** podporuje indexy pre tzv. "underflow" a "overflow" biny[19]. Prvý viditeľný bin sa teda nachádza na indexe 26, keďže prvých 25 binov je mimo rozsahu zvoleného binovania. Taktiež objekt `MatrixCache` obsahuje 2 koordináty, pretože histogram neobsahuje dáto-

vé sety. V prípade lineárneho indexovania H označuje indexy binov histogramu, W označuje indexy ohraničenia binov.



Obrázok 5.2: Vizualizácia typov indexovania

5.6 Algoritmus vykreslenia n-dimenzonálneho histogramu

V tejto časti sa zameriame na samotný rekurzívny algoritmus vykreslenia histogramu. Pred samotným algoritmom si predpočítame hodnoty maximálneho počtu binov pre každú vetvu stromu a taktiež maximálnu hodnotu binu pre každú vetvu. Vytvoríme základný Three.js materiál a základný mesh binu, čo je Three.Box. Výsledne zobrazeným objektom bude `InstancedMesh` z balíka Three.js, ktorému pridelíme materiál, základný mesh boxu so škálami 1, 1, 1 a maximálny počet inštancií na zobrazenie. Vytvorený objekt instanced-mesh preiterujeme, kde každej inštancii pridelíme nulovú škálu, aby boli zdanlivo neviditeľné. Pre objasnenie, nulovú škálu sme nemohli prideliť pri vytváraní instanced mesh-u, keďže každé ďalšie škálovanie závisí od toho pôvodného.

Funkcia pre vykreslenie histogramu `renderHistogram` berie 3 parametre, tu patria: štartovací index, konečný index a vrstva. Parametre indexov patria pod typ lineárneho indexovania pre objekt typu instanced mesh. Parameter vrstvy udáva konečnú vrstvu vetiev, ktoré chceme zobraziť. Na začiatku definujeme

Three.`Object3D` dummy objekt, pomocou ktorého budeme meniť pozície a škály inštancií v instanced mesh-i. Potom získame indexy vybraných setov, ktoré budú určovať, ktoré vetvy z jednotlivých vrstiev stromu vykreslíme. Na konci voláme rekurzívnu metódu render, ktorej predáme parametre štartovacieho a konečného indexu, aktuálnu vrstvu, ktorá je vždy nula, keďže algoritmus začína od koreňa stromu. Samotnú vetvu z koreňa stromu, čo predstavuje **JSROOT** objekt histogramu a objekt obsahujúci pozíciu a veľkosť pre celý histogram. Vo funkcii render najprv definujeme radix počítadlo, kde limitom čísel pridelíme počet binov na osi. Takto môžeme získať pozíciu binu v histograme počas algoritmu a taktiež bude slúžiť ako poistka vo for cykle, kde vieme, že ak celé počítadlo pretečie, niekde je chyba, keďže by sme ďalej chceli vykresliť neexistujúce biny. Toto počítadlo nastavíme na inicializačnú hodnotu podľa štartovacieho indexu. Pred cyklom si vypočítame hodnotu `stepFor`, ktorá udáva odsadenie jednotlivých binov v poli instanced mesh, aby zápis do poľa bol v súlade so zadefinovaným indexovaním. Následne získame padding binov v histograme z configu, alebo pridelíme prednastavenú hodnotu. Základ for cyklu teda vyzerá takto:

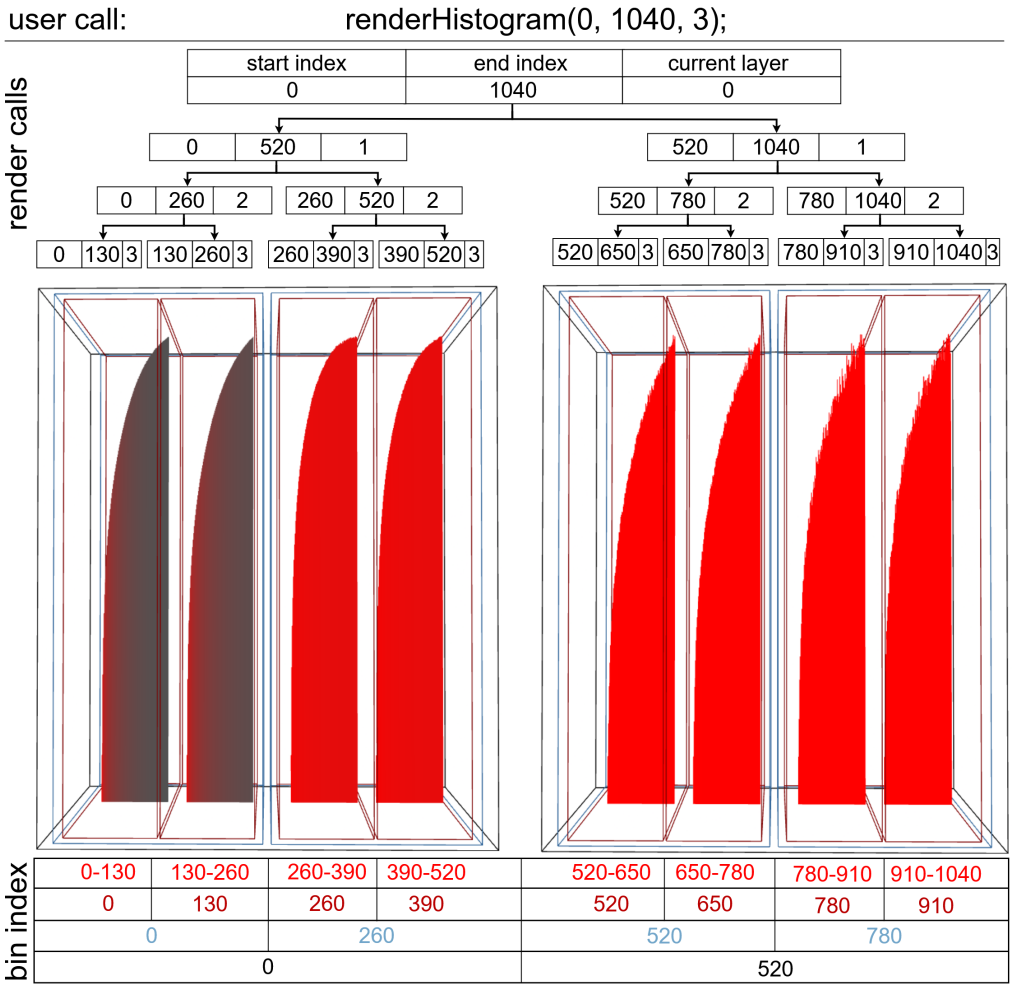
Výpis 5.1: Základ for cyklu metódy render

```
for (i = startIndex; i < endIndex; i += stepFor) {

  if (!counter.increment(0)) break;
}
```

V tele cyklu vyberieme pozíciu binu z Radix počítadla, ktorú predáme do pomocnej funkcie `computeAFrameBinSizePos` spolu s **JSROOT** objektom na aktuálnej vetve, paddingom, limitami, pozíciou pre histogram a indexom aktuálne vykresľovanej vrstvy. V tejto metóde je vypočítaná veľkosť a pozícia binu na základe jeho spodnej a hornej hranice, tieto hodnoty sú potom škálované podľa limitov celého histogramu a odsadené o padding na osi. Hodnoty sú upravené v pomocnej funkcii `rootSizePosToAFrame`, kde osi y a z sú zamenené, aby sme konvenciu súradnicovej sústavy v **JSROOT**-e pretransformovali na konvenciu Three.js. Ďalej získame hodnotu binu na aktuálnej pozícii a podľa nej farbu a výšku binu, ak sa jedná o nie 3-dimenzionálny histogram. Výsledné hodnoty zapíšeme do poľa matrix cache, ak vykresľujeme vetvu setu, tak navyše nastavíme hĺbku binu na 0.1 a pozíciu posunieme podľa indexu vetvy na vrstve stromu, aby sa jednotlivé vetvy na jednej vrstve nezobrazovali na rovnakej pozícii. Ak sa aktuálna vrstva vykresľovania rovná poslednej, ktorú chceme zobraziť, nastavíme pozíciu a veľkosť na dummy objekt. `InstancedMesh` následne aktualizujeme na indexe, na ktorom je práve index for cyklu s dummy objektom. Ak je aktuálna

vrstva menšia ako požadovaná posledná vykresľovaná z aktuálnej vetvy, podľa aktuálnej pozície a predvybraných setov vyberieme vetvu z ďalšej vrstvy. Následne rekurzívne voláme funkciu render, kde štartovací index je aktuálny index for cyklu. Konečný index ostáva rovnaký. Aktuálnu vrstvu inkrementujeme o 1, keďže sa presúvame na ďalšiu vrstvu. Funkcii predáme objekt vetvy na ďalšej vrstve. Taktiež predáme limity pre tento ďalší histogram, čo je hodnota z matrix cache ktorú sme práve zapísali, čo znamená, že pozícia a veľkosť ďalšieho histogramu bude rovnaká, ako tá z binu, na aktuálnej vetve. Týmto spôsobom sa funkcia render rekurzívne volá, až kým nie sú vykreslené všetky biny na požadovanej vrstve, pričom predchádzajúce biny sú uložené v matrix cache.

S takto definovanou funkciou `renderHistogram` je požadované vykreslenie priamočiare. Ak chceme vykresliť celý histogram na požadovanej vrstve, zavoláme `renderHistogram` s parametrami 0, súčin maximálneho počtu binov vetvy pre vrstvu a n, kde n je index vrstvy stromu, ktorú chceme zobraziť. Jednoduché je taktiež dokreslenie časti histogramu. Keďže funkcia `renderMethod` nemusí vykresliť celý histogram. Časť histogramu, ktorú chceme vykresliť jednoducho definujeme skrz štartovací, končiaci index a konečnú vrstvu, kde tieto indexy sú zašpecifikované podľa indexovania pre objekt typu instanced mesh. Rekurziu týchto volaní spolu s hodnotami parametrov a indexovaním je možné vidieť na obrázku 5.3.



Obrázok 5.3: Vizualizácia volania metódy renderHistogram

5.6.1 Algoritmus vykreslenia ohraničenia binov

Pozície a veľkosti ohraničení binov získame z objektu `MatrixCache`, kde sú tieto hodnoty dostupné pre jednotlivé biny, tieto hodnoty stačí prepoužiť, keďže ohraničenia majú kopírovať tvar jednotlivých binov. Pre technológiu zobrazenia sme sa rozhodli použiť `InstancedBufferGeometry` z balíka Three.js, keďže `InstancedMesh` nepodporuje vykresľovanie čiar. Inštancované objekty sú totiž interne reprezentované objektom `THREE.Mesh`, čo shader-u prikazuje, aby tieto objekty previedol na trojuholníky. Objekt `InstancedBufferGeometry` nám navyše ponúka lepšiu kontrolu nad inštancovanými objektmi. Pre modulárnosť kódu sme vytvorili JS triedu `HistogramWireframeClass`, v ktorej je definovaný algoritmus a akcie nad ohraničeniami. Základ pre inštancovanie tvorí geometria kocky obsahujúca len jej hrany, čiže čiary, oproti trojuholníkom pre základnú kocku. `InstancedBufferGeometry` tvoria jeho parametre. Ich úlohou je držať zvolené hodnoty pre jednotlivé inštancie, potrebujeme ich 4: pre pozíciu, veľ-

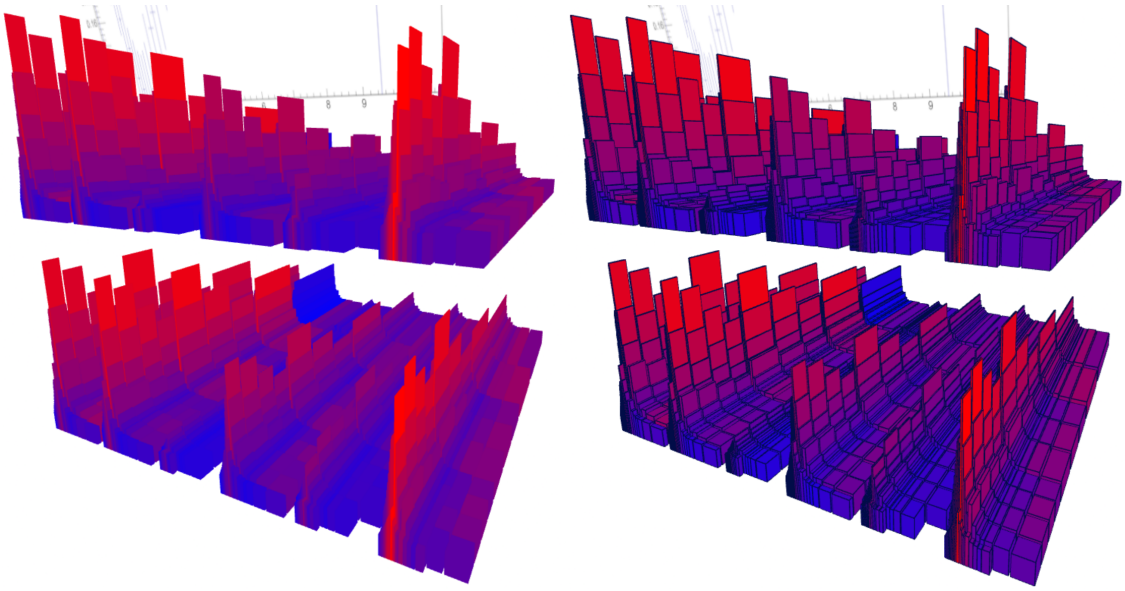
kosť, farbu inštancie a pole farieb. Paremetre sú typované polia Float32 čísel. Tie sú zabalené do objektu `InstancedBufferAttribute`, ktorému predáme pole a počet prvkov pre jednu inštanciu. Pre pozíciu a veľkosť je táto hodnota 3 (x, y, z) a pre farbu 1. Objekt `InstancedBufferGeometry` je na konci zabalený do objektu `LineSegments`, kvôli tomu, aby sa engine nesnažil inštancie previesť na polygóny trojuholníkov, ale vykreslil ich ako čiary.

Po definovaní objektov, je prvým krokom nastavenie poľa dostupných farieb z konfigurácie. V poli sú 3 hodnoty (r, g, b) pre každú farbu, ktoré sú prevedené metódou `toArray` na objekte `Three.Color`. Pre definovanie ďalších polí potrebujeme zistiť počet zobrazovaných inštancií. To z časti dokážeme iterovaním poľa `MatrixCache`, spočítaním objektov, ktoré majú príznak `rendered` nastavený na true. Tento príznak je však nastavený len na objektoch poslednej zobrazovanej vrstvy, počet teda potrebujeme doplniť o ohraničenia binov na ich vyšších vrstvách. Pole `MatrixCache` musíme teda iterovať celé spôsobom, že ak príznak `rendered` je nastavený na false, pozeráme sa taktiež na príznaky binov jeho spodných vrstiev. S počtom zobrazovaných binov následne definujeme polia pozície, veľkosti a farby. Tie sú naplené opätovnou iteráciou, polia pozície a veľkosti sú naplnené hodnotami rovnakými pre `MatrixCache` ak je bin zobrazený, no ak sa jedná o bin na vyššej vrstve musíme veľkosť ohraničenia zväčšiť o faktor 0.1, inak by sa jednotlivé ohraničenia prekrývali, čím by sa znížila prehľadnosť. Ako posledné je naplnené pole farieb jednotlivých inštancií podľa zostupnej priority farieb pre set, vrstvu a defaultnú farbu. Tieto polia opätovne prevedieme na objekty `InstancedBufferAttribute` a priradíme ich objektu `InstancedBufferGeometry`.

Posledným krokom je priradenie materiálu, pre definovanie vykreslenia 3D objektov. Pre materiál sme zvolili `ShaderMaterial` z knižnice Three.js. Ten tvorí krátky program napísaný v OpenGL Shading Language a je vykonávaný priamo na grafickej karte. V tomto programe máme dostupné atribúty, ktoré sme pridelili objektu `InstancedBufferGeometry`. Keďže nepoužívame zvyčajné matice pre pozíciu, veľkosť a rotáciu, tieto hodnoty transformujeme jednoducho tak, že defaultná pozícia je vynásobená vektorom veľkosti, čím objekt získa požadovanú veľkosť. Ďalej len k tomuto vektoru pripočítame vektor pozície inštancie. Vektor pozície následne stačí štandardne previesť z priestoru sveta to tzv. clip-space. To robíme násobením objektov `projectionMatrix`, `modelViewMatrix` a matice vektora 4, kde pôvodný `Vector3` sme rozšírili o homogénnu súradnicu. Týmto sme previedli všetky súradnice z 3D bodu do normalizovaného priestoru. Ak sú súradnice v normalizovanom svete, WebGL dokáže tieto objekty pre-

mietnuť do 2D normalizovaného priestoru obrazovky. V poslednom kroku ešte inštancii priradíme farbu, vybraním prvku z poľa dostupných farieb, podľa nastaveného indexu z poľa `instaceColorIndex`. Túto farbu následne stačí previesť do gl_FragColor, rozšírením o alpha, čím získame vektor 4 pre rgba.

Rozdiel v prehľadnosti po pridaní ohraničení binov je možné vidieť na obrázku 5.4.



Obrázok 5.4: Vizualizácia ohraničenia binov

5.7 Vykreslenie objektov cez JSROOT

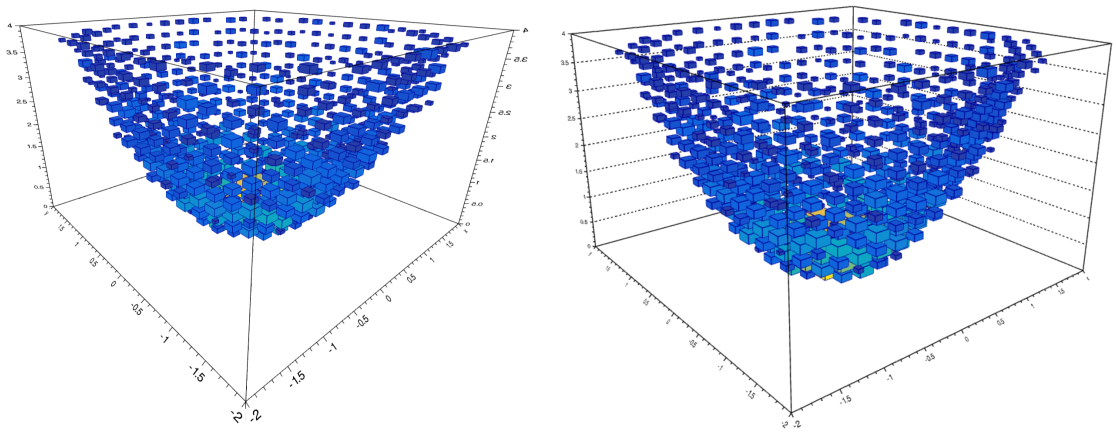
V podkapitole analýzy riešenia sme zadefinovali spôsob, akým dokážeme cez **JSROOT** vykresliť a interagovať s histogramom. Túto implementáciu sme prebrali a pre účely testovania zabalili do A-Frame entity rovnako, ako to bolo pri 3D objekte histogramu. Túto entitu sme pridali do scény. Pre overenie interakcie nám poslúžil raycaster, ktorý sme rovnako vytvorili v implementácii nášho riešenia. Udalosť priesečníku lúča raycastera sme odchytili, hodnotu indexu `instanceId` sme cez pole bins previedli na index binu histogramu. Pridaný bol handler pre tieto udalosti, v ktorom pri udalosti prechodu lúča raycastera nad prvkom binu histogramu sme mu menili farbu, podľa zisteného indexu.

Druhá implementácia vznikla v spolupráci s ROOT vývojármi po druhom ROOT PPP mítingu[13]. Prínosom bolo doplnenie funkcionality na základe vzniknutej požiadavky na repozitári **JSROOT**-u[14]. Výstupom tejto požiadavky bolo

[13] `https://indico.cern.ch/event/1592358/`
[14] `https://github.com/root-project/jsroot/issues/368`

pridanie statickej funkcie `build3d` do knižnice **JSROOT**-u. Ňou je možné vytvoriť Three.js 3D objekt na základe **JSROOT** objektu. Použitie tejto funckie značne uľahčilo a zlepšilo našu implementáciu, nakoľko už nie je treba vytvárať "dummy" div element mimo obrazovku zariadenia na ktorý sme volali funkciu `redraw` po čom, sme získali vytvorený objekt.

Výsledok porovnania vieme porovnať na obrázku 5.5. Vykreslený je rovnaký histogram, ktorý je vizualizovaný v našej demo aplikácii na ľavej strane. Na pravej strane je ukážka dema **JSROOT**[15].



Obrázok 5.5: Porovnanie vykreslenia v našom a JSROOT prostredí

5.8 Raycasting histogramu

Pre potreby používateľskej interakcie s histogramom používame bázový Raycaster z balíka Three.js. Objekt `InstancedMesh` podporuje raycasting, kde pri intersekcii sa vracia index inštancie, ktorú lúč pretol, aj napriek tomu, že všetky inštancie patria pod jeden objekt. Výhodou a taktiež dôvodom, prečo sme takto zadefinovali algoritmus pre vykreslenie je ten, že indexy inštancií v instanced meshi, sú rovnaké, ako indexy binov v celkovom histograme. Týmto spôsobom vieme jednoducho získať pozíciu binu v celom strome histogramu. S týmto indexom vieme taktiež veľmi ľahko ďalej pracovať, ak chceme napríklad zobraziť vetvu histogramu pod pozíciou na tomto bine.

Výkonnostné testy uvedené na obrázku 2.2 boli vykonávané bez možnej interakcie používateľa s histogramom, keďže počas testov bol raycasting pre celú scénu vypnutý. Po pridaní raycastera a funkcií pre interakciu sme si však všimli pomerne razantný pokles FPS, v porovnaní s pôvodným výkonostným testom.

[15] `https://jsroot.gsi.de/latest/examples.htm#th3`

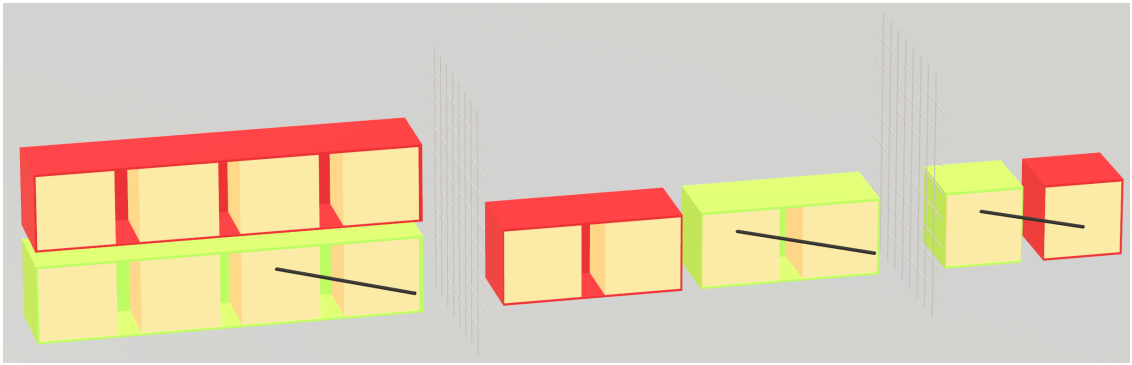
Pre porovnanie, s histogramom o veľkosti 400 000 binov, kde bolo pôvodne 110 FPS, po pridaní raycastera sa počet znížil o 40, teda na hodnotu 70 FPS. Pričom krivka s hodnotami FPS sa ďalej znižovala oproti tej pôvodnej. Obrázok 2.2 ukazuje lineárny pokles výkonu, ktorý od 100 000 binov predstavuje pokles s koeficientom približne -4.83×10^{-5}. Po pridaní interakcie spájajúcej sa s výpočtom prieniku raycastera s jednotlivými binmi sa krivka poklesu upravila. Rozdiel je možné pozorovať od 80 000 binov, kde krivka ďalej lineárne klesá s koeficientom -1.5625×10^{-4}. Vplyv interakcií na výkon je teda markantný, keďže FPS klesá približne 3,2-krát rýchlejšie.

Z tohto dôvodu sme sa rozhodli pozrieť sa do implementácie samotného raycastera a taktiež do implementácie správania objektu `InstancedMesh` pri kontrole priesečníku s lúčom. Raycaster pri kontrole rekurzívne prechádza cez objekty scény, kde kontroluje priesek s nimi. Zaujímavú vec sme zistili pri skúmaní správania kontroly intersekcie lúča s instanced mesh objektom. Toto správanie je pomerne priamočiare. Rovnako tak ako raycaster postupne kontroluje intersekciu s každým objektom v scéne, tak pre objekt `instancedMesh` je zadefinovaný for cyklus, kde postupne prechádza všetky inštancie v objekte `instancedMesh` a na konci vráti identifikátory inštancií, ktoré lúč presekol. To je všeobecne dobré riešenie a taktiež jediné správne, ak chceme kontrolovať objekty, kde nevieme či jednotlivé inštancie sú nejako systematicky uložené v scéne. Preskúmaním tohto správania sme teda zistili, prečo bol pokles výkonu pri pridaní interakcie s histogramom tak veľký. Samotná kontrola prieseku lúča s objektom je totiž pomerne náročná operácia. Operácia kontroly prieseku na objekt instancedMesh nám teda dáva časovú zložitosť $O(n)$, čo predstavuje pomerne zlý koeficient. Platí to najmä vtedy, ak ešte uvážime, že túto operáciu je potrebné vykonávať každých 100 ms, keďže takýto interval kontroly sme si zadefinovali pre pohyb myšou. Ak k tomu pripočítame náročnosť operácie, ktorá sa musí vykonať n-krát každých 100 ms, pokles výkonu je potom primeraný.

Pri skúmaní tejto problematiky sme taktiež objavili fakt, že Three.js taktiež podporuje pridanie vlastného správania pri kontrole priesečníku s objektom. Ak by sme teda zadefinovali vlastné správanie pre túto kontrolu, kde môžeme využiť fakt, že jednotlivé biny histogramu sú uložené systematicky, dokázali by sme náročnosť kontroly prieseku zmenšiť. Porobne upraviť správanie kontroly prieseku sme skúšali ešte skôr, rozdelením inštancií do BVH štruktúry, no to vieme, že nefungovalo a naopak výkon zhoršilo. Napriek tomu sme si mysleli, že problémom musí byť implementácia tohto algoritmu, keďže samotná myšlienka tohto prístupu je ideálna pre náš prípad. Preto sme sa rozhodli túto funkcionalitu pre-

implementovať.

Myšlienka spočíva v rozdelení inštancií objektu `InstancedMesh` do virtuálnej stromovej štruktúry. Kde celý histogram sme rozdelili do virtuálnych polovičných podskupín. V našom prípade teda prvé dve podskupiny obsahujú obe polovicu celého histogramu. Každá z nich teda potom obsahuje podskupiny jednej štvrtiny histogramu, až kým podskupina neobsahuje jednu inštanciu, teda jeden bin. Takýmto spôsobom by sme dokázali vykonať binárne vyhľadávanie. Týmto spôsobom by sme získali vyhľadanie preseknutej inštancie s O(log n) časovou zložitosťou, oproti pôvodnej n, čo predstavuje razantné zlepšenie výkonu. Kontrola takýchto podskupín je pomerne jednoduchá. Z podskupiny vyberieme opačné body z prvej a poslednej inštancie. Z týchto dvoch bodov následne vytvoríme tzv. Bounding box, na ktorom skontrolujeme priesečník s lúčom. Podľa návratových hodnôt zistíme, či lúč presekol inštanciu v prvej, alebo druhej polovici histogramu. Pretnutú podskupinu opäť rozdelíme na polovicu a kontrolujeme priesek. Takáto implementácia by nám zaručila binárne vyhľadávanie. Vizualizáciu navrhnutého algoritmu je možné vidieť na obrázku 5.6.



Obrázok 5.6: Vizualizácia BVH algoritmu

$$x = \langle 0, fXaxis.fNbins \rangle, \quad y = \langle 0, fYaxis.fNbins \rangle,$$

$$z_1 = \left\langle 0, \frac{fZaxis.fNbins}{2} - 1 \right\rangle, \quad z_2 = \left\langle \frac{fZaxis.fNbins}{2} - 1, fZaxis.fNbins \right\rangle.$$

Pri nepárnom počte binov na osi z, by sme ale rozdelením inštancií na polovicu získali rozmedzie:

$$x1 = \langle 0, 0 \rangle, \quad y1 = \langle 0, fYaxis.fNbins \rangle,$$

$$z_1 = \left\langle 0, \frac{fZaxis.fNbins}{2} - 1 \right\rangle$$

V prípade nepárneho počtu binov na osi z (*fZaxis.fNbins*) by sme po rozdelení na polovicu dostali skupinu, ktorá na osi x začína aj končí prvým binom, čo je v rozpore s požadovaným plným rozsahom na tejto osi. Príkladom je histogram s počtom binov na osiach x, y, z: *fXaxis.fNbins* = 4, *fYaxis.fNbins* = 4 a *fZaxis.fNbins* = 5. Celkový počet binov je 80, pričom predel skupín prvého rozdelenia vzniká medzi indexmi 39 a 40. Pozícia binu na indexe 39 je na osi x rovná 0, zatiaľ čo požadovaná pozícia je *fXaxis.fNbins - 1*. Vytvorená BVH skupina by tak mala šírku jedného binu namiesto plnej šírky histogramu.

Preto sme museli tento algoritmus upraviť. Po zvážení iných možných prístupov a možností sme sa rozhodli algoritmus upraviť takto. Princíp binárneho vyhľadávania ostane rovnaký, rozdielom bude rozdeľovanie do podskupín. V tejto verzii pozíciu preseknutého binu budeme hľadať prostredníctvom troch separátnych krokov, každý pre jednu os. Pri hľadaní budú teda 3 cykly pre 3 osi. V každom cykle sa jedna os bude rozdeľovať polovicou na podskupiny. Ak teda počas prvého cyklu budeme hľadať pozíciu preseknutej inštancie na osi z, jednotlivé podskupiny budú vždy obsahovať plný rozsah na osiach x a y. Takto zabezpečíme, že rozdelením do podskupín, vždy skontrolujeme celý rozsah podskupiny. Algoritmom v tejto podobe by sme však dokázali nájsť inštanciu na jednej vetvi stromu, čiže maximálne v jednom maximálne 3-dimenzionálnom histograme. Doplniť ho však vieme o rekurziu. Práve tu je jeden z dôvodov, prečo sme sa rozhodli si zapamätať pozíciu a škálu všetkých binov histogramu v poli matrix cache. Ak v koreňovej vetve nájdeme preseknutý bin a na tejto pozícii sa nachádza namiesto binu vetva s ďalším histogramom, tak si jednoducho zapamätáme pozíciu preseknutého binu a algoritmus začína odznova na nájdenej vetve ďalšej vrstvy. Takto sa algoritmus opakuje, pokiaľ nedôjde do poslednej zobrazenej vrstvy stromu pre danú vetvu. Týmto prístupom je algoritmus trochu náročnejší, kde počet iterácií pre nájdenie preseknutého prvku predstavuje $k \cdot 3 \cdot \log_2(n)$, kde n je počet binov histogramu na vetve a k je počet parametrov pre hľadaný dátový bod. V širšom meradle, ak sa pozeráme na asymptotický rast funkcie, je časová zložitosť porovnateľne rovnaká s predchádzajúcim algoritmom.[20] Keďže k (počet parametrov pre dátový bod) je dostatočne nízke a nezávislé od vstupu n, teda nerastie úmerne s počtom binov. To znamená, že ju možno zaradiť ako multiplikatívnu konštantu, čo zanecháva logaritmický nárast zložitosti[21, 22]. S takto upraveným algoritmom sme mohli prejsť na jeho implementáciu.

Základom je vytvorená funkcia `checkIntersection`, ktorá berie parameter lúča, oproti ktorému budeme kontrolovať priesečník. Na začiatku vytvoríme Three.`Vector3` objekt, do ktorého bude zapísaná pozícia prieseku lúča s ob-

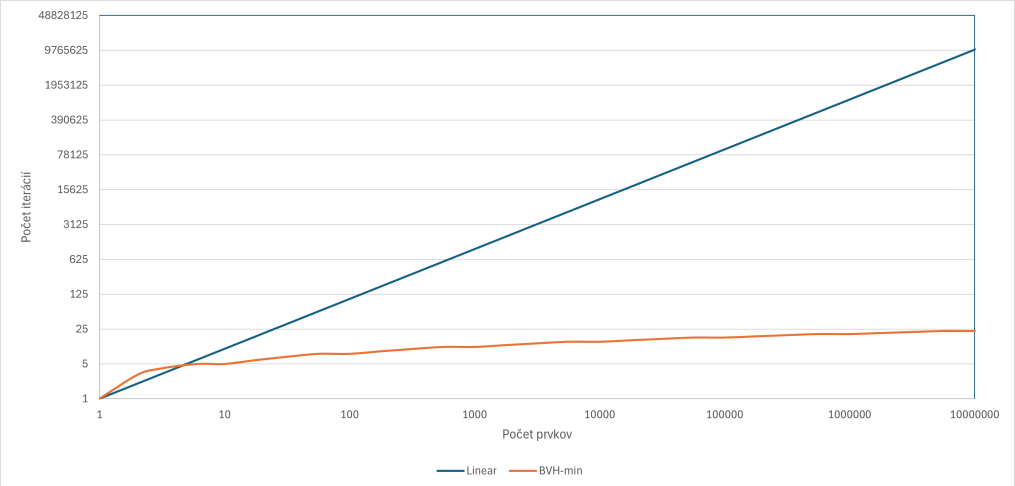
jektom. Ďalej definujeme pomocnú metódu `createBox3`, ktorá berie parametre vrstvy, indexu a setu. Táto metóda vytvorí základný `Box3` z objektu v matrix cache, čiže binu histogramu na definovanej vrstve, indexe a setu. Ďalšou pomocnou funkciou je `checkAxis`. Prvým parametrom je step, ten predstavuje krok medzi inštanciami na osi, keďže v poli matrix cache sú zoradené lineárne. Štartovací a končiaci index predstavujú indexy prvého a posledného binu podskupiny. Offset, čiže odsadenie, je opäť kvôli lineárnosti uloženia inštancií, kde toto odsadenie označuje hodnotu posunu v poli, podľa posunu na osi v predchádzajúcom cykle na inej osi. Posledný parameter set, označuje vybraný set, na ktorom sa bude kontrolovať priesek. V tele funkcie si z štartovacieho a konečného indexu vypočítame index half na polovici medzi nimi. Definujeme 4 boxy pre prvý index, half - 1, half a posledný index. Tieto boxy operáciou union z objektu Three.`Box3` spojíme do 2 boxov, ktoré budú slúžiť ako boundary boxy dvoch podskupín. Na tieto boxy aplikujeme pozíciu celého histogramu, pre prípad, ak by bol celý histogram posunutý v scéne. Následne na objekte lúča kontrolujeme priesek oproti týmto boundary boxom. Ak došlo k prieseku, vytvorí sa objekt, do ktorého sa zapíšu indexy boxu, z ktorého bol boundary box vytvorený, target a vzdialenosť lúča od jeho začiatku po target. Funkcia vracia pole týchto objektov. Treťou pomocnou funkciou je `dfs`. Tá berie parametre step, start, end, offset, layer a set, kde tieto parametre majú rovnaký význam ako pri funkcii `checkAxis`. V tele funkcii je najprv zadefinované prázdne pole output, doň sa budú zapisovať pozície na osi, kde bol bin preseknutý. Zavolaná je vnútorná funkcia `traverse`, ktorá ako parameter berie objekt, ktorý vracia funkcia `checkAxis`. Pre prvú iteráciu jej pridelíme objekt, kde prvý a druhý index sú indexy štart a stop z parametrov funkcie `dfs` a hodnoty target a distance nastavíme na null. Následne je opakovane volaná funkcia `checkAxis`, ktorá nám povie, s ktorou podskupinou došlo k prieseku. Tieto podskupiny sú po každej iterácii zmenšované o polovicu. Koniec nastáva, ak štartovací a konečný index sa rovnajú, čo znamená, že sme získali pozíciu binu na osi, na ktorej došlo k prieseku. Tento index sa zapíše do poľa output, ktoré funkcia na konci vracia. Výsledkom tejto funkcie je teda pole všetkých indexov na osi, na ktorej došlo k prieseku.

Algoritmus sa začína funkciou `recursiveSearch`, ktorej predávame parameter koreňa stromu v podobe **JSROOT** histogramu a hodnoty 0, čo predstavuje nultú vrstvu. Funkcia ďalej berie parametre offset a set, ktoré majú rovnaký význam ako pri predchádzajúcich funkciách a parameter path, cez ktorý, ak algoritmus prejde do ďalšej vetvy stromu, teda ďalšieho histogramu, zapamätáme si pozíciu z predchádzajúcej vetvy. Parametre offset, set a path pri prvom vola-

ní funkcie nenastavujeme, keďže tie sú nastavené neskôr pri rekurzívnom volaní. V tele funkcie si najprv získame hodnoty `fX`, `fY` a `fZ`, čo predstavuje počet binov na jednotlivých osiach. Definované sú `stepZ`, `stepY` a `stepX`, ktoré budeme predávať ako parameter step a predstavujú odsadenie jednotlivých binov na jednej osi. Definované je tiež pole result, do ktorého budú zapísané informácie o všetkých preseknutých binoch. Následne začíname prvým volaním funckie `dfs`, ktorej predáme hodnoty `stepZ`, 0 a `fZ - 1`, keďže chceme skontrolovať celú os, offset, layer a set. Pre každú valídnu pozíciu z si potom vypočítame `offsetZ`, čiže odsadenie v lineárnom indexe, podľa pozície z. Potom opätovne voláme funkciu `dfs` s parametrami `stepY`, 0, `fY - 1`, `offsetZ`, layer a set. Pre každú valídnu pozíciu na osi y vypočítame `offsetY`, čo je opäť odsadenie v lineárnom indexe podľa valídneho z a y. Potom voláme funkciu `dfs` s parametrami `stepX`, 0, `fX - 1`, `offsetY`, layer a set. Pre každé valídne x teda získavame pozíciu binu, ktorý lúč presekol. Definujeme hodnotu `fullPath` spojením parametra path a získanej pozície na x, y a z. Získame si z objektu **JSROOT** histogramu funkciou `getBin` lineárny bin index podľa indexovania špecifikovaného ROOTom. Priebeh sa tu rozvetvuje, ak aktuálna vetva nemá parameter children, vieme že sme na poslednej vrstve stromu, takže do poľa result zapíšeme výsledok o priesečníku. Tento objekt obsahuje informácie o pozícii v celom strome, čiže n-dimenzionálnom histograme, presný bod a vzdialenosť k tomuto prieseku, identifikátor inštancie, pole ROOT indexov, kde každý index predstavuje ROOT index na danej vetve stromu, range pozostáva z poľa rozsahov binu na jednotlivých osiach, jsrootObj, ktorému je pridelený **JSROOT** objekt vetvi, na ktorom sa algoritmus skončil a hodnoty content a error pre preseknutý bin. Ak histogram má children, tento parameter prechádzame javascript funkciou `flatMap`. V každej iterácii dostane callback funkcie hodnotu kľúča setu, alebo reťazec content, ak ďalšia vrsta neobsahuje sety a pole histogramov na ďalšej vrstve, ktoré vychádzajú z tejto vetvy a sú ďalšie vetvy, alebo listy na strome. V callbacku flatmap získame histogram z poľa children podľa skôr vypočítaného lineárneho indexu prieseku binu. Vypočítame si hodnotu `childOffset`, ktorá predstavuje index prvého binu v ďalšom histograme, podľa špecifikácie indexovania pre matrix cache. Z poľa matrix cache si následne vytiahneme prvok na tomto indexe, kde ak je tento prvok nulový, vieme že sme došli do bodu, kde nájdený preseknutý bin nie je na poslednej vrstve, ale na poslednej vykreslenej vrstve + 1, čo znamená, že v predchádzajúcej iterácii bol posledný vykreslený bin, čiže vraciame prázdne pole. Ak tento prvok nie je nulový, algoritmus opäť pokračuje ďalšou iteráciou `recursiveSearch`, ktorej pridelíme parametre **JSROOT** objektu, čiže vetvy na ďalšej vetve stromu, pre vrstvu pri-

delíme hodnotu aktuálnej vrstvy + 1, `childOffset`, `fullPath`, a hodnota setX ktorá predstavuje set, ktorý sa bude kontrolovať. Výsledná hodnota funkcie flatmap na konci obsahuje pole priesekov na histograme pre ďalšiu vrstvu stromu a jednotlivé sety. Ak toto pole obsahuje nejaké prvky, sú pridané do výsledného poľa result. Ak pole neobsahuje žiadne prvky, je vypočítaná hodnota indexu binu v špecifikácii pre matrix cache. Ak tento prvok z matrix cache má nastavený príznak `rendered` na true, znamená to, že sme sa v algoritme dostali do bodu, kedy nie sme na poslednej vrstve histogramu, ale na poslednej vykreslenej vrstve, vraciame teda z funkcie `recursiveSearch` rovnaký objekt, ako z bodu, keď sme sa dostali do poslednej vrstvy stromu. Výsledné pole result na konci roztriedime podľa hodnoty distance, čo nám zaručí, že prvý prvok v tomto poli bude najbližší k používateľovi. Celkový výsledok prieseku lúča s histogramom je návratová hodnota prvého volania `recursiveSearch`.

Vo výsledku, z pôvodného správania s $O(n)$ časovou zložitosťou, sme prispôsobeným algoritmom pre kontrolu priesečníkov získali algoritmus s $O(\log_2 n)$ časovou zložitosťou. Pre predstavu tohto zlepšenia môžeme povedať, že v histogramoch, ktoré majú počet binov väčší ako 50 000, sme dokázali získať zrýchlenie v desaťtisícoch oproti pôvodnému algoritmu. Na obrázku 5.7 je graficky znázornený rozdiel počtu iterácii medzi základným lineárnym hľadaním a BVH binárným vyhľadávaním. Hodnoty grafu vychádzajú z analytického výpočtu na základe logiky algoritmu a výslednej štruktúry BVH stromu, nejedná sa teda o namerané hodnoty.



Obrázok 5.7: Porovnanie alrogitmov intersekcie

5.9 Optimalizácia algoritmu výpočtu

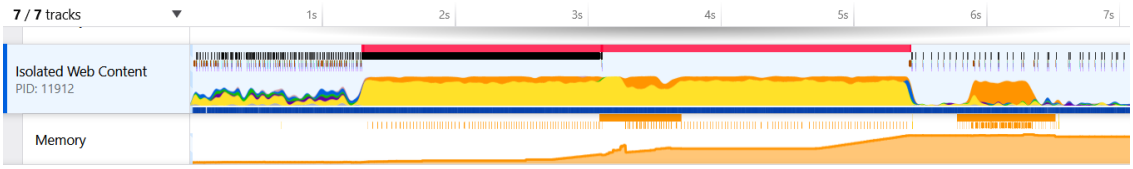
Pre zlepšenie výkonu vykresľovania bol ako najväčší problém identifikovaný fakt, že doposiaľ sme nerozlišovali indexovanie medzi poľom `MatrixCache` a poľom v objekte `InstancedMesh`. Nakoľko to nevytváralo rozdiel pri umelo naplnených histogramoch, pri reálnych histogramoch bol tento problém markantný. Rozdiel je vo fakte, že v reálnom histograme je veľa binov nezaplnených, čo znamená, že pole `InstancedMesh`-u bolo oveľa väčšie, než muselo byť. Podľa pipeline špecifikácie[23], platnej pre OpenGL to znamená, že všetky tieto inštancie s nulovou veľkosťou boli zachytené až v kroku rasterizácie vykresľovania, kde sa proces zastavil, keďže objekt zaberal 0 pixelov. Tým sme sa vyhli fáze fragment shader-a, no nevyhli sme sa najťažšej časti prepočtu a to prevedeniu bodov z globálneho sveta do tzv. 2D „view" sveta. Preto aj keď z binov bola viditeľná len ich časť, výkon sa správal rovnako, ako keby boli zobrazované všetky biny histogramu. Taktiež do tohto poľa patrili inštancie binov ktoré boli zaplnené, no patrili do vrstvy histogramu ktorý bol na inej, ako zobrazovanej vrstve. Preto jediným riešením je nahradenie poľa v objekte `InstancedMesh` za druhý, naplnený len o biny, ktoré sú viditeľné. Týmto spôsobom sa medzi inštancie dostanú len tie biny, ktoré majú byť zobrazené. Čím získame dôležitú vlastnosť, že už naďalej nezáleží na počte binov histogramu, no len na počte aktuálne viditeľných binov histogramu.

Dôležitou informáciou však je, že takto skomprimovať pole inštancií je možné len kvôli tomu, že sme zadefinovali vlastné správanie raycastera, ktorý kontroluje prieniky podľa hodnôt zapamätaných v poli `MatrixCache`. To je z dôvodu, že po skomprimovaní tohto poľa by sme bázovým Three.js raycasterom nedokázali naspäť dekódovať správny index binu podľa indexu v poli inštancií. Týmto zlepšením sme dosiahli najlepší možný výkon z pohľadu vykresľovania histogramov po inicializačnom prepočte, nakoľko všetky zobrazované objekty sú inštancované cez jeden objekt

Implementáciu sme taktiež optimalizovali pre pamäť využitím `InstancedBufferGeometry`. Toto zlepšenie vychádza zo zadefinovaného princípu dizajnu a to, že histogram je staticky položený objekt v scéne, kde používateľ sa pohybuje okolo histogramu, nie naopak. To znamená, že histogram nebude nikdy rotovaný, čiže hodnoty rotácie sú vždy nulové, histogram je teda neustále súbežný s globálnymi osami sveta. Na základe tohto princípu sme previedli objekt `InstancedMesh` na objekt `InstancedBufferGeometry` z knižnice THREE.js. To z dôvodu, že pri tomto objekte má používateľ kontrolu nad paramet-

rami, ktoré sú predávané shaderu cez používateľom definovanú funkciu. Z tohto dôvodu, môžeme previesť štandardne používané pole `Matrix4` objektov obsahujúce informáciu o pozícii, veľkosti a rotácii pre každý objekt. To sme zamenili za 2 typované polia Float32 o veľkosti n * 3, čiže počet inštancií vynásobený o 3 (počet osí sveta). To znamená, že každá z inštancií zaberá len 6 Floatov namiesto 16, čím získavame 267 % zefektívnenie pamäte vykresľovaného objektu.

Testovaním histogramu o veľkosti 3 parametrov a počte 500 000 binov sa taktiež zistilo, že implementovaný algoritmus je pomalý. To bolo najviac zrejmé pri interakciách, kde pri každom kliku sa pomocné objekty histogramu museli prepočítať. To zapríčinilo, že pri každej interakcii sa celá aplikácia na približne 4 sekundy zastavila. To by bolo obzvlášť nepríjemné pre zážitok vo VR prilbe, kde by sa celý premietaný obraz zastavil na niekoľko sekúnd. Na celý prepočet sme sa pozreli prostredníctvom profilácie ponúkanej prehliadačom Mozilla Firefox. Merania v tejto kapitole boli vykonané na zariadení Dell disponujúcim operačným systémom Windows 11, s 8 GB operačnej pamäte, procesorom Intel Core i5-6300U a integrovanou grafickou kartou Intel HD Graphics 520.



Obrázok 5.8: Graf profilácie výkonu 1

Na grafe profilu zobrazenom na obrázku 5.8 je vidieť vyťaženie procesora na približne 4 sekundy, kedy bol výpočet vykonávaný. Okrem samotného výpočtu je vidieť taktiež vysokú mieru garbage collector-a (GC), ktorý tvoril 18-19 % celkového času. Čo znamená, že pomocné objekty pri výpočtoch nie sú používané efektívne. Čas tohto prepočtu zaberajú 2 kroky, prvý je prepočet pomocných objektov histogramu, druhým je vytvorenie BVH stromu pre raycasting. Pri spätnom stopovaní volaní a ich časov sme odhalili, že veľkú mieru prostriedkov zaberajú akcie, ktoré sa na prvý pohľad nezdajú až tak náročné, ako napríklad inicializovanie vektora 3 z knižnice Three.js. Ďalším cieľom teraz bolo ďalšie odhaľovanie neefektívnych metód a objektov. Prvým krokom bolo nahradenie pomalých Javascriptových metód poľa pre iteráciu ako napríklad map, či forEach. Tie sme nahradili natívnymi for cyklami. Ďalej sme upravili používanie vytváraných objektov tak, že pomocné dočasné objekty sú vytvorené raz na začiatku algoritmu a následne sú prenášané cez argumenty funkcií. Pri výpočtoch teda používame jeden objekt, ktorý opätovne prepisujeme, tým sa zmenšila miera gar-

bage collector-a a taktiež času potrebného pre opakujúcu sa inicializáciu. Takto upravené boli operácie transformácie osí z **JSROOT** prostredia do Three.js koordinátov, transformácie matice lokálneho sveta do globálneho súradnicového systému a samotný prepočet veľkosti a pozície binu. Prepoužitý bol taktiež objekt `Color` z Three.js pri prepočte gradientu farby podľa hodnoty vstupov binu.

Matematické metódy nad vektormi boli taktiež identifikované za neefektívne. Kvôli tomu boli všetky objekty vektorov z knižnice THREE.js nahradené za Float32 typované polia[24] o troch prvkoch, nad ktorými vykonávame čisté matematické operácie namiesto operácií ponúkaných objektom `Vector3`. Transformované boli taktiež ďalšie metódy z knižnice Three.js. Príkladom takejto transformácie, je aplikovanie matice sveta na vytorenom boxe pre BVH strom. V pôvodnom riešení (v ľavej časti obrázka 5.9) je metódou `decompose` extrahovaná pozícia, rotácia a veľkosť, ktoré sú okrem rotácie aplikované na pozíciu a veľkosť kocky. V optimalizovanom riešení (v pravej časti obrázka 5.9) pracujeme s poľom `Float32Array` o veľkosti 6 pre 3 hodnoty pozície a veľkosti, k matici pritupujeme priamo cez jej hodnoty, namiesto extrahovaním metódou `decompose`. Hodnoty boxu násobíme získanými hodnotami, čím dostaneme rovnaký numerický výstup s tým pôvodným.

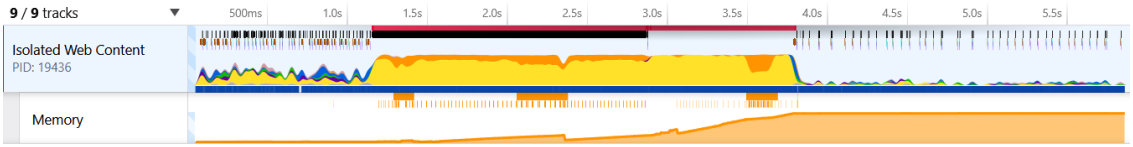
```
function applyWorldMatrix (box, matrixWorld) : {position: Vector3, scale: Vector3} {
  // Decompose matrixWorld into position and scale
  const worldPos : Vector3 = new THREE.Vector3();
  const worldScale : Vector3 = new THREE.Vector3();
  const worldQuat : Quaternion = new THREE.Quaternion();

  matrixWorld.decompose(worldPos, worldQuat, worldScale);

  // Transform position
  const newPos : Vector3 = new THREE.Vector3()
    .copy(box.position)
    .multiply(worldScale)
    .add(worldPos);

  // Transform scale
  const absWorldScale : Vector3 = new THREE.Vector3(
    Math.abs(worldScale.x),
    Math.abs(worldScale.y),
    Math.abs(worldScale.z)
  );

  const newScale : Vector3 = new THREE.Vector3()
    .copy(box.scale)
    .multiply(absWorldScale);

  return { position: newPos, scale: newScale };
}
```

```
function applyWorldMatrix(box, matrixWorld) : void { Show usages  Daniel Chovanec
  const e = matrixWorld.elements;

  // Extract scale from matrixWorld
  const sx : number = Math.sqrt( x: e[0] * e[0] + e[1] * e[1] + e[2] * e[2]);
  const sy : number = Math.sqrt( x: e[4] * e[4] + e[5] * e[5] + e[6] * e[6]);
  const sz : number = Math.sqrt( x: e[8] * e[8] + e[9] * e[9] + e[10] * e[10]);

  // Set position
  box[0] = box[0] * sx + e[12];
  box[1] = box[1] * sy + e[13];
  box[2] = box[2] * sz + e[14];

  // Set scale
  box[3] = box[3] * sx;
  box[4] = box[4] * sy;
  box[5] = box[5] * sz;
}
```

Obrázok 5.9: Transformácia funkcie applyWorldMatrix

Implemetovanými zlepšeniami sme dosiahli razantné zlepšenie prepočtu, kde z pôvodých 4115ms, sme dobu skrátili na 2636ms.

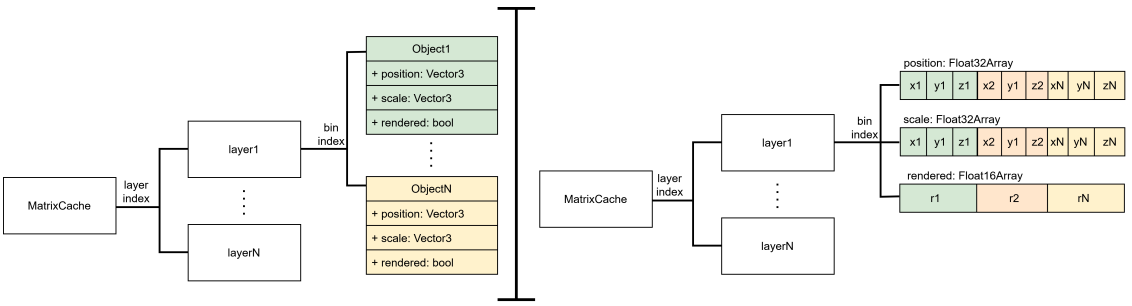
Miera garbage collector-a aj keď klesla z celkových 16 % na 8.4 %, bola však stále vysoká. To je najviac vidno na konci grafu (obrázok 5.10), kde v časti do 3. sekundy je vidieť vysokú mieru garbage collector-a, taktiež na chvoste grafu je vidieť enormnú záťaž časti major GC, v ktorej sa musia vyčistiť všetky nazbierané



Obrázok 5.10: Graf profilácie výkonu 2

objekty, ktoré neboli vyšistené postupne počas minor GC. V profilácii bolo taktiež do značnej miery vidieť operácie nad poľami. Preto sme algoritmus opätovne prepísali.

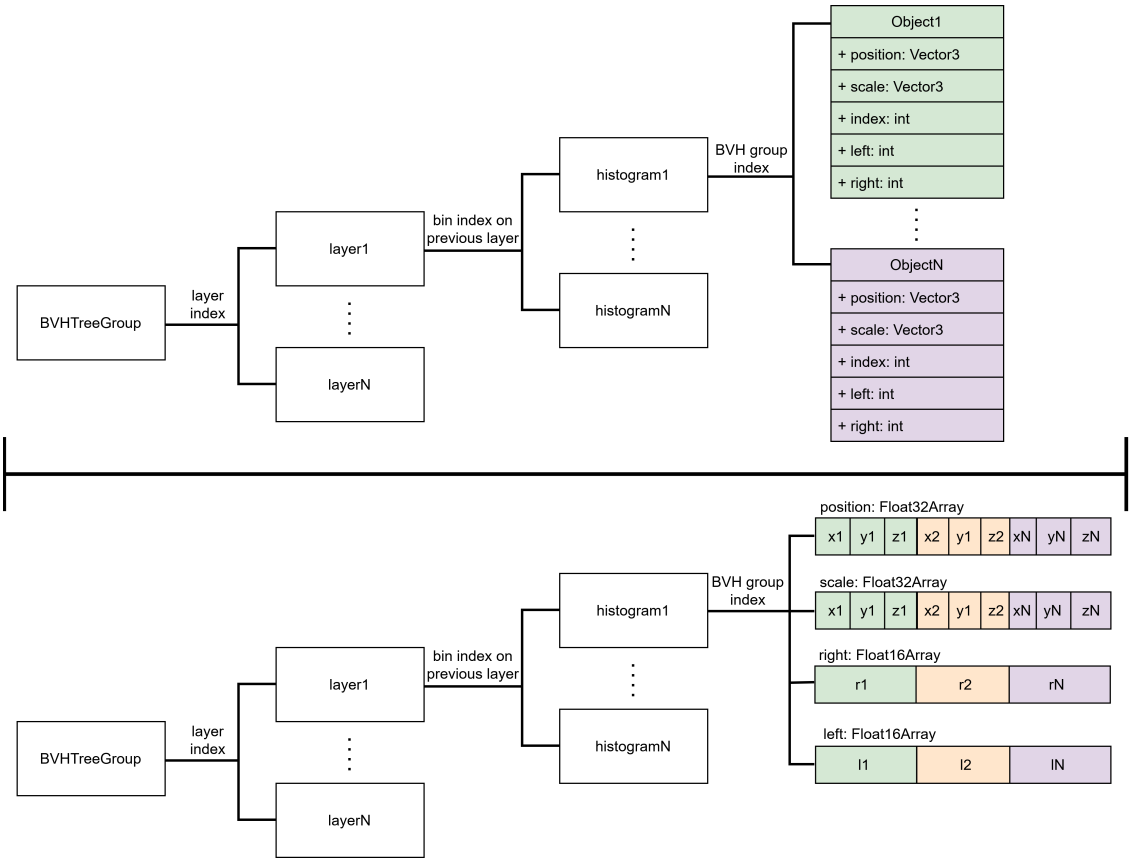
JavaScript natívne polia sme nahradili za Float32 typované polia. Týmto spôsobom sa dokážeme vyhnúť neefektívnym metódam ako napr. split, či samotnému zapisovaniu a čitaniu objektov z JavaScript poľa. To bolo pomerne jednoduché pre objekt ohraničenia binov, nakoľko v ňom už boli používané typované polia. Osobitým problémom bola transformácia používaných natívnych Javascript polí, ktoré sme používali ako pomocné objekty pri prepočte histogramu. Pôvodne sme tieto polia napĺňali pomocou iných natívnych polí a následne ich transformovali do typovaných pre objekt `InstancedBufferGeometry`. Teraz sme však pri výpočtoch všetky tieto polia nahradili typom `Float32Array`. Prvým je `matrixCache`, pôvodný objekt tvorili polia objektov pre vrstvy jednotlivých inštancií zložených z hodnôt pozície, veľkosti a príznaku `rendered`. To je zamenené trojicou `Float32Array` pre každú z vrstiev histogramu. Táto zmena umožňuje priamo napĺňať typované polia používané objektom `InstancedBufferGeometry`, bez potreby dodatočnej transformácie dát. K hodnotám jednotlivých inštancií je následne možné pristupovať prostredníctvom indexu konkrétneho binu a jeho vrstvy. Štrukturálny rozdiel je možné vidieť na obrázku 5.11, kde ľavý diagram reprezentuje pôvodnú štruktúru objektu, na druhom je transformovaný `matrixCache`.



Obrázok 5.11: Transformácia objektu matrixCache

Podobne je prevedený objekt BVH stromu. Hierarchia polí naprieč vrstvami a histogramami je zachovaná. Rozdiel predstavujú konečné polia obsahujúce

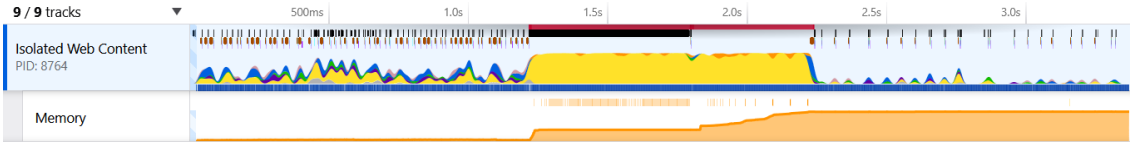
hodnoty BVH skupín. Rovnako ako pri štruktúre `matrixCache`, sú objekty obsahujúce hodnoty BVH podskupiny substituované za štvoricu polí pre pozíciu, veľkosť a indexy podskupín na pravej a ľavej vetve stromu. Keďže sa k hodnotám pristupuje prostredníctvom indexov, je arbitrárny atribút indexu podskupiny vypustený, keďže táto hodnota je implicitne daná jeho pozíciou v poli. Vizualizáciu transformácie BVH sromu je možné vidieť na obrázku 5.12.



Obrázok 5.12: Transformácia objektu BVHTree

Týmito krokmi sme dosiahli požadovanú rýchlosť prepočtu pomocných objektov histogramu. Miera GC sa znížila z 16 % na 1,2 % a pôvodný čas 4115 ms, sme dokázali skrátiť na 1002 ms. Dôsledkom prepísania pomocných objektov histogramu na typované polia sme taktiež dokázali markantne optimalizovať pamäť, oproti pôvodnému použitiu JavaScript natívnych polí s oddelenými objektmi, zaberajúcich značné množstvo pamäte spojenej s hlavičkou objektov[25]. Konečný graf profilácie je možné vidieť na obrázku 5.13.

Pri testovaní na ďalších histogramoch bola identifikovaná neoptimálna metóda, ktorá pre ohraničenia binov kontrolovala, či je niektorý z binov na nižšej vrstve viditeľný. Problém sa výrazne prejavil pri histograme so 7 parametrami, kde väčšina binov nebola zaplnená. V tomto prípade čas výpočtu dosiahol 17 sekúnd, pričom samotná metóda `isVisible` zaberala 16,4 sekundy. Implementácia lo-



Obrázok 5.13: Graf profilácie výkonu 3

giky tejto metódy je korektná, no príčinou preťaženia bola jej rekurzívna povaha. Metóda je mnohonásobne volaná v kóde, pričom tieto volania sú navyše násobené jej rekurziou. Overhead spojený s volaniami a opakované kontrolovania, tak významne spomaľovali celý prepočet. Túto metódu sme prepísali do iteratívneho postupu zdola-nahor, oproti rekurzívnemu postupu zhora-nadol. Iteratívna implementćia eliminovala overhead volaní pri rekurzii a obráteným postupom kontroly zdola-nahor sme zabezpečili, že každá inštancia je kontrolovaná najviac raz. Po tejto úprave sa čas prepočtu skrátil približne na jednu sekundu.

6 Vyhodnotenie

Cieľom tejto kapitoly je vyhodnotenie návrhu a implementácie riešenia z hľadiska splnenia stanovených cieľov práce, jeho výkonnostných vlastností a odozvy odbornej komunity. Vyhodnotenie sa zakladá na výsledkoch výkonnostných testov a na kvalitatívnej spätnej väzbe získanej od vývojárov knižnice ROOT.

V rámci práce bola vykonaná detailná analýza existujúceho riešenia NDMVR, na základe ktorej bol navrhnutý nový koncept vizualizácie n-dimenzionálnych histogramov. Tento koncept bol následne implementovaný s využitím pokročilých techník 3D grafiky, ako je inštancovanie geometrie, či raycasting optimalizovaný BVH algoritmom. Implementácia taktiež zahŕňala analýzu knižnice **JSROOT** a návrh algoritmov umožňujúcich jej integráciu do nového vizualizačného prostredia.

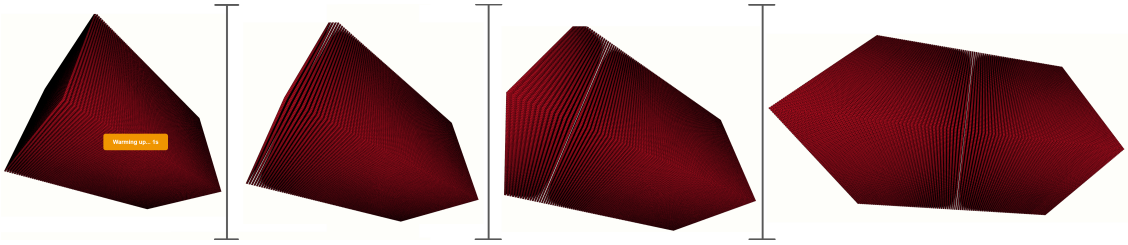
Vzhľadom na rozdiely v architektúre medzi pôvodným riešením NDMVR a novým systémom, no primárne kvôli výrazne odlišnej úrovni optimalizácie, sme zhodnotili, že vykonávať priame porovnanie ich výkonnosti by nebolo prínosné. Predbežné testovania, ktorých výsledky sú ilustrované v grafe uvedenom v predchádzajúcej kapitole, ukazujú niekoľkorádový rozdiel vo výkone. Z tohto dôvodu sa ďalšie vyhodnotenie sústreďuje výhradne na aktuálne riešenie. Hodnotenie pozostáva z výkonnostných testov a analýzy spätnej väzby odborníkov z vývojárskej komunity ROOT.

6.1 Výkonnostné testy

Pre dodatočnú validáciu sme vytvorili test, ktorého úlohou je kvantifikovať výkonnosť nášho riešenia. Merania v tejto podkapitole sú vykonávané na systéme s procesorom AMD Ryzen 5 2600, grafickou kartou Nvidia RTX 2060 s 6 GB VRAM, 16 GB DDR4 operačnej pamäte a operačným systémom Windows 10. Merania boli zobrazované na monitore, ktorý disponoval obnovovacou frekvenciou 75 Hz, preto vo výsledkoch je maximálna nameraná hodnota FPS limitovaná na 75. Test pozostáva z merania časových intervalov medzi snímkami vykresľova-

nia, z čoho je neskôr možné odvodiť počet snímok za sekundu. Prostredie pozostáva z Three.js scény, do ktorej sme umiestňovali objekty histogramov s premenlivým počtom binov. Pre plné potvrdenie našich tvrdení sme merania taktiež uskutočnili pre histogramy s rozdielnym základným typom a typom kontroly priesečníkov.

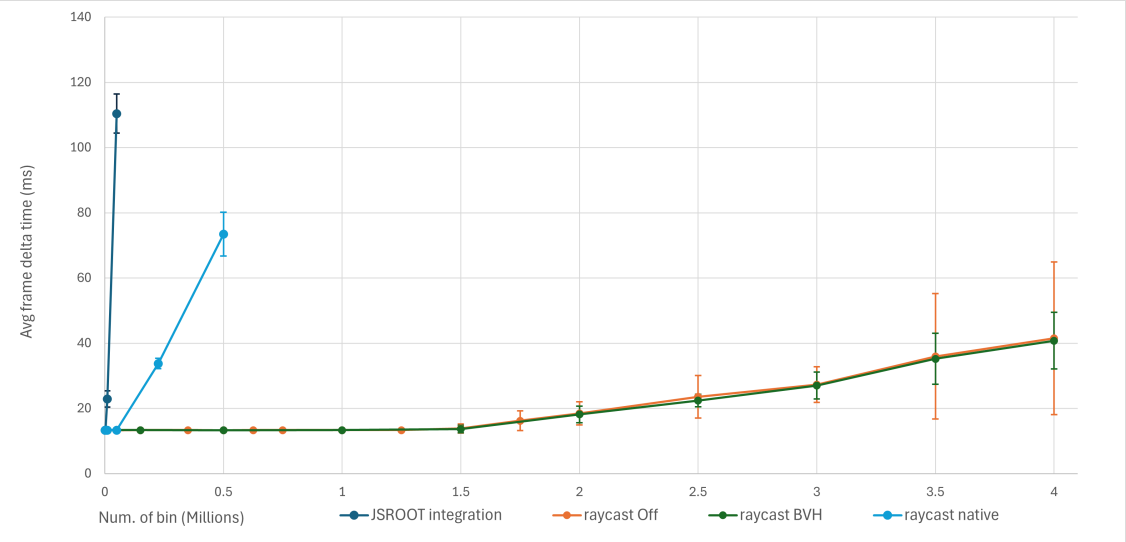
Chronologicky, test pozostával z 2-sekundového intervalu pre inicializáciu a ustálenie prostredia, následne pokračoval 10-sekundovým intervalom, počas ktorého boli merané časové intervaly medzi snímkami. Počas intervalu merania bola kamera posúvaná okolo objektu histogramu a zároveň bol vykonávaný pohyb myšou pre spúšťanie algoritmu kontroly intersekcie, ktorý sa vykonával pri frekvencii 1 ms. Snímky obrazovky počas vykonávania testu je možné vidieť na obrázku 6.1.



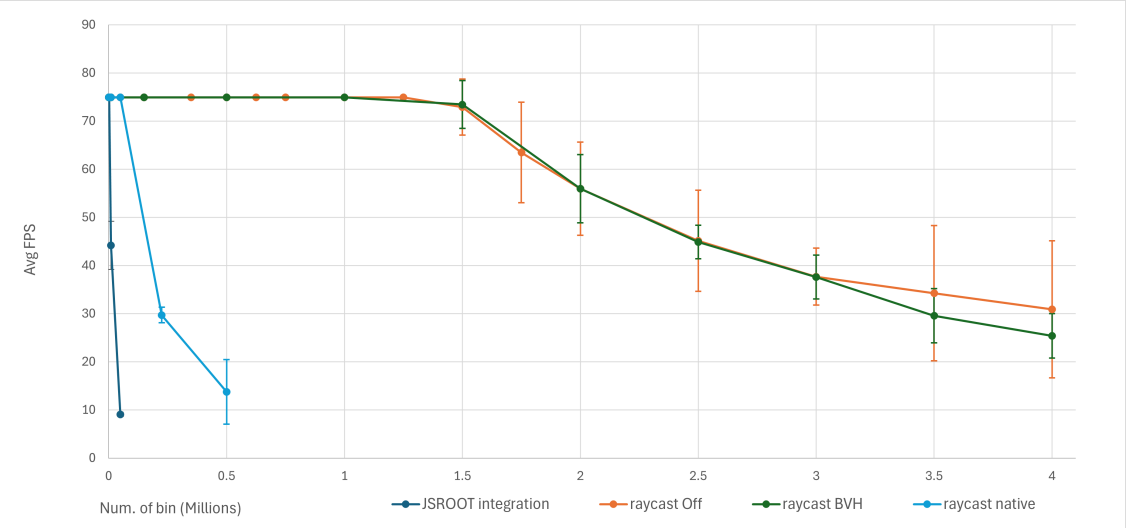
Obrázok 6.1: Snímky obrazovky počas merania

Po vykonaní merania boli dáta zapísané do lokálnej pamäte prehliadača, z ktorej sme ich vedeli exportovať pre dodatočnú analýzu. Výsledky analýzy je možné vidieť na grafoch 6.2 pre časové intervaly medzi snímkami proti počtu binov a na grafe 6.3 pre snímky za sekundu proti počtu binov. Na grafoch 6.2 a 6.3 sú taktiež zapísané smerodajné odchýlky jednotlivých hodnôt, keďže pre hodnoty v bodoch bola použitá ich stredná hodnota.

Vo výsledku sme potvrdili, že naše aktuálne riešenie ponúka najlepšiu výkonnosť pri zachovaní kontroly priesečníkov. Zároveň bolo dokázané, že použité `InstancedBufferGeometry` objekty doplnené o náš BVH algoritmus sú mimoriadne efektívne, nakoľko sa do 3 miliónov binov prakticky nevyskytoval rozdiel oproti absencii kontroly priesečníkov. Meranie ukázalo najhoršiu výkonnosť pre **JSROOT** integrované objekty. Tento výsledok je zapríčinený faktom, že v rámci dostupného času nebolo možné tieto objekty dodatočne optimalizovať. Hlavnou príčinou však bola nadmerná kontrola priesečníkov, keďže sme frekvenciu nechali na úrovni 1 ms pre integritu testu. V tomto prípade je dôležité dodať, že pri integrovaných **JSROOT** objektoch bez
glslinkray castingray casting-u bola výkonnosť približne rovnaká s naším riešením, keďže sú založené na rovnakom princípe inštancovania a podkladovom ob-



Obrázok 6.2: Graf výkonu pre časové intervaly medzi snímkami



Obrázok 6.3: Graf výkonu pre počet snímkov za sekundu

jekte s ním spojenom.

Dopad tohto problému je možné vidieť aj v implementácii knižnice **JSROOT**, kde je tento limit vyriešený pozastavením kontroly priesečníkov počas pohybu alebo rotovania objektu tak, aby pre používateľa nebol viditeľný pokles počtu snímok za sekundu. Keďže kamera je následne v scéne uložená staticky, čo znamená, že tento pokles nie je pre používateľa viditeľný. To by však pre nás nebolo vhodné riešenie, keďže požiadavkou bolo, aby naše objekty boli vhodné pre prostredie virtuálnej reality, v ktorej je kamera neustále dynamická.

6.2 Odozva ROOT vývojárov

Naše riešenie v kombinácii s implementáciou Bc. Oleha Leshchenka bolo prvýkrát prezentované na dvoch po sebe nasledujúcich mítingoch ROOT Physics Performance and Parallelism (PPP), ktoré sa konali v priestoroch pre ROOT tím v inštitúte CERN.

Na prvom mítingu[16] bolo predstavené naše riešenie spracovania n-dimenzionálnych histogramov prostredníctvom softvéru NDMSPC, ako aj nové riešenie vizualizačného systému NDMVR. Súčasťou prezentácie boli technologické aspekty riešenia a výsledky výkonnostných testov podporené živými demonštráciami.

Prvé demo bolo zamerané na ukážku kompatibility a konzistencie s knižnicou **JSROOT**, kde sme rovnaký histogram v našom prostredí vizualizovali pomocou **JSROOT** a nášho riešenia. Ďalšie dve demá slúžili na potvrdenie prezentovaných tvrdení výkonnostných testov systému. V prvom prípade bol použitý histogram s 500 000 binmi, pričom bola demonštrovaná interakcia s jednotlivými binmi v reálnom čase. V druhom prípade bol vykonaný stresový test na menšom histograme s variabilným počtom binov (približne 100). Tento test bol zameraný na ukážku času inicializačného vykreslenia. Týmto testom sme preukázali schopnosť systému pracovať s aktualizačnou periódou na úrovni jednotiek milisekúnd pre aktualizáciu objektov vizualizácie.

Ďalšie demo prezentovalo dynamické prepojenie vizualizačnej vrstvy s natívnou ROOT aplikáciou prostredníctvom websocket spojenia. Prezentácia bola taktiež doplnená o interaktívnu ukážku vo virtuálnej realite, v ktorej boli dostupné viaceré n-dimenzionálne histogramy odvodené z rovnakej trojrozmernej dátovej sady, pričom sa líšili spôsobom mapovania parametrov na osi a vrstvy histogramu.

Spätná väzba získaná počas mítingu poukázala najmä na vysokú úroveň plynulosti vizualizácie, škálovateľnosť riešenia a prínos imerzívneho zobrazenia dát. Pozitívne hodnotená bola aj integrácia s knižnicou **JSROOT**, ktorá viedla k začiatku diskusie s jej hlavným vývojárom Sergueiom Linevom.

Druhý míting[17] bol orientovaný predovšetkým na technické detaily spracovania dát v softvéri NDMSPC. Diskusia sa neskôr zamerala na možnosti silnejšej integrácie knižnice **JSROOT** do nášho prostredia. Táto diskusia viedla k pozitívnemu výstupu v podobe dohody o spísaní požiadaviek na rozšírenie funkciona-

[16] `https://indico.cern.ch/event/1589976/`
[17] `https://indico.cern.ch/event/1592358/`

lity **JSROOT** pre naše potreby.

Na základe prezentovaného riešenia sme boli pozvaní predstaviť náš softvér na ROOT workshope [18], kde nám bol ponúknutý priestor pri tzv. „poster session“ vo forme „hands-on“ ukážok pre účastníkov workshopu. Pre túto sériu relácií sme pripravili VR stánok pozostávajúci z VR prilieb Meta Oculus 2 a 3 prepojených na externý monitor na ukážku pre ostatných účastníkov a zároveň pre jednoduchšie inštruovanie aktuálnych testerov. Tieto relácie priniesli veľa pozitívnej odozvy, kde respondenti uznali potenciál v tomto riešení. Zároveň priniesli vecné otázky napríklad na kompatibilitu s inými VR prilbami či veľa vecných nápadov, pre zlepšenie nášho riešenia spojených s víziami, ako by im dokázalo naše riešenie pomôcť s vizualizáciami ich dát.

Odozva respondentov je taktiež zachytená v dotazníku. Ten podporuje naše tvrdenia o prehľadnosti a efektivite zobrazovania dát prostredníctvom n-dimenzionálnych histogramov. Kvôli časovej tiesni však dotazník vyplnila pomerne malá štatistická vzorka 7 respondentov z približne 35 účastníkov, ktorí naše riešenie testovali. Napriek menšiemu počtu respondentov išlo výhradne o expertov z cieľovej skupiny, čo zvyšuje kvalitatívnu hodnotu ich spätnej väzby, pričom dotazník zároveň poskytuje dobrý obraz o celkovej odozve. Výsledky celého dotazníka sa nachádzajú v prílohe A.

[18] `https://indico.cern.ch/event/1505384/`

7 Záver

V diplomovej práci sme pracovali na vytvorení softvérového riešenia, slúžiaceho na vizualizáciu n-rozmerných dát vo forme histogramov. Pre vytvorenie takéhoto riešenia, bola potrebná analýza prvotnej knižnice NdmVr, z časti projektu NdmSpc[3, 26, 1]. V nej bola rozobratá jeho architektúra, systém komunikácie medzi komponentmi a rovnako vykresľovanie komponentov s dôrazom na výkonnosť.

Pre želanú integráciu objektov balíka **JSROOT** musel byť analyzovaný z pohľadu štruktúry prepojenia jednotlivých tried, modulov a použitých objektov pre vykreslenie. Výsledkom bolo prvé úspešné extrahovanie 3D objektu vytvoreného v prostredí **JSROOT** do separátnej Three.js scény.

Hlavnou časťou pre získanie poznatkov a zistení, bola analýza pokročilých techník a metód vykresľovania. Konkrétne bola skúmaná technika inštancovania objektov pre optimalizáciu ich vykresľovania. Počas nej bolo porovnané natívne inštancovanie z balíka Three.js a inštancovanie z balíka a-frame-instanced-mesh v rámci nadstavbového rámca A-Frame. Ďalším skúmaným prístupom bola metóda kontroly intersekcie cez algoritmus hierarchického členenia objektov do BVH skupín. Záverom tejto časti bolo preukázanie výkonnosti techniky inštancovania objektov a zavedenie myšlienky kontroly priesečníkov algoritmom BVH.

Počas návrhu riešenia bol zadefinovaný prístup vizualizácie n-dimenzionálnych dát na základe hyperkocky, prepojenej s technikou vnorenia svetov do svetov[9]. Tento prístup umožňuje systematicky vizualizovať takéto dáta so súladom s praktickými zásadami na základe časti psychológie pre vyhľadávanie vzorov a ľudskú percepciu a kogníciu.

Súčasťou práce bolo aj vytvorenie infraštruktúry pre nasadenie riešenia, zloženého z MongoDB databázy a REST API rozhrania, ktoré sú nasadené v Kubernetes klastri. Celý proces vývoja a distribúcie riadia vytvorené CI/CD pipeline-y. Tie zabezpečujú kontinuálne nasadzovanie dema na Gitlab Pages a publikovanie verejného npm balíka.

Implementácia vizualizácie pozostávala z piatich hlavných častí. Prvou bola definícia formátu n-dimenzionálneho histogramu na základe stromovej štruktúry, ktorá umožňuje hierarchické vnáranie histogramov. Tento formát je čitateľný pre používateľa a zároveň efektívny pre implementáciu rekurzívneho algoritmu vykresľovania. Dôležitou súčasťou bolo zavedenie indexovania binov – indexovanie podľa ROOT špecifikácie pre používateľa, lineárne indexovanie pre objekt typu `InstancedMesh`, rozdelenie do matrix cache objektu a lineárne indexovanie pre ohraničenia binov.

Vytvorený algoritmus vykresľovania využíva rekurzívny prístup, kde postupne prechádza jednotlivými vrstvami stromu histogramu a vykresľuje biny na požadovanej úrovni. Pre efektívne vykresľovanie bol využitý objekt `InstancedBufferGeometry` z knižnice Three.js, ktorý umožňuje inštancovanie geometrie s vlastnými shadermi. Tým sme dosiahli úsporu pamäte, z dôvodu že každá inštancia zaberá len 6 Float hodnôt namiesto štandardných 16.

Zásadnou časťou implementácie bola optimalizácia algoritmu pre kontrolu intersekcie lúča s histogramom. Kde pôvodný Three.js raycaster má O(n) časovú zložitosť, čo pri histogramoch s veľkým počtom binov výrazne znižovalo výkon. Preto bol navrhnutý a implementovaný vlastný algoritmus založený na BVH štruktúre, ktorý využíva systematické uloženie binov v histograme. Tento algoritmus dosahuje O(log n) časovú zložitosť a pri histogramoch s viac ako 50 000 binmi poskytuje zrýchlenie v desiatkach tisíckrát za sekundu oproti pôvodnému riešeniu.

Počas vývoja bola vykonaná rozsiahla optimalizácia výpočtov, čím sa výrazne znížila pamäťová náročnosť a záťaž garbage collectora. Optimalizované boli taktiež matematické operácie nad vektormi a maticami. Výsledkom týchto optimalizácií bolo skrátenie času prepočtu pomocných objektov histogramu, kde sme pri 500 000 objektoch dosiahli približne 5-násobné zrýchlenie výpočtu.

Do nášho riešenia bola integrovaná knižnica **JSROOT**, pričom spolupráca s jej hlavným vývojárom viedla k pridaniu novej funkcionality build3d do samotnej knižnice **JSROOT**. Táto integrácia umožňuje používateľom v našom prostredí vizualizovať histogramy buď pomocou nášho optimalizovaného riešenia, alebo pomocou štandardných **JSROOT** objektov, čím sa zachováva kompatibilita s existujúcim ekosystémom ROOT-u.

Výkonnostné testy ukázali, že riešenie ponúka plynulé vykresľovanie pri histogramoch do 3 miliónov binov. Testy zároveň preukázali schopnosť systému reagovať s latenciou na úrovni jednotiek milisekúnd, čo umožňuje dynamické prepojenie s natívnou ROOT aplikáciou prostredníctvom websocket spojenia.

Riešenie bolo prezentované na dvoch ROOT PPP mítingoch v inštitúte CERN a na ROOT workshope, kde získalo pozitívnu odozvu od vývojárskej komunity. Respondenti ocenili najmä plynulosť vizualizácie, škálovateľnosť riešenia a potenciál imerzívneho zobrazovania dát vo virtuálnej realite. Výsledky dotazníkového prieskumu podporujú tvrdenia o prehľadnosti a efektivite zobrazovania n-dimenzionálnych dát pomocou navrhnutého konceptu hyperkocky.

Celkovo práca priniesla efektívne riešenie pre vizualizáciu n-dimenzionálnych histogramov, ktoré je výkonné, širokokompatibilné a dostupné pre použitie v aplikáciách fyzikálnych výskumov. Riešenie je publikované ako npm balík a je priamo integrovateľné v projektoch postavených na knižnici Three.js. Na komunikáciu s koncovou aplikáciou používa RxJs subjekty. Toto rozhranie umožňuje modularitu a jednoduchú integráciu do rôznych prostredí.

V rámci ďalšieho vývoja sa ponúka niekoľko vylepšení. Systém by sa dal rozšíriť o podporu dodatočných ROOT objektov a používateľské funkcie, ako je filtrovanie dát či výber konkrétneho rozsahu. Pre lepšiu prehľadnosť by histogramom pomohlo pridanie osí a plynulé animácie pri prechode medzi ich vrstvami. Pre zaručenie stability kódu by taktiež bolo vhodné vytvorenie automatizovaných testov.

Zoznam použitej literatúry

1. KORECKO, Stefan; VAĽA, M.; FEKETE, M. VISUALIZATION OF EXPERIMENTAL DATA IN WEB-BASED VIRTUAL REALITY. In: 2021, s. 149–152. Dostupné z DOI: 10.54546/MLIT.2021.42.97.001.
2. ROOT TEAM. *JSROOT* [https://root.cern/js/]. 2025. JavaScript ROOT data visualization.
3. FEKETE, Martin. *Vizualizácia experimentálnych údajov v zdieľanej rozšírenej realite a jej používateľské rozhranie*. Katedra počítačov a informatiky, 2021. Dipl. pr. Technická Univerzita v Košiciach.
4. CHOVANEC, Daniel. *Univerzálny ovládač používateľského vstupu pre virtuálno-realitné prostredie vizualizujúce experimentálne údaje*. Katedra počítačov a informatiky, 2024. Dipl. pr. Technická Univerzita v Košiciach.
5. THREE.JS AUTHORS. *Three.js* [https://threejs.org/]. 2025. JavaScript 3D library.
6. AKENINE-MÖLLER, T.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering, Fourth Edition*. CRC Press, 2018. ISBN 9781351816151. Dostupné tiež z: https://books.google.sk/books?id=0g1mDwAAQBAJ.
7. PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically Based Rendering, fourth edition: From Theory to Implementation*. MIT Press, 2023. ISBN 9780262048026. Dostupné tiež z: https://pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies.
8. RUBIN, Steven; WHITTED, Turner. A 3-dimensional representation for fast rendering of complex scenes. In: 1980, zv. 14. Dostupné z DOI: 10.1145/965105.807479.
9. FEINER, S. K.; BESHERS, Clifford. Worlds within worlds: metaphors for exploring n-dimensional virtual worlds. In: *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*. Snowbird,

Utah, USA: Association for Computing Machinery, 1990, s. 76–83. UIST '90. ISBN 0897914104. Dostupné z DOI: 10.1145/97924.97933.

10. KEIM, D.A.; KRIEGEL, H.-P. VisDB: database exploration using multidimensional visualization. *IEEE Computer Graphics and Applications*. 1994, roč. 14, č. 5, s. 40–49. Dostupné z DOI: 10.1109/38.310723.
11. SANTOS, Selan; BRODLIE, Ken. Visualizing and investigating multidimensional functions. 2002, 173–ff. ISBN 1-58113-536-X.
12. WARE, Colin. Information Visualization: Perception for Design: Second Edition. In: 2004. ISBN 9780080478494.
13. TODOROVIC, D. Gestalt principles. *Scholarpedia*. 2008, roč. 3, č. 12, s. 5345. Dostupné z DOI: 10.4249/scholarpedia.5345. revision #91314.
14. RENSINK, Ronald. Change Detection. *Annual Review of Psychology*. 2002, roč. 53, s. 245–77. Dostupné z DOI: 10.1146/annurev.psych.53.100901.135125.
15. ASTRELIN, Andrey [online]. [B.r.]. [cit. 2025-11-17]. Dostupné z: https://superliminal.com/andrey/mc7d/.
16. A-FRAME AUTHORS. *A-Frame: WebVR Framework* [https://aframe.io/]. 2025. Framework for building VR experiences.
17. GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. Addison-Wesley Professional Computing Series. ISBN 9780321700698. Dostupné tiež z: https://books.google.sk/books?id=6oHuKQe3TjQC.
18. HAYWARD, William. Effects of Outline Shape in Object Recognition. *Journal of Experimental Psychology: Human Perception and Performance*. 1998, roč. 24, s. 427–440. Dostupné z DOI: 10.1037/0096-1523.24.2.427.
19. *Histograms* [online]. ROOT Team [cit. 2026-02-13]. Dostupné z: https://root.cern/manual/histograms/.
20. BLACK, Paul E. *big-O notation* [https://www.nist.gov/dads/HTML/bigOnotation.html]. 2019. [cit. 2026-02-17].
21. CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd. MIT Press, 2009. ISBN 978-0-262-03384-8. Dostupné tiež z: http://mitpress.mit.edu/books/introduction-algorithms.

22. KNUTH, Donald E. Big Omicron and big Omega and big Theta. *SIGACT News*. 1976, roč. 8, č. 2, s. 18–24. ISSN 0163-5700. Dostupné z DOI: 10.1145/1008328.1008329.

23. KENZEL, Michael; KERBL, Bernhard; SCHMALSTIEG, Dieter; STEINBERGER, Markus. A high-performance software graphics pipeline architecture for the GPU. *ACM Trans. Graph.* 2018, roč. 37, č. 4. ISSN 0730-0301. Dostupné z DOI: 10.1145/3197517.3201374.

24. MOZILLA CONTRIBUTORS. *MDN Web Docs: JavaScript* [https://developer.mozilla.org/en-US/docs/Web/JavaScript]. 2025. Comprehensive JavaScript documentation.

25. BRUN, Camillo. *Fast properties in V8* [https://v8.dev/blog/fast-properties]. 2017. [cit. 2025-10-17].

26. PROKOP, Natanael. *Vylepšenie používateľského rozhrania virtuálneho prostredia pre vizualizáciu experimentálnych údajov*. Katedra počítačov a informatiky, 2025. Dipl. pr. Technická Univerzita v Košiciach.

27. KOŠICE, FEEI Technical University of. K slovenskému prekladu dotazníkov na meranie používateľskej skúsenosti vo virtuálnej realite. In: 2025. ISBN 978-80-553-4854-4. Dostupné tiež z: https://eei.fei.tuke.sk/wp-content/uploads/2025/07/EEI_16.pdf.

Slovník

Bundle výsledný súbor alebo sada optimalizovaných súborov vygenerovaných nástrojom na zostavovanie (napr. Vite), ktoré obsahujú skombinovaný a minifikovaný zdrojový kód aplikácie vrátane jej závislostí pripravený na produkčné nasadenie..

CI/CD (Continuous Integration / Continuous Deployment) metóda na časté doručovanie aplikácií zákazníkom prostredníctvom zavedenia automatizácie do fáz vývoja aplikácií; konkrétne ide o kontinuálnu integráciu, kontinuálne doručovanie a kontinuálne nasadzovanie..

Commit operácia v systéme Git, ktorá vytvorí nemenný záznam aktuálneho stavu súborov v repozitári, čím ukladá históriu zmien spolu s identifikačnými údajmi autora a popisnou správou..

Context mechanizmus knižnice React, ktorý umožňuje zdieľať hodnoty (stavy) medzi komponentmi bez nutnosti manuálneho odovzdávania vlastností (props) cez každú úroveň stromu komponentov..

CRD (Custom Resource Definition) vlastný zdroj umožňujúci rozšíriť Kubernetes API definovaním vlastných objektov bez potreby úpravy zdrojového kódu samotného Kubernetes..

CRUD (Create, Read, Update, Delete) štyri základné operácie trvalého úložiska v počítačovom programovaní: vytváranie, čítanie, aktualizácia a mazanie..

FPS (Frames Per Second) frekvencia, s akou po sebe nasledujúce obrázky (snímky) tvoria videozáznam alebo obraz generovaný počítačom..

garbage collector (GC) Mechanizmus automatickej správy pamäte v prostredí JavaScriptu, ktorého primárnou úlohou je monitorovať alokáciu objektov a uvoľňovať pamäť obsadenú tými entitami, ktoré už aplikácia nepotrebuje..

Hook špeciálna funkcia v knižnici React, ktorá umožňuje vývojárom využívať stav a ďalšie funkcie Reactu (napr. životný cyklus komponentu).

Ingress API objekt, ktorý spravuje externý prístup k službám v klastri (zvyčajne HTTP) a poskytuje vyvažovanie záťaže, ukončenie SSL/TLS a smerovanie založené na názvoch..

Keycloak open-source produkt na správu identít a prístupu, ktorý umožňuje zabezpečenie aplikácií a služieb s minimálnym množstvom kódu..

Label pár kľúča a hodnoty pripojený k objektom (napríklad k podom), ktorý slúži na špecifikáciu identifikačných atribútov objektov zmysluplných pre používateľa, pričom priamo neovplyvňuje sémantiku jadra systému..

npm (Node Package Manager) správca balíkov pre programovací jazyk JavaScript a predvolený správca balíkov pre runtime prostredie Node.js..

OLM (Operator Lifecycle Manager) nástroj, ktorý pomáha používateľom inštalovať, aktualizovať a spravovať životný cyklus všetkých operátorov a ich pridružených služieb v klastri Kubernetes..

Pod najmenšie výpočtové jednotky, ktoré je možné v Kubernetes vytvoriť a spravovať; ide o skupinu jedného alebo viacerých kontajnerov so zdieľanými úložnými a sieťovými zdrojmi..

Push príkaz v systéme Git, ktorý odosiela lokálne uložené záznamy zmien (commity) do vzdialeného repozitára na serveri, čím synchronizuje prácu medzi lokálnym prostredím a zdieľaným úložiskom..

ray casting metóda určovania priesečníkov lúča s objektmi v scéne, pri ktorej sa zisťujú všetky body prieniku pozdĺž dráhy lúča, avšak bez výpočtu sekundárnych odrazov, lomov alebo iných fyzikálnych interakcií svetla..

ray tracing technika vykresľovania 3D grafiky, ktorá simuluje fyzikálne správanie svetla sledovaním dráhy lúčov od zdroja (alebo kamery) a ich interakcií s povrchmi objektov, vrátane odrazov, lomov a tieňov..

REST (Representational State Transfer) architektonický štýl navrhnutý pre sieťové aplikácie, ktorý využíva bezstavový komunikačný protokol na manipuláciu so zdrojmi prostredníctvom štandardizovaných operácií..

Secret objekt umožňujúci ukladať a spravovať citlivé informácie, ako sú heslá, OAuth tokeny a SSH kľúče..

Service spôsob, ako vystaviť aplikáciu bežiacu v klastri za jediným navonok smerujúcim koncovým bodom (endpoint), a to aj v prípade, že je záťaž rozdelená medzi viacero backendov..

Zoznam príloh

Príloha A Používateľský dotazník

Príloha B Používateľská príručka

Príloha C CD médium – záverečná práca v elektronickej podobe,

Príloha D Repozitár výsledného projektu je dostupný na adrese
`https://gitlab.com/ndmspc/ndmvr-core`

A Používateľský dotazník

Výsledky dotazníka sú rozdelené do viacerých tabuliek kvôli obmedzenému rozsahu strany, respondenti ho však vypĺňali ako celok v presnom poradí, v akom sú nižšie uvedené. Keďže išlo o zahraničných respondentov, dotazník bol vypracovaný v anglickom jazyku. Z dôvodu zachovania integrity sú preto otázky uvedené v originálnom znení. Interpretovať ich do slovenského jazyka je možné podľa vedeckého príspevku v zborníku FEI TUKE. [27]

Tabuľka A.1: Používateľský dotazník 1

	No. of answers
Age	
21 25 26 29 34 38	
2 1 1 1 1 1	
Gender	
Male	5
Female	2
What is Your relation to ROOT?	
User: I use ROOT for data analysis.	6
Developer: I develop for ROOT.	1
How do you browse/view ROOT files?	
ROOT native GUI (TBrowser)	4
Batch version (without GUI)	3
JSROOT (web browser)	3

Tabuľka A.2: Používateľský dotazník 2

Statement	1	2	3	4	5	6	7
1 = fully disagree, 7 = fully agree							
I understood histogram visualisation in NDMVR.	0	0	0	1	1	2	3
The display of bin information is well-designed.	0	0	1	0	0	4	2
The virtual reality mode provides a much richer and more effective data visualization.	0	0	0	1	3	2	1
1 = not at all, 7 = very much							
In the computer generated world I had a sense of "being there".	0	0	0	1	3	1	2
1 = fully disagree, 7 = fully agree							
Somehow I felt that the virtual world surrounded me.	0	0	0	2	1	2	2
I felt like I was just perceiving pictures.	0	1	1	4	0	1	0
1 = did not feel present, 7 = felt present							
I did not feel present in the virtual space.	1	2	0	3	0	0	1
1 = fully disagree, 7 = fully agree							
I had a sense of acting in the virtual space, rather than operating something from outside.	0	1	0	1	2	1	2
I felt present in the virtual space.	0	0	0	2	2	1	2
1 = extremely aware, 7 = not aware at all							
How aware were you of the real world surrounding while navigating in the virtual world? (i.e. sounds, room temperature, other people, etc.)	0	0	1	0	6	0	0

Tabuľka A.3: Používateľský dotazník 3

Statement	1	2	3	4	5	6	7
1 = fully disagree, 7 = fully agree							
I was not aware of my real environment.	0	1	2	2	2	0	0
I still paid attention to the real environment.	0	0	3	1	3	0	0
I was completely captivated by the virtual world.	0	1	1	0	3	1	1
1 = completely real, 7 = not real at all							
How real did the virtual world seem to you?	0	1	0	3	0	3	0
1 = not consistent, 7 = very consistent							
How much did your experience in the virtual environment seem consistent with your real world experience?	0	0	4	0	2	1	0
1 = about as real as an imagined world, 7 = indistinguishable from the real world							
How real did the virtual world seem to you?	0	5	1	0	1	0	0
1 = about as real as an imagined world, 7 = indistinguishable from the real world							
The virtual world seemed more realistic than the real world.	4	1	2	0	0	0	0

Tabuľka A.4: Používateľský dotazník 4 (0: not at all, 3: very)

Symptom	0	1	2	3
General discomfort	4	3	0	0
Fatigue	5	2	0	0
Eyestrain	2	5	0	0
Difficulty focusing	3	3	1	0
Headache	5	2	0	0
Fullness of head	4	3	0	0
Blurred vision	2	4	1	0
Dizzy (eyes closed)	4	3	0	0
Vertigo	5	1	1	0

B Používateľská príručka

Táto príručka opisuje predpoklady na spustenie, návod na inštaláciu, konfiguráciu a začiatočné kroky používania knižnice `NDMVR-Core`.

Hlavným zdrojom pre získanie systémovej príručky a všetkých dostupných informácii je dokumentácia v elektronickej podobe, dostupná na adrese `https://ndmspc.gitlab.io/ndmvr-core/docs/`.

B.1 Požiadavky a predpríprava

Pred začatím používania knižnice a je potrebné splniť nasledujúce technické požiadavky:

- **Node.js:** Odporúčaná verzia v18.20.2 alebo novšia (podpora ES modulov a kompatibilita závislostí).
- **Základné znalosti:** Predpokladá sa znalosť základného prostredia Three.js (scéna, kamera) a JavaScriptu (ES6+, asynchrónne operácie).
- **Lokálny webový server:**(nepovinné) Kvôli bezpečnostným obmedzeniam prehliadačov (CORS) nie je možné súbory otvárať priamo cez protokol `file://`. Odporúča sa použiť rozšírenie *Live Server* pre VS Code, alebo príkazy `npx serve` či `python -m http.server`.

B.2 Spôsoby začatia

Knižnicu je možné začať používať tromi spôsobmi v závislosti od potrieb používateľa:

1. **GitLab Pages (Rýchly náhľad):** Pre okamžité zobrazenie možností knižnice bez nutnosti inštalácie sú dostupné interaktívne ukážky na oficiálnych stránkach projektu.

2. **Klonovanie repozitára (Vývoj jadra):** Ak je cieľom upravovať priamo komponenty knižnice, je potrebné naklonovať repozitár a nainštalovať závislosti:

   ```
   git clone https://gitlab.com/ndmspc/ndmvr-core.git
   npm install
   npm run dev
   ```

3. **Inštalácia cez NPM (Integrácia):** Pre použitie vo vlastnom projekte sa balík nainštaluje pomocou Node manažéra balíkov:

   ```
   npm install @ndmpc/ndmvr-core
   ```

B.3 Vizualizácia dát pomocou THnPainter

Hlavným nástrojom knižnice je trieda `THnPainter`, ktorá automatizuje proces vykresľovania ROOT histogramov (TH1, TH2, TH3 až THn).

B.3.1 Dátový tok a spracovanie

Proces transformácie surových dát na 3D objekt v scéne prebieha v niekoľkých krokoch:

1. **Načítanie dát:** Import JSON súboru vygenerovaného z frameworku ROOT.
2. **Parsovanie:** Použitie funkcie `parse` z knižnice `jsroot` na transformáciu JSON do JavaScript objektu.
3. **Inicializácia Painter-a:** Vytvorenie inštancie `THnPainter`, ktorá vygeneruje geometriu a materiály.
4. **Pridanie do scény:** Vloženie vygenerovaných sietí (*mesh* a *wireframe*) do Three.js scény.

B.3.2 Praktická implementácia

Príklad základnej implementácie v rámci JavaScript modulu:

Výpis B.1: Príklad inicializácie histogramu

```
import * as THREE from "three";
import { THnPainter } from "ndmvr-core";
import { parse } from "jsroot";
import data from "./h3scat.json" with { type: "json" };

async function loadHistogram() {
  try {
    const histoObj = { obj: parse(data) };
    const painter = new THnPainter(histoObj, "histogram1");

    // Pridanie objektov do scény
    scene.add(painter.mesh);
    scene.add(painter.wireframe.wireframe);

    console.log("Histogram šúspene čnaítaný");
  } catch (error) {
    console.error("Chyba:", error);
  }
}
```

B.4 Konfigurácia prostredia

Pre správne fungovanie v čistom HTML prostredí je nevyhnutné definovať `importmap`, aby prehliadač vedel lokalizovať potrebné moduly.

Výpis B.2: Štruktúra HTML súboru

```
<!DOCTYPE html>
<html>
<head>
  <script type="importmap">
    {
      "imports": {
        "three": "https://unpkg.com/three/build/three.module.js",
        "jsroot": "https://root.cern/js/latest/modules/main.mjs",
        "ndmvr-core": "./dist/ndmvr-core.js"
```

```
      }
    }
  </script>
</head>
<body>
  <script type="module">
    // Volanie čvizualizaných funkcií
  </script>
</body>
</html>
```

B.5 Pokročilé funkcie

Knižnica `NDMVR-Core` podporuje ďalšie rozširovanie, ako je interaktivita pomocou Raycastingu, dynamické aktualizácie dát cez RxJS alebo integrácia do virtuálnej reality prostredníctvom projektu `NdmVr` postaveného na React-Three-Fiber.