Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Vizualizácia experimentálnych údajov v rozšírenej realite s prepojením na distribuované počítanie

Diplomová práca

2023 Bc. Martin Fekete

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Vizualizácia experimentálnych údajov v rozšírenej realite s prepojením na distribuované počítanie

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	RNDr. Martin Vaľa, PhD.
Konzultant:	Ing. Štefan Korečko, PhD.

Košice 2023 **Bc. Martin Fekete**

Abstrakt v SJ

Táto diplomová práca ma za úlohu zjednotiť výsledky predošlých prác týkajúcich sa výpočtov, monitorovania a vizualizácie experimentálnych údajov v oblasti fyziky vysokých energií a vytvoriť funkcionality na implementáciu vizualizačných komponentov. Úlohou je analyzovať dátové vizualizácie experimentálnych dát použitím rozšírenej reality a dostupné funkcionality, slúžiace na spúšťanie, monitorovanie a vizualizáciu dát obsiahnuté v knižniciach NDMVR, JSROOT, REACT-NDMSPC-CORE a REACT-NDMSPC. V implementačnej časti sú popísané konceptuálne návrhy vylepšení knižníc spolu ich implementáciou a demonštráciou ich použitia, čo predstavuje implementovaný komponent predstavujúci zobrazované dáta vo forme histogramu s možnosťou spúšťania úloh na klástri, monitorovania stavov a následne zobrazenie výsledkov vo forme dátovej projekcie. Vo vyhodnotení sú následne zhrnuté a otestované dosiahnuté ciele tieto práce spolu s definíciou ďalších vylepšení.

Kľúčové slová v SJ

rozšírená realita, dátová analýza, JSROOT, React, vizualizácia, histogram, A-Frame, javascript

Abstrakt v AJ

The purpose of this master thesis is to unify the results of previous works related to the calculation, monitoring and visualization of experimental data in high-energy physics and to improve and create new features for the visualization component's implementation. The task is to analyze data visualizations of the experimental data using augmented reality and available features such as job initialization, monitoring and visualization contained in the NDMVR, JSROOT, REACT-NDMSPC-CORE and REACT-NDMSPC libraries. The implementation part describes the conceptual proposals for improvements to the libraries are described together with their implementation and demonstration of their use, which represents an implemented component representing displayed data as a histogram with the possibility such as launching jobs on the cluster, monitoring and showing the results as data projection. Finally, There are evaluations of implemented features with definitions of possible future improvements.

Kľúčové slová v AJ

augmented reality, data analysis, JSROOT, React, visualization, histogram, A-Frame, javascript

Bibliografická citácia

FEKETE, Martin. *Vizualizácia experimentálnych údajov v rozšírenej realite s prepojením na distribuované počítanie*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2023. 83s. Vedúci práce: RNDr. Martin Vaľa, PhD.

68020

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**

Študijný program: **Informatika**

Názov práce:

Vizualizácia experimentálnych údajov v rozšírenej realite s prepojením na distribuované počítanie

Experimental Data Visualization in Web-based Extended Reality with Distributed Computation

Študent: **Bc. Martin Fekete**

Školiteľ: **RNDr. Martin Vaľa, PhD.**

Školiace pracovisko:

Konzultant práce: **Ing. Štefan Korečko, PhD.**

Pracovisko konzultanta: **Katedra počítačov a informatiky**

Pokyny na vypracovanie diplomovej práce:

1. Analyzovať aktuálne prístupy k vizualizácií experimentálnych údajov v rozšírenej realite.
2. Analyzovať súčasný stav dostupných knižníc obsahujúcich komponenty pre riadenie získavania údajov z fyzikálnych experimentov a ich vizualizáciu.
3. Na základe analýzy navrhnúť rozšírenia knižníc, umožňujúce spravovať získavanie údajov a ich vizualizáciu.
4. V rámci možností overiť implementované riešenie za použitia údajov z reálnych experimentov.
5. Návrh a implementáciu koordinovať s ďalšími riešiteľmi systému pre vizualizáciu experimentálnych údajov.
6. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský

Termín pre odovzdanie práce: 21.04.2023

Dátum zadania diplomovej práce: 31.10.2022

prof. Ing. Liberios Vokorokos, PhD.

dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 21.4.2023 . . . .

Vlastnoručný podpis

Poďakovanie

Na tomto mieste by som rád poďakoval svojmu vedúcemu práce a konzultantovi za jeho čas, odborné vedenie a cennú pomoc počas riešenia mojej záverečnej práce a to hlavne za odbornosť, skúsenosti, trpezlivosť, motiváciu a povzbudenie počas ťažkých chvíľ, ktorých bolo nespočetne veľa. Takisto by som chcel poďakovať aj za neustále kladenie vysokých cieľov vďaka, ktorým sa podarilo tuto prácu dokončiť na vysokej úrovni.

Rovnako by som sa rád poďakoval svojim rodičom a priateľom za ich podporu a povzbudzovanie počas celého môjho štúdia.

V neposlednom rade by som sa rád poďakoval pánom *Donaldovi E. Knuthovi* a *Leslie Lamportovi* za typografický systém LaTeX.

Obsah

Úvod **1**

1 Analytická časť **3**

1.1 Vizualizácia dátových analýz vo virtuálnej realite 3
1.1.1 Vizualizácia dát, fyzická a virtuálna 3
1.1.2 Vizualizácia vedeckých dát vo VR 6
1.1.3 Vizualizácie a vizualizačné prostredia VR 10
1.1.4 Genuage 11
1.1.5 Zhrnutie dátových analýz vo VR 13
1.2 Analýza Knižnice NDMVR a JSROOT 14
1.2.1 NDMVR 14
1.2.2 Analýza nedostatkov NDMVR 16
1.2.3 REACT-NDMSPC-CORE 18
1.2.4 JSROOT 19
1.2.5 JSROOT migrácia na 7.1.0 22
1.3 Analýza monitorovacej aplikácie pre systém SALSA 23
1.4 Technológie pre zostavovanie webových aplikácií 25
1.4.1 Webpack 26
1.4.2 Vite 26
1.4.3 Parcel 27
1.4.4 Zhrnutie a výber technológie 27

2 Návrh riešenia **28**

2.1 Definícia požiadaviek 29
2.1.1 Diagram prípadov použitia 29
2.1.2 Zhrnutie požadovaných funkcionalít 30
2.2 Návrh komponentu NDMSPC 31
2.2.1 Architektúra REACT-NDMSPC po aplikovaní koncepčných riešení 32

2.2.2 Funkčné komponenty REACT-NDMSPC 34
2.2.3 Koncept spracovania udalostí 34
2.2.4 Koncept modifikácie vizualizácie na základe vonkajších parametrov 36
2.2.5 Komunikácia medzi nezávislými komponentami a zdieľanie stavov 37
2.3 Návrh vylepšení NDMVR 38
2.3.1 Nová architektúra NDMVR po aplikovaní koncepčných riešení 39
2.4 Návrh vylepšení REACT-NDMSPC-CORE 40
2.4.1 Subjekt pre vykonávanie funkcií 40
2.4.2 Import knižnice JSROOT v7.2.1 a vytvorenie súvisiacich funkcionalít 42
2.4.3 Návrh systému pre distribúciu údajov medzi komponentami 43
2.5 Usporiadanie funkcionalít a minimalizácia závislostí medzi knižnicami 45

3 Implementácia 47

3.1 Implementácia vylepšení knižnice NDMVR 47
3.1.1 Refaktorizácia NDMVR na moduly 47
3.1.2 Rozšírenie vizualizácie TH1 histogramu 48
3.1.3 Rozšírenia distribúcie dát o binoch mimo komponent 49
3.1.4 Rozšírenie pre definovanie vzhľadu binov na základe externej funkcie 51
3.2 Implementácia vylepšení knižnice REACT-NDMSPC-CORE 53
3.2.1 NdmspGlobalScope - charakteristika a atribúty 53
3.2.2 NdmspGlobalScope - spracovávanie získaných údajov . . . 54
3.2.3 NdmspGlobalScope - dátový distribútor 56
3.3 Implementácia komponentu knižnice NDMSPC 56
3.3.1 Jednoduchý komponent pre selekciu binov 57
3.3.2 Implementácia hlavného kontextu a distribútorov 58
3.3.3 Implementácia interakcie používateľa 59
3.3.4 Implementácia logiky komponentu 60
3.3.5 Implementácia vizualizačných komponentov 1 a 2 61
3.3.6 Implementácia logiky komponentu REACT-NDMSPC vyvíjaného v práci 66
3.3.7 Implementácia kontextu 66
3.3.8 Implementácia používateľských interakcií 67

3.3.9 Implementácia spracovania dát o stavoch úloh 68
3.3.10 Komponent pre vizualizáciu 70

4 Vyhodnotenie **71**
4.1 Vývoj vizualizačného komponentu 71
4.1.1 Monitorovanie úloh na klástri 71
4.1.2 Prehliadač ROOT súborov 73
4.1.3 Konfigurátor histogramov 73
4.1.4 Zobrazovanie projekcií 74
4.2 Vývoj vizualizačného komponentu knižnice REACT-NDMSPC-CORE 75
4.3 Vylepšenia komponentu NdmvrScene 76
4.3.1 Generovanie entít histogramu 76
4.3.2 Modifikovanie atribútov z vonkajšieho kontextu 76
4.3.3 Definovanie spätne volaných funkcií pre spracovanie udalostí 77
4.4 Zhrnutie prínosu riešenia a ďalšie rozšírenia 78

5 Záver **79**

Literatúra **81**

Zoznam skratiek **84**

Zoznam príloh **85**

Zoznam obrázkov

1.1 Vizualizácie pre aplikáciu testovania interakcií. (Zdroj: [9]) 4
1.2 Výsledky a porovnanie spätnej väzby účastníkov v oblasti jednoduchosti, rýchlosti, výkonu a zdieľania (Zdroj: [9]) 5
1.3 Výsledky a porovnanie potrebného času na vykonanie operácií. (Zdroj: [9]) 6
1.4 Ukážka grafického používateľského rozhrania vo virtuálnej realite pre viazanie dát a objektov. (Zdroj: [12]) 8
1.5 Príklad vizualizácií experimentálnych vedeckých údajov. (Zdroj: [13]) 9
1.6 Zreťazenie vizualizačných aspektov integrovaných s responzívnym virtuálnym prostredím (Zdroj: [14]) 10
1.7 Príklad vizualizácie použitím softvéru genuage (Zdroj: [15]) . . . 12
1.8 Softvérová architektúra knižnice NDMVR (Zdroj: [22]) 15
1.9 Názorná ukážka zobrazenia vizualizácie histogramu TH2 pomocou NDMVR (Zdroj: [22]) 16
1.10 Experimentálne prostredie NDMVR 17
1.11 Komponentový diagram systému pre spravovanie úloh (Zdroj: [32]) 24
1.12 Sekvenčný diagram systému (Zdroj: [32]) 24

2.1 Diagram prípadov použitia 30
2.2 Konceptuálny model architektúry komponentu NDMSPC 33
2.3 Model popisujúci koncept spracovávania udalostí mimo vizualizačný komponent 35
2.4 Model popisujúci koncept modifikácie vizualizácie na základe parametrov mimo vizualizačný komponent 36
2.5 Zobrazenie konceptu distribútora dát medzi rôznymi komponentami v aplikácii 37
2.6 Konceptuálny model novej architektúry knižnice NDMVR 39

2.7 Návrh kompozície pre riešenie exekútora funkcií 41
2.8 Zobrazenie konceptu hlavného objektu **AppGlobalScope** pre distribúciu dát do vzdialených komponentov 44
2.9 Zobrazenie závislosti medzi knižnicami pred refaktorizáciou . . . 45
2.10 Zobrazenie závislosti medzi knižnicami po preusporiadaní a refaktorizácii 46

3.1 Zobrazenie základného konceptu vyvíjaného komponentu 57
3.2 Vizualizácia komponentu 1 s označením binov 65
3.3 Vizualizácia komponentu 2 s označením binov 65

4.1 Zobrazenie stavu úloh vykonávaných na klástri. 72
4.2 Zobrazenie stavu úloh vykonávaných na klástri vo virtuálnej realite, implementované v tejto diplomovej práci. 72
4.3 Zobrazenie vizualizácie histogramu z otvoreného root súboru. V histograme sú zobrazené aj stavy bežiacich úloh reprezentujúce biny na ktorých boli spustené. 73

Úvod

Virtuálna realita poskytuje používateľovi nový druh zážitku, je využívaná v čoraz rozsiahlejších doménach aj za účelom dátových analýz. V prípade dátovej analýzy ponúka nový pohľad na dáta, ktoré vieme, ako používatelia vyčítať z prostredia a patrične na nich reagovať. Táto možnosť interakcie používateľa s dátami je skvelá ale, aby to bolo možné je nutné preto údaje patrične pripraviť a vytvoriť pozadie (komputačnú logiku). To umožní používateľovi čo najefektívnejšie analyzovať údaje a následne podľa potrieb tieto dáta modifikovať, poprípade si zobraziť iné údaje, ktoré súvisia so zvolenou dátovou jednotkou v aktuálne zobrazenej množine údajov.

Tento proces prezentácie dát vo virtuálnej realite je jednoduchý ak je všetko potrebné pripravené a cieľom je skúmanie dát v scéne virtuálnej reality. Častokrát dáta nereprezentujú jednoduché dátové štruktúry a je nutné si ich čiastočne pripraviť pred samotnou vizualizáciou, čo v prípade špecifických oblastí výskumu nie je triviálna vec. Príprava takýchto dátových pohľadov často pozostáva zo zložitých úloh, ktoré vyžadujú vysoký výpočtový výkon a je potrebné tieto dáta získať zo vzdialených výpočtových centier, ktoré disponujú potrebnými zdrojmi. Vyžadujú sa pre tento účel od výskumníkov a vedcov pokročilé schopnosti v oblasti IT, aby si vedeli dátové štruktúry pripraviť ale rovnako aj vykonať potrebné úpravy, čo môže byť pomerne náročná a zdĺhavá vec.

Teda virtuálna realita má svoje výhody v prezentácií dát, ktoré majú v pozadí nejakú logiku, ktorá umožní potom vizualizované dáta prezentovať, meniť ich formu poprípade meniť ich charakter podľa potrieb používateľa. Jej sila spočíva predovšetkým v podávaní informácií používateľovi ale rozhodne nie vo výpočtoch nad dátami. Umožní tak vykonávať dôležité zobrazenia dát na základe logiky v pozadí na požiadavku používateľa. Avšak proces prípravy dát, ktoré budú neskôr vizualizované vyžaduje špeciálne prostriedky pre danú doménu výskumov, ako sú rôzne knižnice, softvérové rámce a monitorovacie aplikácie, ktoré generujú, monitorujú a následne posielajú výsledky späť používateľom.

Táto práca sa bude zameriavať na prípravu takéhoto prostredia pre doménu

experimentov v oblasti fyziky vysokých energií, kde sú dátové analýzy vizualizované pomocou histogramov, ktoré majú charakter definície dátových oblastí (biny histogramu) a následne prebieha selekcia podoblasti (binu) a spúšťa sa pre ňu výpočet. Ako výsledok výpočtu vznikne dátová projekcia pre zvolenú dátovú podoblasť (bin). V tejto doméne experimentov je bežné generovanie obrovského množstva dát a na ich spracovanie sú použité zoskupenia výpočtových jednotiek (klástre), kde sa tieto dáta generujú, ako výsledok úloh.

V spojení s vizualizáciou vo virtuálnej realite, keďže VR má veľký prezentačný potenciál a kláster zas disponuje vysokým výpočtovým výkonom, prichádza otázka spojenia týchto 2 oblastí a vytvorenie vizualizačného prostredia, ktoré využije svoj prezentačný potenciál pre vizualizáciu dát a spracovanie interakcií od používateľa, aby sa následne výpočty nad dátami podľa definovanej interakcie vykonali na výkonnom klástri, ktorý následne distribuuje výsledky späť do prostredia za účelom vizualizácie. Tento príklad popisuje možný scenár pre návrh systému určeného pre danú oblasť výskumov.

Formulácia úlohy

Úloha pozostáva z analýzy vizualizácií dátových analýz vo virtuálnej realite na základe ktorých sa bude postupovať v návrhu vhodného prostredia pre dátovú analýzu v oblasti fyziky vysokých energií. Pre návrh bude nutné analyzovať už existujúcu aplikáciu pre monitorovanie úloh na dávkovacom systéme SALSA [1], čo bude predstavovať hlavný komunikačný prvok a takisto aj knižnice NDMVR [2], ktorá predstavuje už existujúce komponenty pre vizualizáciu histogramov vo virtuálnej realite.

Hlavným cieľom práce je na základe nadobudnutých poznatkov z analýzy vhodne upraviť a prispôsobiť knižnice NDMVR [2], REACT-NDMSPC-CORE [3] a REACT-NDMSPC [4] pre použitie na účely vizualizácie a potom aj samotný návrh prostredia pre definíciu logiky vizualizácie. Logika vizualizácie, by sa mala definovať v prostredí webovej aplikácie bez použitia virtuálnej reality a mala by poskytovať možnosť konfigurovať histogram pre spúšťanie výpočtov na klástri, možnosť spúšťania výpočtov vo forme úloh a adekvátnu vizualizáciu priebehu výpočtov týchto úloh na klástri. Samotná vizualizácia by mala obsahovať, tak zobrazenie histogramov vo virtuálnej realite pomocou knižnice NDMVR, ako aj bežné zobrazenie pomocou knižnice JSROOT [5] v prostredí webovej aplikácie s možnosťou voľby požadovanej vizualizácie. Navrhnuté riešenie implementovať a vhodne otestovať.

1 Analytická časť

V tejto kapitole sme sa zamerali na získavanie všetkých dôležitých poznatkov, ktoré sú nevyhnutné pre správny návrh prostredia pre definovanie dátovej analýzy a následné vizualizácie.

Na začiatku kapitoly sme sa pozreli na existujúce vizualizácie s potenciálne podobným účelom, ktorý by mal predstavovať náš systém. Snažili sme sa zistiť ich výhody, implementačné riešenia a postupy pre ich tvorbu.

V nasledujúcich častiach sme analyzovali už samotný dávkovací systém SALSA, ktorý bude nevyhnutný pre spúšťanie úloh na klástri, monitorovanie stavu a distribúciu dát. Pre tento účel už existuje aplikácia, ktorá toto pokrýva na UPJŠ, kde ju aktívne využívajú na monitorovacie účely. Aplikácia bola vyvinutá a podrobne zdokumentovaná v dokumentácii [6].

Na záver sme analyzovali aj existujúcu knižnicu pre vizualizáciu histogramov vo virtuálnej realite NDMVR postavenú na softvérovom rámci A-Frame [7] a JS-ROOT [8]. Keďže našou úlohou bolo predovšetkým návrhnúť možné vylepšenia a prispôsobenia tejto knižnice pre využitie v našom navrhovanom systéme na vizualizácie.

1.1 Vizualizácia dátových analýz vo virtuálnej realite

V úvode je dôležité zamyslieť sa predovšetkým nad účelom samotnej vizualizácie pomocou určitých technológií, platforiem a určiť ich silné a slabé stránky. Preto sme sa zamerali v tejto kapitole na analýzu rôznych metód vizualizácie experimentálnych dát vo virtuálnej realite, aby sme zistili výhody týchto vizualizácií a aké požiadavky musia spĺňať. Takisto aby sme nadobudli poznatky z tejto oblasti, ktoré by sme neskôr vedeli použiť pri návrhu a implementácií riešení.

1.1.1 Vizualizácia dát, fyzická a virtuálna

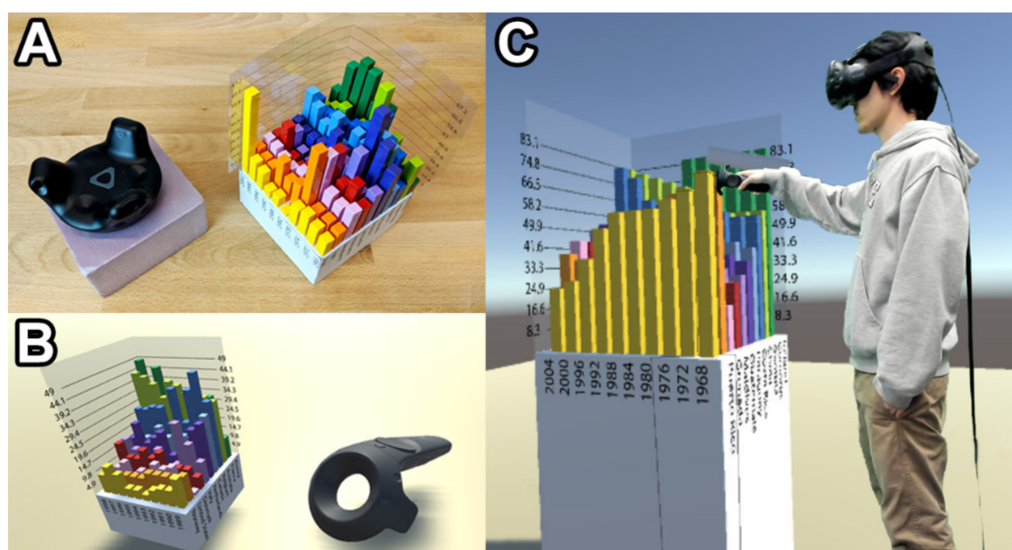
Klasická vizualizácia dát vo webovom prostredí bola do nedávnej doby hlavným trendom prezentácií dát hlavne pre jej jednoduchosť a dostupnosť širokej škály

nástrojov pre jej rýchlu a efektívnu tvorbu. V dnešnej dobe je však viac spájaná aj s inými formami prezentácie, ktoré nie sú až také bežné a sú náročnejšie na implementáciu. Medzi tieto formy patrí aj fyzická vizualizácia, ktorá prináša odlišný pohľad na vizualizované dáta a zdá sa, že má pomerne veľa spoločného aj s rozšírenou realitou.

Na tento problém sa zamerali autori z *university of Calgary* v publikácii [9], ktorí sa zamerali práve na toto porovnanie správania účastníkov pri analýzach využitím fyzických nástrojov a virtuálnych nástrojov. Správanie účastníkov bolo otestované na odsledovaní a meraní času, a to na rôznych úlohách s použitím rôznych nástrojov a na rôzných vizualizačných dátach.

Medzi jeden experiment vykonaný v publikácií [9] patrí aj vykonanie základných operácií:

- určenie rozsahu elementu
- zoradenie elementov
- porovnanie elementu

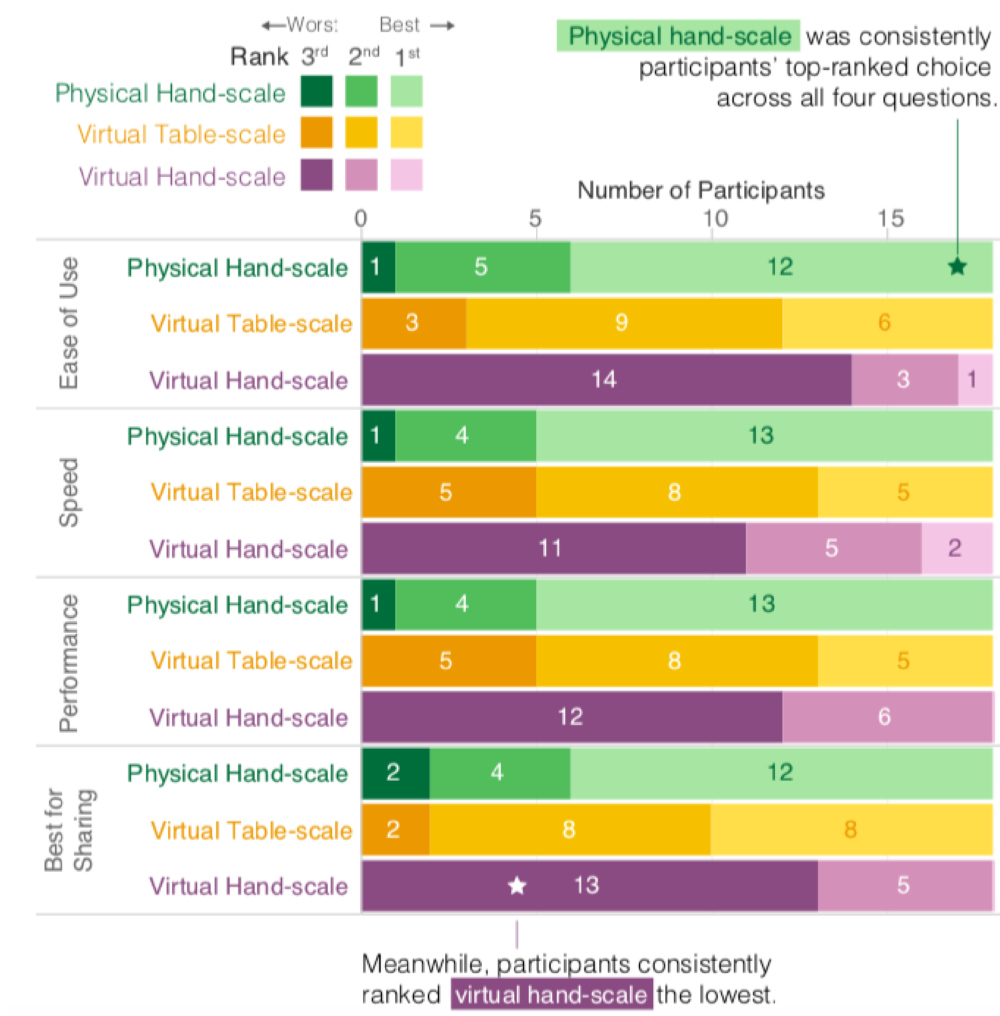


Obr. 1.1: Vizualizácie pre aplikáciu testovania interakcií. (Zdroj: [9])

Operácie boli aplikované nad dátovými vizualizáciami, zobrazenými na obr. 1.1, kde išlo o:

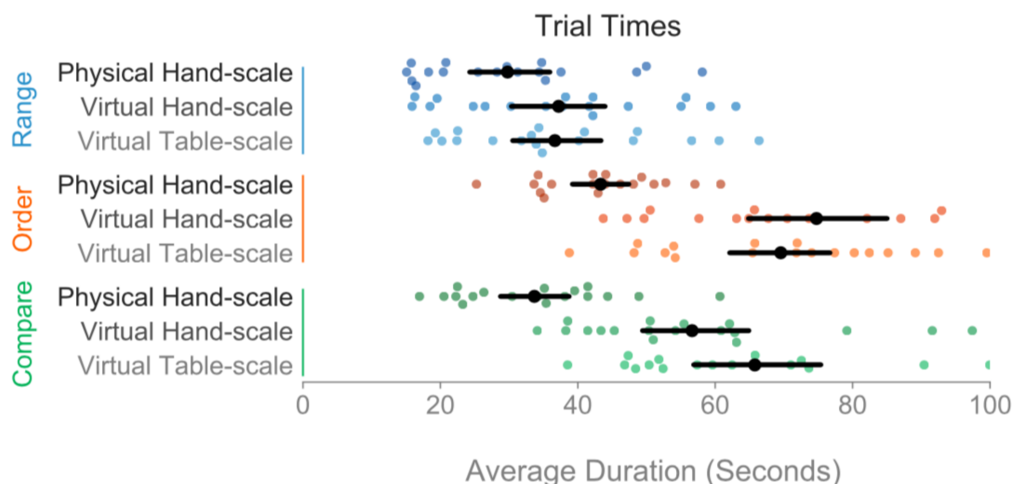
- A - fyzickú vizualizáciu histogramu
- B - virtuálnu vizualizáciu histogramu pripevnenú na ovládač používateľa

- C - virtuálnu statickú vizualizáciu, kde je histogram situovaný na určitom mieste v priestore (Tento typ vizualizácie je podobný aj v NDMVR)



Obr. 1.2: Výsledky a porovnanie spätnej väzby účastníkov v oblasti jednoduchosti, rýchlosti, výkonu a zdieľania (Zdroj: [9])

Výsledky z používateľského hľadiska boli založené na ich pocitoch z interakcií, schopnosti rýchlo vykonať danú operáciu a zážitku v danom prostredí. Ako je znázornené na obrázku 1.2, tak virtuálna vizualizácia s pripevneným histogramom na ovládač (vizualizácia - B) získala najslabšie hodnotenie z hľadiska celkového počtu účastníkov a to vo všetkých meraných oblastiach. Ako najlepšia, najrýchlejšia a najjednoduchšia na použitie, pripadala väčšine účastníkom práve fyzická. Virtuálna statická vizualizácia získala celkovo solídne hodnotenie porovnaní s ostatnými vizualizáciami.



Obr. 1.3: Výsledky a porovnanie potrebného času na vykonanie operácií. (Zdroj: [9])

Na nasledujúcom obrázku 1.3 je zobrazenie porovnaní času potrebného na vykonanie operácií nad príslušnou vizualizáciou. V prípade operácie určenia rozsahu ide o pomerne vyrovnané trvania vo všetkých vizualizáciách, kde pri ostatných operáciách už je zobrazená výhoda fyzickej vizualizácie a väčšia časová priepasť. Zaujímavým faktom je aj súperenie oboch virtuálnych vizualizácií pri operáciách triedenia a porovnávania elementov.

Zhrnutie tohto experimentu ale aj ostatných experimentov, ktoré boli predmetom publikácie [9] je u autorov hlavne zistenie sľubného potenciálu statických virtuálnych vizualizácií, hlavne v analýzach grafov a tabuľkových vizualizácií, ktoré vytvárajú rovnováhu medzi čítateľnosťou a dostupnosťou. To podľa autorov umožní používateľom skúmanie a jednoduchú manipuláciu s vizualizovanými dátami. Rozdiel medzi fyzickými vizualizáciami a virtuálnymi je prevažne v súčasnom stave VR nástrojov a nedostatku hmatateľností, ktoré absentujú na virtuálnych modeloch. To podľa autorov spôsobilo lepší výkon účastníkov s fyzickou vizualizáciou.

1.1.2 Vizualizácia vedeckých dát vo VR

V rôznych zdrojoch je definovaná vizualizácia vedeckých dát ako obrovská dátová štruktúra zložená z dát, ktoré sú medzi sebou prepájané rôznymi vzťahmi.

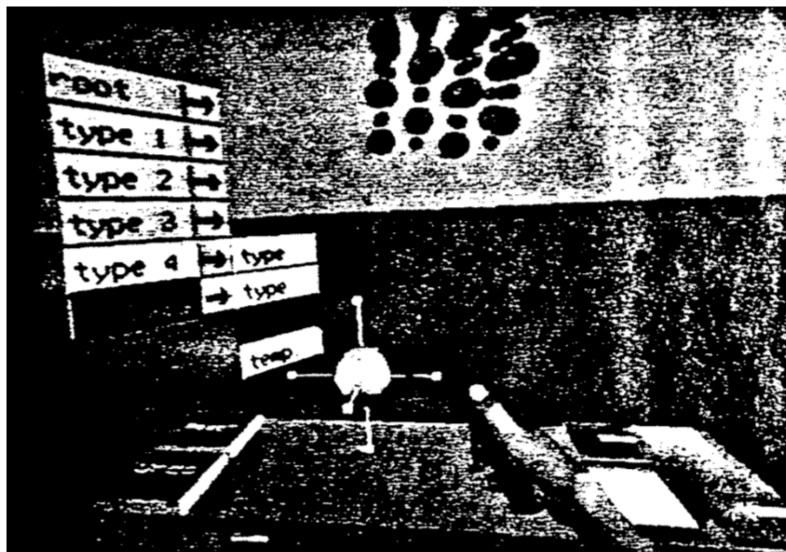
Napríklad podľa Van Dama [10] vedecká vizualizácia nie je samoúčelná ale je súčasťou mnohých úloh, ktoré zvyčajne zahŕňajú určitú kombináciu interpretácie a manipulácie s vedeckými údajmi a modelmi. Na uľahčenie porozumenia vedci vizualizujú údaje, aby hľadali vzory, vlastnosti, vzťahy a anomálie. Vizualizácia by sa mala chápať skôr ako *"task driven"* (riadená úlohami) než ako *"data driven"*

(riadená dátami). Následne sa autor vyjadruje aj k téme vizualizácie viacrozmerných dát, kde vzniká problém s koreláciou dát medzi viacerými hodnotami. Náš vizualizačný systém je schopný rozpoznávať rôzne vzory a anomálie v dátach, čo v prípade bežných zobrazení dát v 2D prostrediach nie je možné.

Podľa McCormicka [11], vedecká vizualizácia je použitie počítačovej grafiky na vytváranie a generovanie vizuálnych obrazov, ktoré pomáhajú porozumieť zložitým, častokrát až masívnym numerickým reprezentáciám vedeckých konceptov alebo výsledkov.

Dataset môžeme považovať za dátovú štruktúru. Vo väčšine sfér naberá komplexitu a z toho vzniká problém vo vizualizácií dát použitím bežných dostupných prostriedkov alebo ak sa dáta vizualizujú, tak klesá prehľadnosť pri analýzach čo má nepriamy vplyv na efektivitu výskumu. V mnohých prípadoch sa dáta vyznačujú určitým rozmerom dát, kde v prípade 2 alebo 3 rozmerov sú dáta pomerne dobre vizualizovateľné pomocou bežných prostriedkov, kde v prípade 4, 5 a viac rozmerných dátových štruktúrach už nastáva problém so zobrazením dát bez toho, aby bolo nutné vytvoriť komplikované pohľady na údaje, kde je nutné patrične interagovať len za účelom manažmentu vizualizácie.

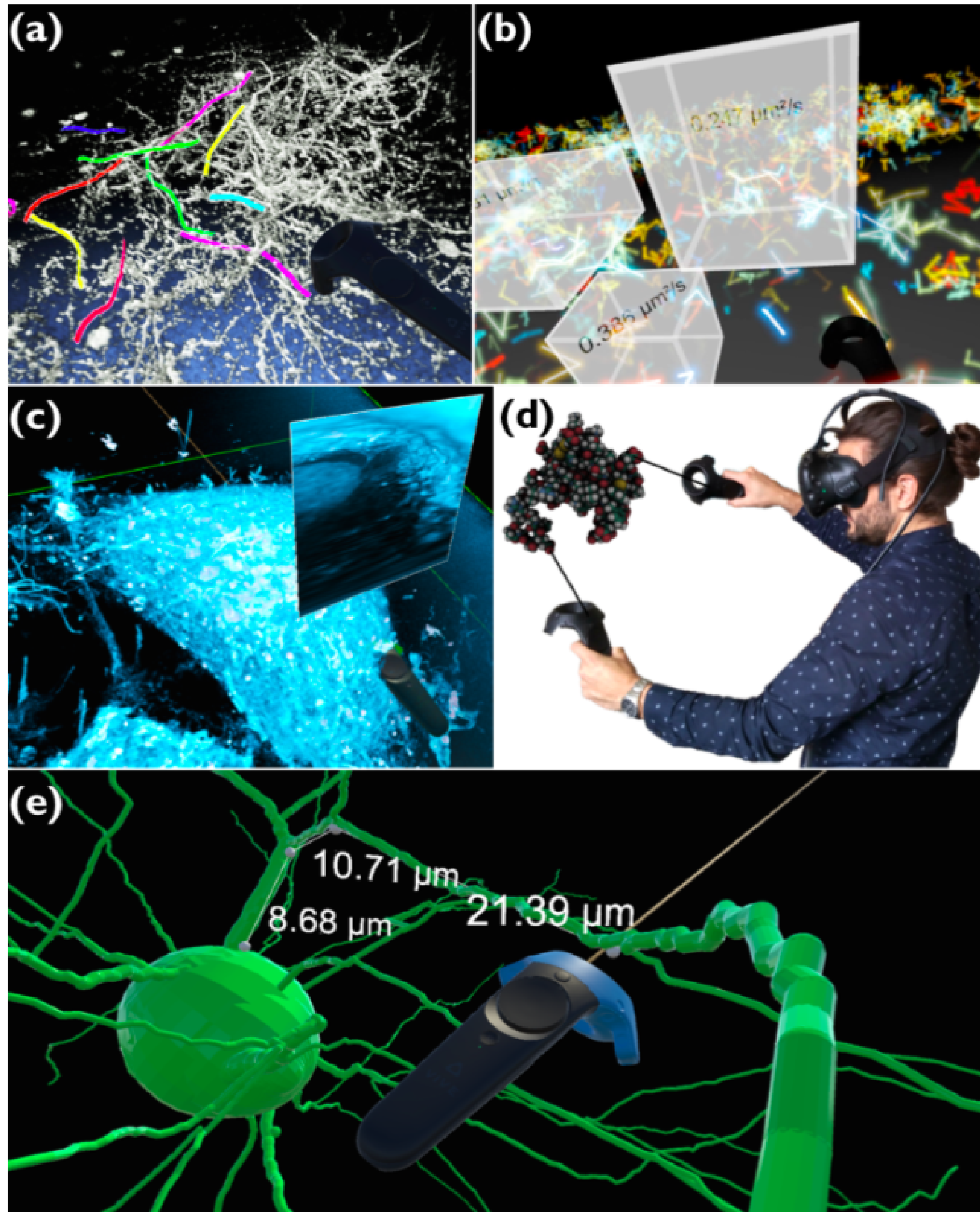
Tento problém je opísaný aj v článku [12], kde je zvýraznený účel virtuálnej reality pri analýze viacrozmerných dátových štruktúr založený na glyfoch. V tomto článku je popísaný princíp vizualizácie dát obrovských rozmerov. Virtuálne prostredie má za úlohu poskytnúť používateľovi možnosť si naviazať dátové zložky na objekty (tzv. glyfy) priamo v prostredí bez toho aby bolo nutné zasahovať do kódu. Týmto princípom si používateľ vie vytvoriť vlastné prostredie, ktoré poskytuje širokú škálu objektov pre naviazanie dát. Objekty predstavujú glyfy, ktorých atribúty, (ako sú pozícia, tvar, orientácia, veľkosť, farba a pod.) je možné viazať na dáta. Používateľ následne používa grafické používateľské rozhranie na viazanie týchto dát a vytváranie svojho prostredia.



Obr. 1.4: Ukážka grafického používateľského rozhrania vo virtuálnej realite pre viazanie dát a objektov. (Zdroj: [12])

Na obrázku 1.5 je ukážka rozhrania pre viazanie dát a objektov. Používateľ po naviazaní dát následne analyzuje dáta a vie si meniť atribúty vizualizácie, ako napríklad škálu zobrazovaných dát alebo rozsahy a ofsety, kde pri zmene sa v definovanom prostredí automaticky menia naviazané atribúty objektov podľa aktuálnych dát.

V článku [13] zaoberajúcom sa vizualizáciami vo vedeckom prostredí je opäť VR považovaná ako "most"medzi vedeckými aplikáciami vďaka svojím možnostiam vizualizácie dát a navigácií v komplexných 3 rozmerných dátových štruktúrach. Autori poznamenávajú, že otázka zlučovania interakcií s dátami, používateľmi, ako aj algoritmy pre umelú inteligenciu je stále vo vývoji a predstavujú budúcnosť v tomto smerovaní. Medzi hlavnú výhodu vizualizácie dát vo VR autori považujú hlavne voľnosť a slobodu používateľa v skúmaní prostredia. Pri použití moderných okuliarov a ovládačov je možné dosiahnuť milimetrovú precíznosť. Používateľ nadobúda prirodzený pocit v prostredí a je schopný rýchlo vykonávať komplikované úlohy v porovnaní s použitím myši a monitora. Jednoduché pohyby v rámci scény umožňujú detegovať vzory záujmu a prirodzene podporujú zvedavosť a chuť objavovať a skúmať, čo je vo vedeckých sférach potrebné.



Obr. 1.5: Príklad vizualizácií experimentálnych vedeckých údajov. (Zdroj: [13])

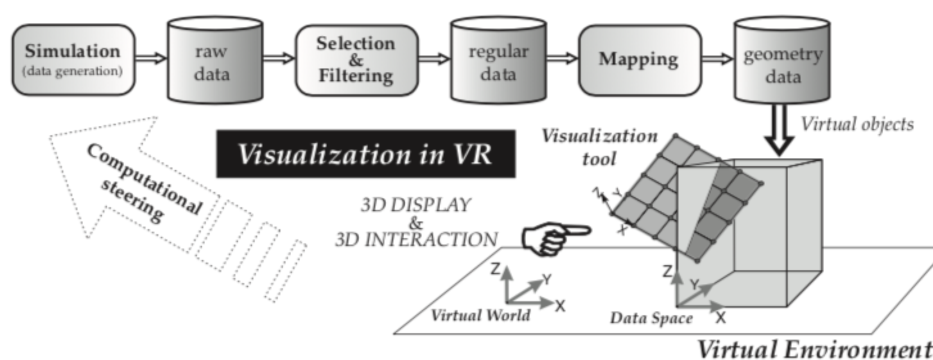
Na obrázku 1.5 sú zobrazené príklady vizualizácií vedeckých údajov:

- **a** - je zobrazená segmentovaná časť neurónov čuchového systému myši
- **b** - je zobrazená Bayesovská inferencia lokálnej difuzity. Používateľ identifikuje oblasť záujmu v 3D priestore a použije ovládač na definovanie potrebných limitov a následne spustí program, ktorý počíta difúziu pomocou Bayesovho prístupu. Výsledky sa potom prekryjú.
- **c** - zobrazuje projekčný nástroj zarovnaný v ovládači VR, používa sa na mapovanie vnútrajška bunky z elektrónovej mikroskopie.

- **d** - zobrazuje interakciu používateľa s makromolekulou.
- **e** - zobrazuje meranie segmentov axónov pomocou ovládača VR v databáze myších neurónov.

1.1.3 Vizualizácie a vizualizačné prostredia VR

Podrobnejšie zhodnotenie možností virtuálnych prostredí pre výskum a ich návrh zahŕňujúci vizualizácie ale aj interakcie, rieši autor vo svojej práci [14], kde sa okrem analýzy existujúcich vizualizácií venuje aj ich použiteľnosti, samotnému návrhu a vývoju.



Obr. 1.6: Zreťazenie vizualizačných aspektov integrovaných s responzívnym virtuálnym prostredím (Zdroj: [14])

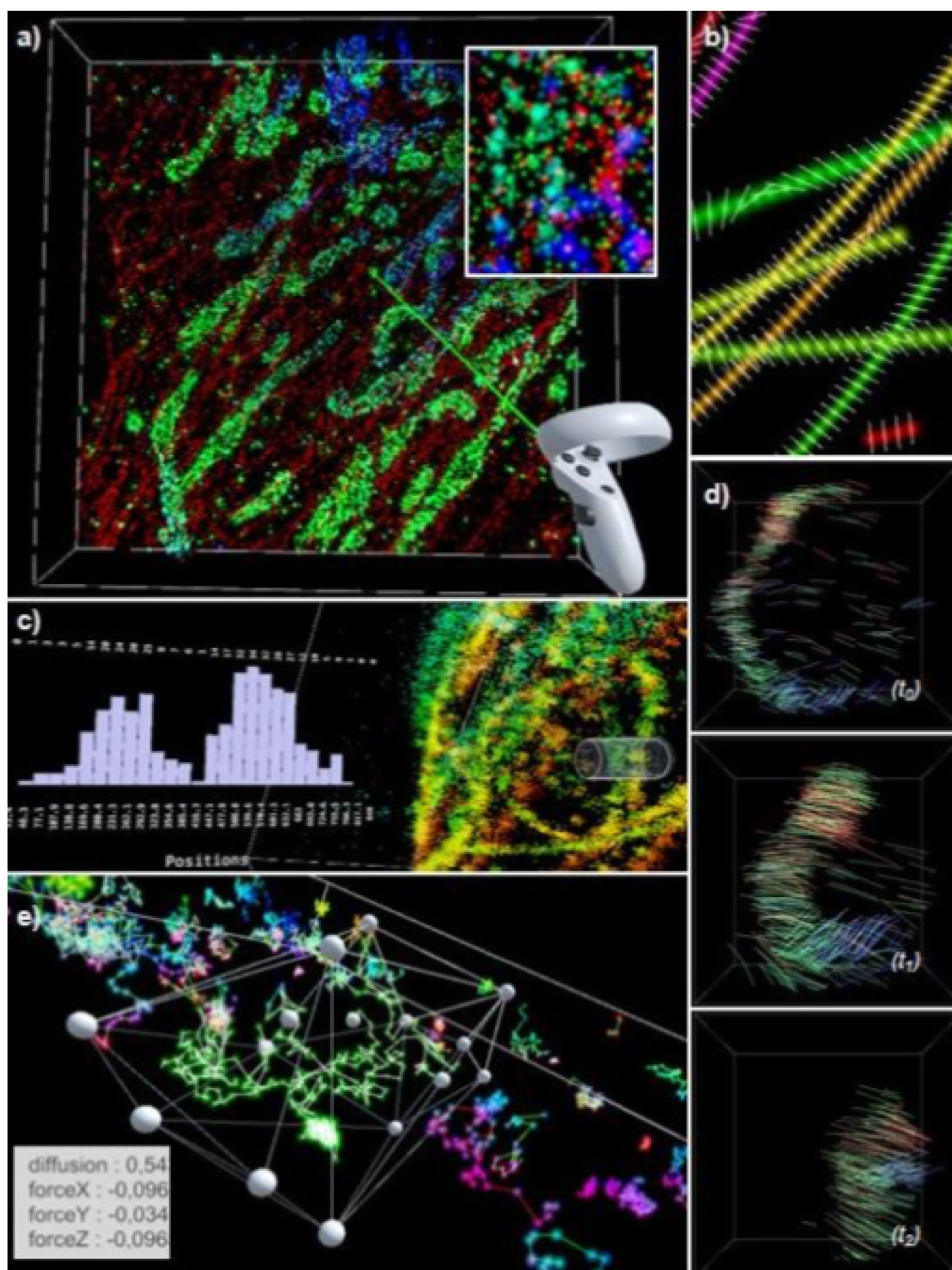
Autor popisuje vizualizáciu, ako proces sústavnej simulácie dát, kde sa ako výstupy generujú *surové* dáta, ktoré sa v neskorších fázach selektujú, filtrujú a mapujú do príslušných vizualizačných objektov v prostredí virtuálnej reality. Fázy komputácie a simulácie sú riadené interakciami v prostredí (v diagrame 1.6 nazvané **riadenie komputácie**) a tento proces pozostáva z neustálej simulácie, vytvorenia aktuálneho náhľadu údajov a spracovania interakcie, čo odznova spúšťa celú sekvenciu 1.6 výpočtu.

Autor ďalej popisuje aj nedostatky pri týchto typoch vizualizácií, kde nastávajú problémy pri zvyšovaní dimenzií zobrazovaných dát, kde interakcie pri nižších rozsahoch môžu byť efektívne a optimálne ale pri vyšších rozsahoch už to nemusí platiť. V tomto prípade je nutné sústrediť sa na prepracovanie riadenia simulácií, komputácií a prepracovať vhodne interakčné nástroje vo vizualizačnom prostredí.

1.1.4 Genuage

Genuage je voľne dostupná softvérová platforma určená pre analýzu viacrozmerných dát. Softvérová platforma je definovaná v článku [15] ako nástroj, ktorý umožňuje percepciu, interakciu a analýzu komplexných viacrozmerných dátových množín. Vzhľadom na rastúcu komplexitu dát, stúpajúci a rastúci koncern dátových prezentácií a interpretácií v počiatočných fázach výskumu. Najčastejšie je softvér používaný v spojení s lokalizáciami molekúl, cytometriou a astrofizikou. Softvér obsahuje duálne vizualizačné rozhranie, desktop mód pre načítanie dátovej vizualizácie spolu s vyhodnotením výsledkov na obrazovke počítača a VR mód pre efektívne dátové vizualizácie, interakcie a analýzy.

Hlavnou vlastnosťou softvéru je načítanie a prezentácia miliónov dátových bodov. Body predstavujú viacrozmernú informáciu, ako napríklad molekulárnu orientáciu, trajektóriu, rýchlosť častíc, farebný kód. Ďalšou vlastnosťou sú množiny nástrojov špecifických pre prostredie VR, ako napríklad merania vzdialeností, uhlov, hustoty, profily histogramu. Medzi výhodu patrí aj možnosť čítania dát z externých zdrojov, napríklad z JSON súboru a prístup k metadátam. Genuage je kompatibilný aj s inými softvérmi pre dátové analýzy, ako sú python [16] a MATLAB [17], pre potrebu výmeny dátových množin po modifikácií alebo pre prípad ďalších analýz. Ako prídavok softvér ponúka aj Bayesov analytický balík derivovaný zo softvéru inferenceMAP [18] pre živé analýzy dynamických dát vo virtuálnej realite.



Obr. 1.7: Príklad vizualizácie použitím softvéru genuage (Zdroj: [15])

Na obrázku 1.7 sú znázornené ukážky rozhrania softvéru genuage vo VR. Na ukážkach sú zobrazené viaceré vizualizácie pomocou rozličných nástrojov.

Genuage je na základe dokumentácie [19] možné považovať za natívnu aplikáciu, ktorá je založená na engine unity [1], kde preložený program vieme spustiť ako súbor s príponou **exe**. Pre použitie je nutné zabezpečiť splnenie systémových požiadaviek, ktoré nie sú zanedbateľné.

Predstavené riešenie predstavuje zaujímavý spôsob pre analýzu viacrozmer-

[1] `https://unity.com/`

ných dát, ktoré sú skôr určené pre dáta s odlišným charakterom ako v prípade JSROOT. Aj napriek tomu, že analyzovaný softvér genuage podporuje načítanie rozsiahlych dátových množín z JSON súborov ako hlavná nevýhoda pre použitie technológie pre naše potreby je sústredenosť na platformu windows. Samotný softvér nepodporuje až toľko veľa zariadení pre VR ako A-Frame a nevýhoda je aj absencia akéhokoľvek potenciálu a záruky, že bude možné aj zdieľanie scény a interakcií pre viacerých používateľov. Taktiež následná snaha o prepojenie softvéru s webom nie je možná vzhľadom na požiadavky nášho systému.

1.1.5 Zhrnutie dátových analýz vo VR

Ako bolo písané v tejto sekcii na svete existujú reprezentatívne riešenia vizualizácií vedeckých dát v prostrediach virtuálnej reality. Bolo dokázané, že virtuálne prostredie má veľký potenciál z hľadiska ponúkaných možností dátových vizualizácií ale aj interakcií s dátami. Pre naplnenie tohto potenciálu vedie zložitý proces návrhu údajových množín a prostredí pre vizualizáciu, čo obsahuje ďalšie výzvy a je potrebná kolaborácia s expertmi v danej doméne analýz pre návrh dostatočne kvalitného prostredia. Táto oblasť návrhu kvalitného prostredia je priam kritická pre úspešnosť danej vizualizácie v cieľovej doméne výskumu.

Hlavným cieľom v tomto prípade je po zhodnotení cieľovej domény vhodne popísať prostriedky pre dátové komputácie v danej doméne, aby následne na to mohli byť pripravené požadované interakčné nástroje pre používateľa. Následne je určenie vhodnej vizualizačnej metódy závislej na charaktere zobrazovaných údajov a vyladenie prostredia, pretože v pohĺcujúcej virtuálnej realite je dvojnásobne dôležité, aby používateľ nadobudol príjemný pocit. Ako bolo spomenuté v časti týkajúcej sa porovnania fyzických analýz s virtuálnymi, je na mieste pristupovať k vývoju vizualizácie so zameraním sa, na vymedzenie všetkých zložitých interakcií s vizualizovanými objektami a poskytnúť používateľom iba primitívne interakcie, kde bude efektivita a rýchlosť vykonania na rovnakej úrovni. Zložité interakcie by mali byť vhodne prepočítavané a výsledný náhľad dát po týchto interakciách by mal byť zobrazený používateľovi vo forme rôznych projekcií. V tomto prípade VR poskytuje radu výhod oproti reálnemu svetu.

Vzhľadom na to, že VR nie je až tak rozšírená a stále existuje veľa predsudkov v spojení s touto metódou dátovej analýzy. Ako sa ukázalo aj pri experimente 1.2, tak sa našli aj používatelia, ktorí ohodnotili vizualizácie vo VR najvyššou známkou, aj keď ich bolo podstatne menej ako v prípade fyzických vizualizácií. Z tohto dôvodu je dôležité zaoberať sa vývojom riešenia vizualizácie vo VR ale ponechať aj klasické vizualizácie v 2D prostrediach na weboch, na ktoré sú používatelia

zvyknutí a to špeciálne v našej doméne.

1.2 Analýza Knižnice NDMVR a JSROOT

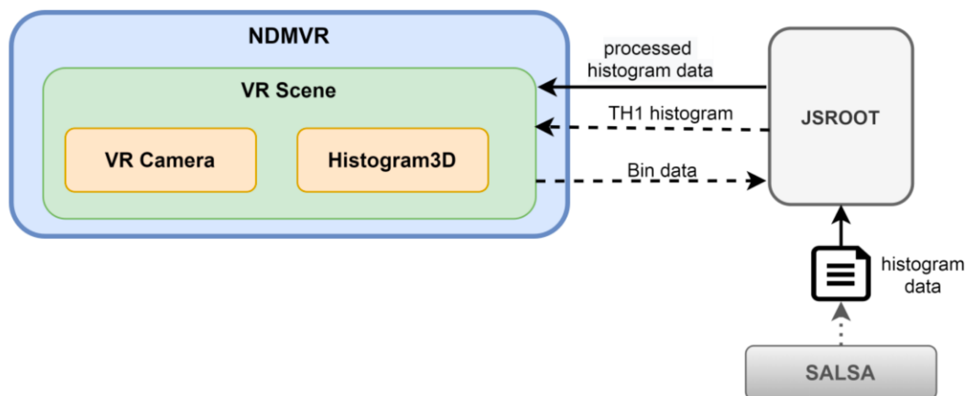
V tejto kapitole sme sa pozreli detailne na balík NDMVR a taktiež aj na knižnicu JSROOT [8], ktorá je vyvíjaná vývojármi CERN-u [2] [20] a GSI inštitútu [3] na účely vizualizácie dát na webe pomocou *Three.js* [21]. Hlavne jej najnovšie vylepšenia budú predmetom tejto analýzy z dôvodu použitia niektorých funkcií na vytváranie histogramov pre účely tvorby a vizualizácie dátových analýz.

1.2.1 NDMVR

Ide o knižnicu určenú na vizualizáciu histogramov TH2 a TH3 vo virtuálnej realite. Knižnica je podrobne popísaná v konferenčnom príspevku [22] a takisto aj v bakalárskej práci [23]. Knižnica je založená na technológiách A-Frame, čo je softvérový rámec pre virtuálnu realitu na webe a *RxJS* [24] použitý na vytváranie komunikačných kanálov medzi komponentami. Celý tento koncept ďalej využíva na potrebné obmieňanie a manažment zobrazovaných entít v DOM knižnicu *React* [25] alebo už existujúcu knižnicu vyvinutú priamo s integráciou A-Frame [7]. Ide o knižnicu React-Aframe [26] knižnica však nebola použitá pri vývoji knižnice NDMVR. NDMVR obsahuje komponent **NdmVrScene**, ktorý zobrazí v štruktúre DOM vnorenú VR scénu a následne v nej vizualizuje histogram, ktorý sa poskytne, ako argument vo forme JSROOT [8] objektu. JSROOT objekty môžeme definovať, ako štruktúry obsahujúce potrebné dáta o entitách binov, ich obsahy, označenia, ktoré sú reprezentované vnorenými objektami s dátami popisujúcimi osi, označenia a v neposlednom rade aj funkcie pre získavanie samotných binov ale aj ich modifikáciu.

[2] `https://home.cern/about`

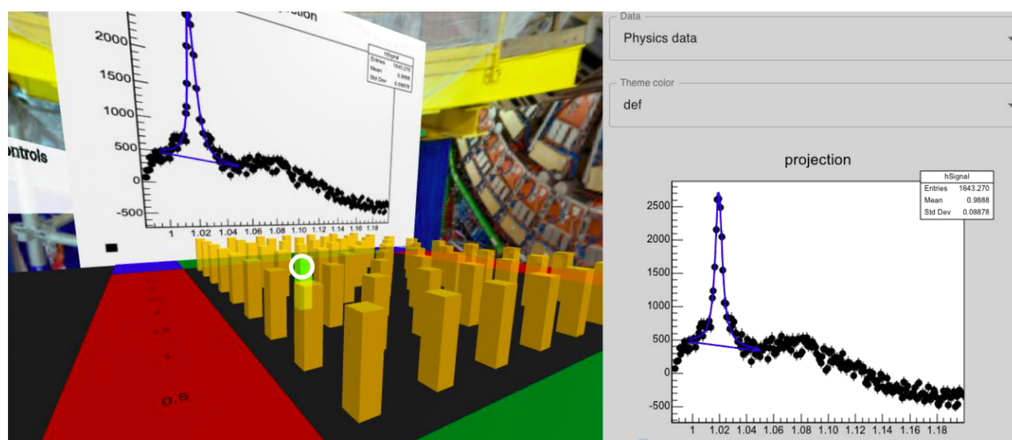
[3] `https://www.gsi.de/en/start/news`



Obr. 1.8: Softvérová architektúra knižnice NDMVR (Zdroj: [22])

Na obrázku 1.8 je zobrazená architektúra knižnice vo vzťahu s knižnicou JSROOT [8] a dávkovacím systémom SALSA [1]. NDMVR obsahuje komponent, ktorý definuje VR scénu, jej vlastnosti a atribúty. Samotný histogram je generovaný komponentom **histogram3D**, ktorý vykresľuje histogram, ako entitu v scéne. Pre interakciu je definovaný aj komponent **VRCamera**, ktorý predstavuje používateľovu kameru s nástrojmi, ktoré sa týkajú predovšetkým ovládania a nie samotného histogramu. Môžeme povedať, že ide o 2 nezávislé komponenty knižnice, ktoré zaobaľuje scéna. Knižnica zobrazuje histogramy na základe objektov, ktoré prijíma ako parametre.

V 1.8 je aj zobrazený koncept použitia, kde knižnica pre správnu funkciu vyžaduje objekty histogramov, ktoré je možné vytvoriť funkciami zahrnutými v JSROOT-e alebo ROOT-e. Preto sa pri importe tejto knižnice v klientskej aplikácii vyžaduje aj použitie knižnice JSROOT. Na základe spomenutej architektúry môžeme namodelovať názornú ukážku situácie, kedy v klientskej aplikácii je použitý JSROOT na otváranie súborov s vytvorenými histogramami, ktoré sme získali ako výsledok úlohy, ktorá sa vykonala na klástri (to nám zabezpečí SALSA). Následne sa tieto histogramy vizualizujú pomocou NDMVR.



Obr. 1.9: Názorná ukážka zobrazenia vizualizácie histogramu TH2 pomocou NDMVR (Zdroj: [22])

Na obrázku 1.9 je znázornená vizualizácia histogramu použitím komponentov **ndmVrScene** a **jsrootHistogram**, kde vidíme v ľavej sekcii VR scénu s histogramom doplnenú o používateľský panel s vizualizovaným TH1 histogramom v klasickom webovom prostredí. Po kliku na bin v histograme zobrazenom vo VR sa nám zobrazí projekcia histogramu pre označený bin v ľavom paneli.

Ako je popísané v bakalárskej práci [23] celý tento proces komunikácie je založený na komunikačnom kanále implementovanom použitím **subjektu** knižnice *RxJs* [24] kde sa klikom na bin publikujú potrebné dáta a na druhom konci spojenia komponent čaká tieto dáta a následne zobrazí histogram v bočnom paneli (ľavý panel). Finálny komponent zobrazujúci projekciu na ľavom paneli následne vytvorí aktuálny pohľad histogramu vo formáte PNG a nastaví ho ako textúru entite predstavujúcej tabuľu vo VR scéne. Prístup k entite je zabezpečený pomocou definovaného identifikátora pre entitu v scéne.

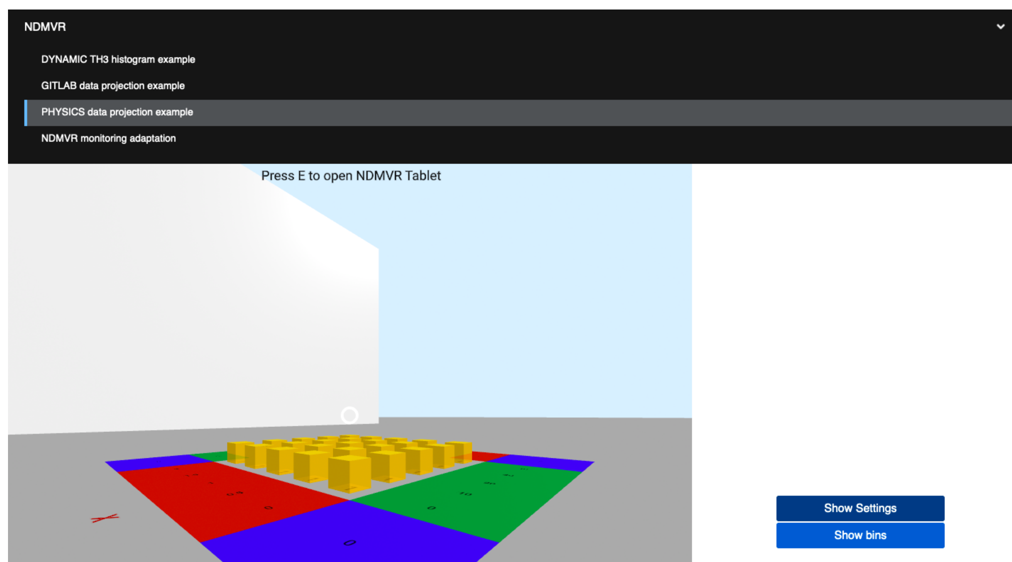
1.2.2 Analýza nedostatkov NDMVR

Na použitie NDMVR pre účel vizualizácie v tejto práci je potrebné otestovať tento komponent a zistiť jeho aktuálny stav vzhľadom na aktuálne požiadavky. Komponent vizualizuje histogram na základe objektu z parametra a neobsahuje žiadnu zložitú logiku vo svojej štruktúre. Obsahuje len logiku týkajúcu sa komunikácie medzi komponentami a logiku pre samotnú interaktivitu používateľa s histogramom a binmi. Teda NDMVR je vyvinutý za účelom univerzálneho zobrazovania vizualizácií založených na objektoch, ktoré si môžeme vytvárať v klientskej aplikácii a následne ich posielať do komponentu.

Keďže sa zameriavame na implementáciu logiky pre vizualizáciu, budeme potrebovať často vytvárať viaceré histogramy vo webovom prostredí alebo na strane

rôznych vzdialených serverov, spracovávať tieto dáta a zobrazovať v určitom časovom intervale. Preto sme sa rozhodli vykonať experiment aby sme zistili či NDMVR spĺňa všetky požiadavky pre takýto scenár použitia. Experiment spočíval vo vytvorení testovacieho prostredia, ktoré obsahuje viacero typov histogramov ktoré:

- mali rozdielne rozsahy - cieľom bolo zistiť schopnosť meniť vizualizácie s rôznym počtom binov na osiach.
- mali rozdielne typy hodnôt - cieľom bolo zistiť flexibilitu zobrazovania označení osí pri rozdielnych škálach (nie iba jednoduché číslovania typu integer)
- rozsahy obsahov binov - cieľom bolo zistiť schopnosť vizualizácie reagovať na extrémne rozsahy hodnôt binov
- schopnosť zobrazovať korektne histogramy v určitom definovanom intervale tak, aby bolo evidentné vidieť zmeny medzi vizualizáciami
- zachovanie rovnakých nastavení pre zobrazovanú sekciu pri zmene vizualizovaného histogramu



Obr. 1.10: Experimentálne prostredie NDMVR

Experimentálne prostredie [4] 1.10 obsahovalo viacero rozličných histogramov, ktoré boli vizualizované. V druhej fáze sa zisťovala schopnosť komponentu reagovať na zmenu histogramu, kde nám generovanie náhodného histogramu zabezpečilo tento jav, ktorý budeme používať neskôr v prípade monitorovania, kde budeme pri reakcii na stav musieť aktualizovať vizualizáciu.

[4] `https://ndmspc.gitlab.io/ndmvr/`

Ako výsledky experimentu sme odhalili nedostatky, ktoré bude potrebné vyriešiť pred použitím komponentu na požadované účely.

Medzi najkritickejšie patria:

- neplynulosť vizualizácie - pri obmene sekcie sa tieto nastavenia pri zmene vizualizačného objektu vymažú a zobrazí sa opäť predvolená sekcia histogramu
- nekompletné škálovanie - v prípade, že jeden diel osi nemá obsahovať celé číslo ale nejakú desatinnú hodnotu z toho vyplýval aj problém so správnym umiestnením binov v scéne, keďže pozícia závisí na jeho súradniciach v rámci osí
- komponent neobsahuje možnosť získať informácie o zvolenom bine - nie je možné získať údaje mimo komponent, čo napríklad umožňuje vizualizácia v JSROOT-e
- vhodné bude aj doplniť hodnotu obsahu na biny, keďže v histograme absentujú označenia pre Y os a nie je možné sa dopracovať ku konkrétnej číselnej hodnote binov v scéne
- transparentnosť projekcií zobrazenej na tabuli vo VR scéne - histogram a jeho označenia sú zahrnuté na čiernom pozadí entity
- nedostatočná univerzálnosť v prípade rôznych možností konfigurácie zobrazovaných sekcií - histogram sa nezobrazí správne alebo v prípade zmeny histogramu sa obnoví zobrazovaná sekcia, čo nepôsobí veľmi intuitívne a prakticky pre používateľa
- histogram neobsahuje možnosti konfigurovať vizuálne vlastnosti binov pri interakciách - vhodné poskytnúť možnosť definovať funkcie, ktoré sa vykonajú pri určitej interakcii s binom podľa potreby mimokomponentovej logiky aplikácie
- histogram po kliknutí na bin poskytuje iný formát údajov ako je v prípade JSROOT - vhodné to zjednotiť, minimálne pridať možnosť aby poskytoval údaje v rovnakej štruktúre

1.2.3 REACT-NDMSPC-CORE

Táto knižnica [3] obsahuje komponenty potrebné pre integráciu a prácu s knižnicou JSROOT v aplikácií založenej na softvérovom rámci *React*. Knižnica obsahuje

komponenty:

- **JsrootProvider** - React kontext zabezpečujúci zobrazenie elementov až po načítaní všetkých potrebných modulov knižnice JSROOT.
- **localStore** - Objekt obsahujúci *RxJs* subjekt a funkcionality pre spravovanie, uchovávanie a presmerovanie dát. Dáta sú v tomto prípade aj uchovávané v lokálnom úložisku prehliadača (localStorage) pred samotným publikovaním.
- **redirectStore** - Objekt obsahuje rovnakú funkcionalitu, ako **localStore** ale absentuje uchovávanie dát v lokálnom úložisku. Jeho úlohou je iba presmerovanie dát medzi uzlami bez ich ukladania.
- **NdmSpcWebSocket** - Implementácia webSocketu pre pripojenie a spracovanie dát.
- **useJsrootVersion** - React Hook zabezpečuje získanie aktuálnej hodnoty verzie knižnice JSROOT použitej v projekte.
- **useLocalStore** - React Hook zabezpečuje vždy aktuálne dáta z lokálneho úložiska **localStore**.
- **useRedirectStore** - React Hook zabezpečuje vždy aktuálne dáta z presmerovacieho úložiska **redirectStore**.

1.2.4 JSROOT

Keďže v popise úlohy pre túto prácu je definovať možnosť 2 typov vizualizácií, tak je potrebné analyzovať aj možnosti vizualizácie v JSROOT-e [8] na základe, ktorého bola vyvíjaná vizualizácia vo VR. Pre plnú funkcionalitu je potrebné zabezpečiť zobrazenie histogramu a následne zabezpečiť, aby sa po kliku na bin získali údaje o príslušnom bine a aby následne mohli nastať ďalšie úlohy závislé na daných údajoch. Knižnica bola podrobne analyzovaná v práci, kde sa riešil vývoj knižnice NDMVR [23] avšak ešte vo verzii **v5**, kde dnes už je k dispozícií nová verzia **v7**, ktorá obsahuje viacero zmien v porovnaní s predchádzajúcimi verziami. Pre analýzu potrebných konštrukcií, pre vyriešenie problému sme čerpali predovšetkým z gitlab dokumentácie JSROOT-u [8] ale pre hlbšie ponorenie do problému sme museli detailnejšie preštudovať aj *JsDoc* dokumentáciu JSROOT-u [27].

Problém spočíva v potrebe vykonať určitú akciu s údajmi po kliknutí na konkrétny bin. Histogram je vygenerovaný ako SVG [28] element v DOM funkciou *redraw* s parametrom funkcie definujúcim identifikátor cieľového elementu, do ktorého sa generovaný histogram vloží. Je potrebné *üviazať"* určitú funkciu, ako obsluhu udalostí aby pri **hover**, **klik**, **dvojklik** udalosti sme zadefinovali funkcionalitu, ktorá sa má vykonať.

JSROOT obsahuje možnosť definovať si vlastné spätne volané funkcie pre pokrytie základných udalostí, ako sú **klik**, **hover** a **dvojklik** a to pri vykonaní funkcie draw alebo redraw, ktoré vracajú objekt *painter* ale objekt je dostupný až po vykreslení histogramu.

Zdrojový kód 1.1: Fragment kódu zobrazujúci proces definície vlastnej spätne volanej funkcie pre vykreslený histogram

```
const handleBinClick = (data) => {
        console.log(data)
}

const drawHisto = async () => {
        const painter = await JSROOT.redraw(
                'jsroot_histo',
                histogram,
                'colz'
        )
        if (painter === null) {
        return null
        }
painter.configureUserClickHandler(handleBinClick)
}
```

v príklade vyššie je zobrazený príklad 1.1 definície spätne volanej funkcie na zachytávanie dát o kliknutom bine. Objekt *painter* sa získa, ako návratová hodnota funkcie *redraw*. Avšak ide o asynchrónnu funkciu, tak je potrebné počkať na *painter*, ktorý sa získa až po vytvorení histogramu, čo je dlhší proces, ako bežné vykonávanie kódu. Následne na získanom objekte *painter* vieme pomocou funkcie *configureUserClickHandler* definovať našu funkciu, ktorá sa bude vykonávať s príslušnými dátami po kliku na bin. V našom príklade 1.1 to je funkcia *handleBinClick*, ktorá získa údaje o kliknutom bine, ako parameter **data** a vypíše hodnotu do konzoly.

Pri vizualizácií histogramov pomocou knižnice JSROOT bude potrebné aj špecifikovať akcie, ktoré sa budú vykonávať pri spustení histogramu nad binmi, nie

len vo VR ale aj v klasickej verzií vizualizácie pomocou JSROOT-u. Pre tento účel obsahuje samotná knižnica objekt, ktorý umožní vytvoriť kontextové menu po vykreslení histogramu za účelom špecifikácie.

Objekt sa získa ako promise po vykreslení histogramu, napríklad použitím funkcie *display*, ktorá zobrazí daný histogram. Následne na získanom objekte vieme vytvárať ďalšie spätne volané funkcie na definíciu prispôsobiteľných funkcionalít. Použije sa podobný princíp, ako aj v predošlom príklade 1.1.

Zdrojový kód 1.2: Fragment kódu zobrazujúci proces získania objektu po vykreslení histogramu a následne definovanie funkcie na generovanie kontextového menu [29].

```
let histpainter = await hpainter.display("hpxpy;1", "colz");

histpainter
.oldFillHistContextMenu = histpainter.fillHistContextMenu;

// assign new context menu handler for TH2 drawing
histpainter.fillHistContextMenu = (menu) => {
     let itemname = this.getItemName();
     if (
     (typeof itemname == "string")
             &&
     (itemname.indexOf("hpxpy") >= 0)
     )
        {
        let tip = menu.painter.getToolTip(
                menu.getEventPosition()
        );
                menu.add(
                ‘sub:Histogram bin [${tip.binx}, ${tip.biny}]‘,
        () => menu.painter.provideSpecialDrawArea()
        );
        menu.add(
        "Show hpx",
        () => menu
                        .painter
                        .provideSpecialDrawArea("bottom")
                .then(() => hpainter.getObject("hpx"))
                .then(
                res => menu.painter.drawInSpecialArea(
```

```
                              res.obj,
                              "*H"
                              )
                    )
          );
          }
      return this.oldFillHistContextMenu(menu);
}
```

V zobrazenom zdrojovom kóde 1.2 je príklad vytvorenia vlastnej spätne volanej funkcie, ktorá vytvorí prispôsobené kontextové menu podľa našich požiadaviek. Po získaní inštancie objektu *histpainter* sa následne vytvorí spätne volaná funkcia, ktorá obsahuje funkcionalitu pre vytvorenie kontextového menu. Tá pozostáva zo získania **nápovedy** s pozíciou kurzora. Funkcia získava, ako parameter objekt reprezentujúci kontextové menu, ten obsahuje funkcie pre manipulácie s týmto objektom, ako napríklad funkciu *add*(), ktorou vieme pridať položku do kontextového menu a spätne volanou funkciou vieme definovať funkcionalitu, ktorá sa vykoná po zvolení tohoto binu. V prvom volaní funkcie *add* na objekte menu sa zobrazí špeciálne kontextové okno pre zobrazenie ďalších vnorených možností pre vizualizáciu, v tomto prípade teda možnosť pre zobrazenie kontajnera v spodnej časti obrazovky s vykresleným objektom.

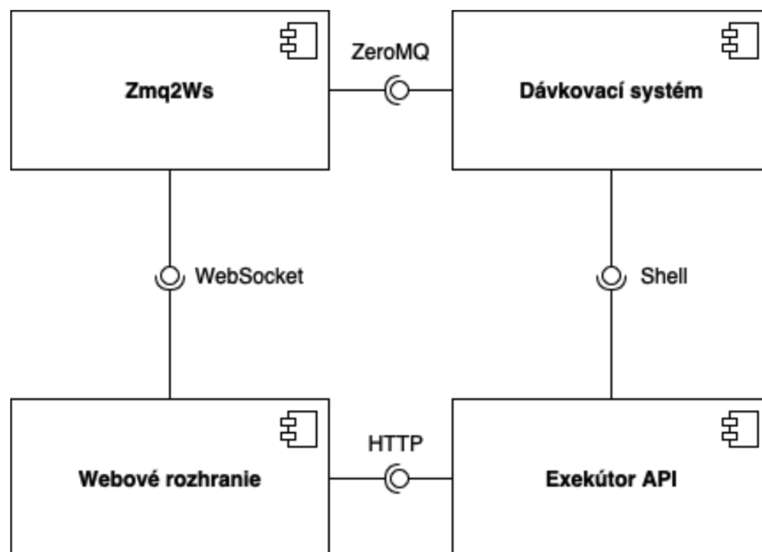
1.2.5 JSROOT migrácia na 7.1.0

Nová verzia knižnice (v tomto čase) obsahuje viacero zmien oproti predchádzajúcej verzii. Odlišuje sa hlavne v štruktúre celej knižnice, kde v predchádzajúcich verziách boli objekty získavané formou funkcií volaných na hlavnom objekte JSROOT, kde v najnovšej verzii je už knižnica organizovaná do modulov. Pre použitie objektov a funkcií je potrebné importovať si potrebné objekty a funkcie pred použitím bez nutnosti importovania globálneho objektu JSROOT. V momentálnej verzii však evidujeme problém s importovaním objektov a použitím vo webovom balíku založenom na softvérovom rámci React. Dôvodom sú v momentálnej fáze reštrikcie v "code style", v ktorom je knižnica napísaná a "code style", ktorý používa samotný React pri zostavovaní súborov pre použitie knižnice v klientskych aplikáciách.

1.3 Analýza monitorovacej aplikácie pre systém SALSA

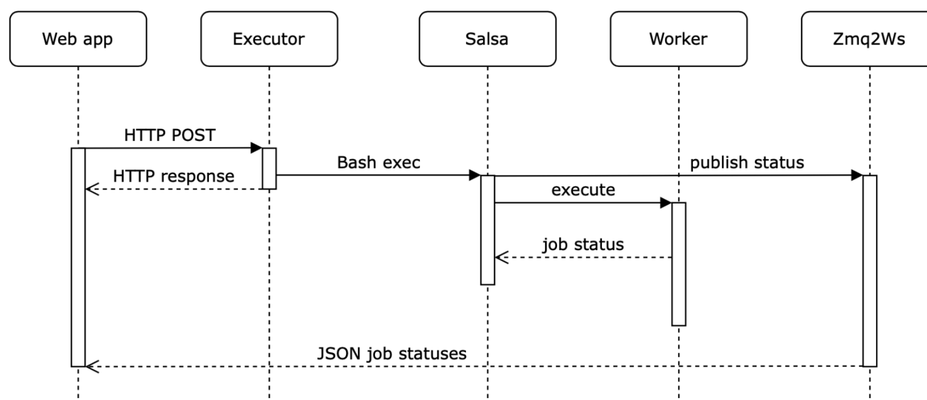
V bakalárskej práci [6] bol popísaný spôsob spracovania dát ohľadom úloh bežiacich na klástroch. Splĺňal účel zobrazovania stavov a progresu daných úloh na klástroch, ktoré boli zobrazované v tabuľke. Spúšťanie úloh pozostávalo zo spustenia makier v shell interpretéri na pracovnej stanici), na čo je potrebné pridať správne konfigurácie a následne sa spustením makra v interpretéri inicializovala úloha. Vykonávanie úlohy prebieha na klástri (klaster definuje viacero výpočtových jadier zlúčených do jednej jednotky [30]) pričom údaje o stave vykonávaných úloh sú monitorované výstupom dávkovacieho systému, ktorý distribuuje a dávkuje vstupné dáta, vhodné pre výpočet na pracovné jednotky. Dávkovací systém distribuuje vykonávanie úloh na pracovné jednotky, podľa dostupnosti, kapacity a vyťaženosti a spracováva výsledky a zabezpečuje distribúciu výsledkov a metadát. Údaje o stave úlohy sú následne publikované z dávkovacieho systému pomocou **zmq2ws** [31], kde pri pripojení na tento webSocket klient získava všetky dáta o úlohách vykonaných na klástri a ich stavoch, progresoch a pod.

Následne v diplomovej práci [32] bola implementácia rozšírená o možnosť spúšťania úloh priamo z klientskej aplikácie pomocou POST žiadosti, kde sa konfigurácie makra definovaného pre experiment doplnia o konkrétne konfigurácie klienta. Klient definuje potrebné vstupné dáta pre experiment ako argumenty, ktoré sa následne pripoja k žiadosti a po zavolaní tejto žiadosti sa spustí daný príkaz na vzdialenom klástri. Po pripojení klienta na **zmq2ws** webSocket sa získavajú dáta o všetkých spustených úlohách, ktoré sú publikované na strane monitorovacej aplikácie.



Obr. 1.11: Komponentový diagram systému pre spravovanie úloh (Zdroj: [32])

Komponentový diagram na obr. 1.11 znázorňuje prepojenie základných komponentov v architektúre, kde je možné vidieť vzťah webového rozhrania s websocketom **zmq2ws** a **exekútorom API**. V prípade **zmq2ws** ide o webSocket použitý už v prvej aplikácii vyvinutej na monitorovanie [6], kde sa dáta získavajú ako broadcast po pripojení klienta. V prípade **exekútor API** ide o REST službu, ktorou bude možné spustiť úlohu s príslušnými parametrami z ľubovoľnej klientskej aplikácie, ktorá nie je závislá na **zmq2ws**, jej úlohou je iba prijatie argumentov od klienta, následná inicializácia príkazu a spustenie. Dávkovací systém už distribuuje dáta pre výpočtové jednotky klástra a zhromažďuje metadáta, ktoré následne publikuje na **zmq2ws**.



Obr. 1.12: Sekvenčný diagram systému (Zdroj: [32])

Sekvenčný diagram na obr. 1.12 zobrazuje sekvenčný postup tejto interakcie založenej na spustení úlohy a jej monitorovaní v klientskej webovej aplikácii. Zavolaním POST metódy a pripojením potrebných parametrov k žiadosti sa inicializuje spustenie úlohy, následne je priradený identifikátor úlohy spolu s ďalšími metadátami, ktoré sa vrátia klientovi ako odpoveď na POST žiadosť. Dávkovací systém (v tomto prípade SALSA) následne spúšťa úlohy (na obrázku 1.12 ekvivalent job) a získava informácie o úlohe. Následne sú tieto informácie publikované na **zmq2ws** [31] a prijaté na opačnom konci spojenia, vo webovej aplikácii klienta, ktorý je pripojený na odber týchto dát.

Zdrojový kód 1.3: Telo objektu potrebného pre zaslanie žiadosti o spustenie úlohy.

```
{
    type: "salsa",
    subtype: "feeder",
    salsa: {
      host: "tcp://ndmspc-sample-sls-submitter-service:41000",
      job_type: "template"
    },
    command: "/app/proj2D.sh",
    args: `${x} ${y}`,
    numberOfTasks: 1,
    indexes: null,
    bins: null
};
```

Vo fragmente 1.3 je príklad definície objektu pre pripojenie k žiadosti. Objekt obsahuje atribúty ako typ dávkovacieho systému, príkaz ktorý je potrebné vykonať, argumenty pre vykonávaný príkaz, počet úloh a ďalšie. Pre použitie a vývoj funkcionalít v rámci tejto diplomovej práce postačuje toto rozhranie definované na 1.3. Podrobnejšie informácie sú obsiahnuté v prácach [6] a [32] a tie nebudú ďalším predmetom analýz v tejto diplomovej práci.

1.4 Technológie pre zostavovanie webových aplikácií

V skupine NDMSPC prebieha vývoj knižníc pre rozširovanie webových aplikácií založených na softvérovom rámci *React*. Knižnice predstavujú zapúzdrenú funkcionalitu, kde je použitá technológia *webpack* [5] pre zostavenie výsledného

[5] `https://webpack.js.org/`

optimalizovaného javascript kódu, ktorý sa preloží po inštalácií knižnice v klientskej aplikácií. Hlavnými charakteristikami je hlavne kompilácia optimalizovaného kodu s použitím všetkých potrebných závislostí pre knižnicu. V konečnom dôsledku sa medzi týmito technológiami porovnávajú hlavne efektivita kompilácie a rýchlosť generovania výsledného kódu. Charakteristické sú následne konfiguračné súbory v ktorých sa definujú konfigurácie na základe ktorých sa zostavujú výsledné zväzky.

Na projektoch v REACT-NDMSPC-CORE a NDMVR je použitá technológia *webpack* pre zostavovanie modulov pre klientsku webovú aplikáciu. Našli sme radu nevýhod v spojení tejto technológie so závislosťou JSROOT, kde pri importe najnovšej verzie knižnice JSROOT **verzie 7.1.0** s podporou modulov nastal problém pri zostavení výsledného zväzku. Dôvodom bola nedostatočná adaptácia všetkých modulov knižnice JSROOT na moderné technológie používane pri vývoji aplikácií (*React*, *Vue*, *Angular*). Z tohto dôvodu sme boli nútení sa porozhliadnuť a analyzovať aj iné technológie v tejto oblasti zostavovania.

1.4.1 Webpack

Ide o statický zväzovač pre aplikácie založené na javascript-e. Pri spracovaní aplikácie webpack interne vytvára graf závislostí z jedného alebo viacerých vstupných bodov a následne kombinuje každý modul, ktorý je potrebný do jedného alebo viacerých zväzkov. Jeho hlavným účelom je pretransformovať a zoskupiť súbory zdrojového kódu (napr. JavaScript, CSS, obrázky) do efektívneho optimalizovaného zväzku (balíka) pre distribúciu na produkčnom serveri.

Webpack poskytuje mnoho funkcií, ako napríklad modularitu, možnosti zmenšovania, ktoré zvyšujú efektivitu a rýchlosť nahrávania webovej stránky. Takisto poskytuje spätnú väzbu o chybách v kóde počas vývoja a zlepšuje testovania aplikácií.

V dnešnej dobe je najčastejšie používaný v modernom vývoji webových aplikácií, najmä v spojení s technológiami React [25], Vue [6] a Angular [7].

1.4.2 Vite

Je nástroj pre vývoj webových aplikácií a statických webových stránok, ktorého cieľom je prevažne zrýchlenie a uľahčenie procesu vývoja. *Vite.js* [8] používa nový prístup k správe závislostí, ktorý umožňuje rýchlejšie načítanie modulov počas

[6]`https://vuejs.org/`

[7]`https://angular.io/`

[8]`https://vitejs.dev/`

vývoja, bez toho aby bolo nutné ich predbežne kompilovať do zväzku. To znamená, že pri každom načítaní modulu, *Vite.js* nevytvára kompletný balík, ale nahráva iba tie moduly, ktoré sú potrebné pre aktuálnu stránku alebo knižnicu.

Vite.js podporuje množstvo rôznych programovacích jazykov a softvérových rámcov vrátane *JavaScript*, *TypeScript*, *Vue.js* a *React*. Okrem toho obsahuje aj vlastný vstavaný webový server, ktorý umožňuje rýchle a jednoduché testovanie aplikácie. Je vhodný pre vývoj moderných webových aplikácií a statických webových stránok, kde je potrebné zabezpečiť rýchlosť načítavania a efektívnosť spracovania závislostí.

1.4.3 Parcel

Parcel [9] je voľne dostupná technológia pre zostavovanie zväzkov a distribúciu webových aplikácií. Podobne ako webpack, umožňuje spájať zdrojové súbory, ako sú JavaScript, CSS, HTML, obrázky, atď. do efektívneho balíka pre nasadenie na produkčný server.

Hlavný rozdiel *Parcelu* je hlavne v tom, že nie sú potrebné konfigurácie a konfiguračné súbory. *Parcel* sa dokáže automaticky prispôsobiť a rozpoznať, aké typy súborov sú v projekte používané a vytvárať z nich zväzky (balíky). Tento prístup umožňuje vývojárom rýchlejší začiatok a menšiu potrebu štúdia samotnej technológie a jej konfigurácií.

Parcel taktiež podporuje množstvo rôznych jazykov a technológií, ako napríklad *JavaScript*, *TypeScript*, *Vue.js*, *React*, ale aj *CSS* preprocesory. Obsahuje aj vstavaný server, ktorý umožňuje rýchle testovanie aplikácie a okamžité zobrazovanie zmien v reálnom čase.

1.4.4 Zhrnutie a výber technológie

Ako voľbu technológie pre naše knižnice a aplikácie sme zvolili *Vite.js* hlavne kvôli možnosti konfigurácie použitia závislostí v aplikácii. Takisto je výhodou aj správa a použitie len potrebných závislostí v zostavovaných zväzkoch pre aplikáciu, čo sa odzrkadľuje aj na rýchlosti zostavovania v porovnaní s ostatnými technológiami.

[9] `https://parceljs.org/`

2 Návrh riešenia

V tejto fáze bude navrhnuté riešenie, ktoré bude v neskorších fázach implementované vo viacerých iteráciách. Keďže ide z veľkej časti o klientsku aplikáciu, ktorá umožní používateľom spravovať, analyzovať, riadiť a monitorovať joby. Vizualizácia dát bude podporovať klasický pohľad použitím vizualizácie v 2D priestore pomocou JSROOT-u aj s podporou vizualizácie vo VR použitím NDMVR.

Návrh, ako aj implementácia budú pozostávať z viacerých kapitol. V prvej fáze budú popísané požiadavky na vyvíjané riešenie na základe, ktorých sa bude následne pristupovať k návrhu vylepšení v jednotlivých knižniciach.

Druhá kapitola bude popisovať návrh vylepšení knižnice NDMVR, do ktorej je potrebné implementovať viaceré vylepšenia a vyriešiť problémy, ktoré sú v rozpore s požiadavkami kladenými na vizualizáciu. Pri implementácii finálneho komponentu a refaktorizácii NDMVR bude využitá aj knižnica REACT-NDMSPC-CORE, ktorá obsahuje viaceré koreňové funkcionality spojené s JSROOT. REACT-NDMSPC-CORE obsahuje aj JSROOT provider, ktorý v aplikácii zabezpečuje načítanie knižnice JSROOT na strane klienta. Pri Implementáciách vylepšení knižnice NDMVR je vhodné extrahovať niektoré funkcionality týkajúce sa len knižnice JSROOT do spomínanej knižnice REACT NDMSPC CORE, ktorá je vyvinutá na tento účel.

Tretia fáza sa bude venovať návrhu používateľského rozhrania, ktoré bude predstavovať celok s integráciou všetkých potrebných funkcionalít a komponentov. Výsledné riešenie bude knižnica NDMSPC, ktorá bude zlučovať vyššie funkcionality do jedného celku. Komponenty knižnice budú predstavovať rozšírenie klientskej aplikácie. Celkový koncept a účel tejto knižnice bude spočívať v definícii rôznych komponentov pre rozličné vizualizácie. Knižnica zabezpečí a poskytne všetky potrebné komponenty a funkcie, aby bolo možné vytvoriť vizualizačný komponent pre prípad konkrétneho používateľského rozhrania v danom odvetví. Následne sa v klientskej aplikácii len nainštaluje daná knižnica a použije sa konkrétny komponent, ktorý zabezpečí plnohodnotné používateľské rozhranie pre danú aplikáciu.

Finálne riešenie by malo predstavovať komponent, ktorý sa bude dať importovať a vložiť do prostredia aplikácie založenej na softvérovom rámci React. Knižnica by mala obsahovať použiteľné komponenty pre jednotlivé vizualizácie. Komponenty by mali spĺňať účely vizualizácie a poskytnúť možnosť interakcií navrhnutých pre špecifický prípad použitia.V našom prípade ide o jeden komponent.

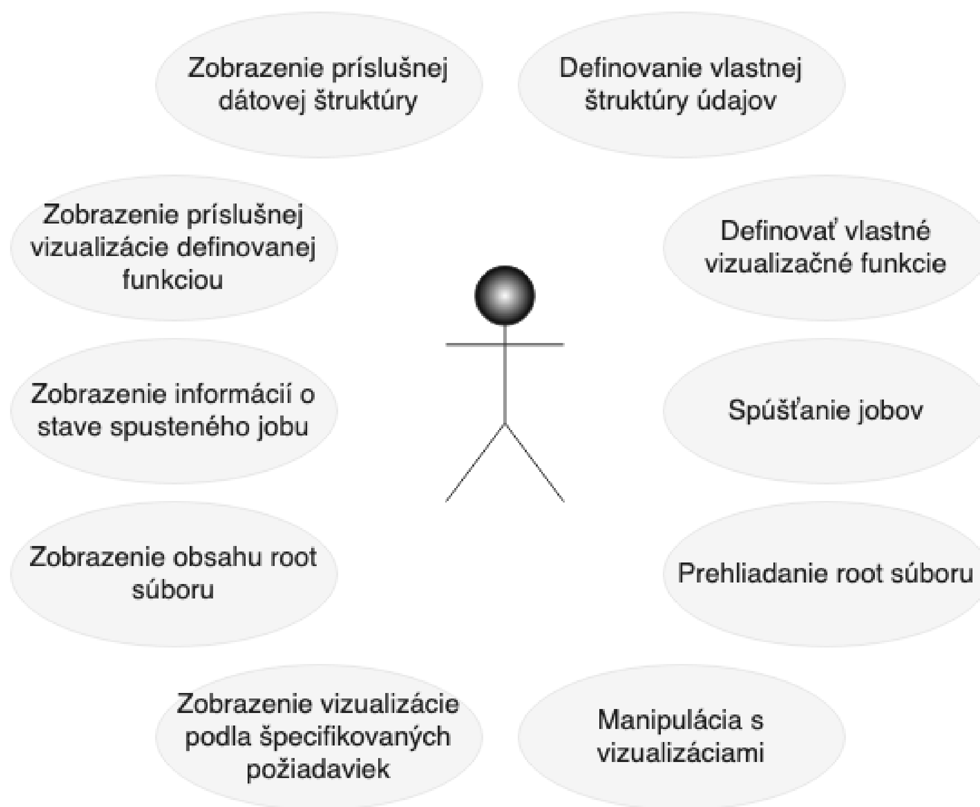
2.1 Definícia požiadaviek

Pre implementáciu riešenia je potrebné vytvoriť príklad takéhoto komponentu, ktorý bude implementovaný v knižnici NDMSPC a popri vývoji tak zistiť slabiny a nedostatky knižníc, ktoré obsahujú komponenty dôležité pre rôzne funkcionality nášho vytváraného používateľského rozhrania. Takto je možné definovať nedostatky funkcionalít potrebných pre implementáciu takéhoto používateľského rozhrania.

V prvotnej fáze je potrebné podrobne špecifikovať požiadavky používateľov na vyvíjané riešenie. Dôležitá je forma riešenia, ktorú by sme mali poskytnúť a to konkrétne či ide o webovú aplikáciu, natívnu aplikáciu alebo komponent využiteľný už v klientovom prostredí. Na základe týchto ale aj ďalších požiadaviek sa definuje finálna podoba riešenia. Pri určovaní požiadaviek sme využili prevažne konzultácie s RNDr. Martin Vaľa PhD, ktorý ako vedec pôsobí v odvetví fyzikálnych experimentov a v tomto projekte zastaval rolu hlavného zákazníka. Hlavné požiadavky boli určené po sérií konzultácií a sústavne pribúdali aj počas implementácie riešenia. Takisto boli konzultované aj vylepšenia a požiadavky nadobudnuté počas analýzy.

2.1.1 Diagram prípadov použitia

Počas série konzultácií so zákazníkom vzniklo pomerne veľké množstvo požiadaviek. V danom projekte figuruje viacero oblastí, ktoré je potrebné implementovať a ktoré sú už v nejakom stave. Pre kapacitu tejto práce nebolo možné zachytiť všetky požiadavky, keďže ide o širokú funkcionalitu využívanú v tejto doméne. Vzniknuté požiadavky sme preto podrobne prekonzultovali na viacerých stretnutiach a určili tak základné požiadavky, na ktoré sme sa zameriavali počas implementácie.



Obr. 2.1: Diagram prípadov použitia

Na obrázku 2.1 sú zobrazené obšírne požiadavky, ktoré je potrebné zabezpečiť pre aktéra. Požiadavky sú rozdelené do dvoch skupín a to podľa vzťahu interakcie k aktérovi. Vľavo sú výstupné požiadavky, ktoré definujú zmeny vizualizačného prostredia, dátových štruktúr ako napríklad zobrazenie obsahu ROOT súboru. Vpravo je možné vidieť vstupné akcie, ktoré vyžadujú vytvorenie udalosti od aktéra. V tomto prípade ide základný diagram požiadaviek pre celé riešenie.

Vo výsledku však navrhnuté komponenty nemusia byť implementované priamo v knižnici NDMSPC ale komponenty, ktoré majú univerzálny charakter a je predpoklad, že sa v cieľovej doméne budú používať pri implementácii týchto vizualizačných komponentov a budú extrahované do knižnice REACT-NDMSPC-CORE a to funkcionality týkajúce sa JSROOTU a klasického webového rozhrania. V prípade funkcionalít súvisiacich s vizualizáciami v rozšírenej realite bude postupne dopĺňaná knižnica NDMVR.Ide o funkcionality, ktoré sa nebudú týkať vyslovene vyvíjanej vizualizácie.

2.1.2 Zhrnutie požadovaných funkcionalít

Na základe charakteru požiadaviek a domény bude finálne riešenie implementované, ako knižnica NDMSPC obsahujúca komponenty potrebné pre použitie v

klientskych webových aplikáciách podľa potreby. Keďže väčšina aplikácií je založená na softvérovom rámci REACT tak aj knižnica NDMSPC musí byť založená na tomto softvérovom rámci. Takisto aj vyššie spomínané závislosti NDMVR a REACT-NDMSPC-CORE.

Medzi základné funkcionality pre vývoj knižnice patria:

- pohľad pre vizualizáciu histogramov v rozšírenej realite s možnosťou identickej vizualizácie aj klasicky v prostredí webu použitím knižnice JSROOT.
- ROOT prehliadač schopný otvárať ROOT súbory a identifikovať dátové štruktúry, následne vytvárať pokročilé funkcionality a vizualizácie, napríklad pomocou kontextového menu alebo pomocou efektov v scéne rozšírenej reality.
- Komponent pre zabezpečenie spúšťania úloh na klástri a aj monitorovanie priebehu ich vykonávania vrátane spracovania výsledkov.
- Konfigurátor objektov, napríklad možnosť vytvárania a generovania vlastných štruktúr (napríklad preddefinované histogramy určené na ďalšie spracovávanie)

Počas implementácie definujeme kritické funkcionality, ktoré sú potrebné pri návrhu a dizajne takéhoto komponentu. Keďže väčšina týchto vizualizácií vyžaduje isté typy funkcionalít (napr. otváranie a vytváranie ROOT objektov, súborov, komunikáciu medzi komponentami na rôznych úrovniach, vytváranie histogramov atď.) tak tieto funkcionality budú extrahované z už implementovaných komponentov do separátnej knižnice REACT-NDMSPC-CORE. Po naplnení knižnice sa v budúcnosti budú dať implementovať rôzne vizualizačné komponenty využitím už implementovaných funkcií a komponentov v knižniciach REACT-NDMSPC-CORE a NDMVR.

2.2 Návrh komponentu NDMSPC

Na základe spísania požiadaviek vyplynulo riešenie, ktoré má nadobudnúť formu komponentu, ktorý sa nainštaluje ako závislosť do klientskej aplikácie. Vybraný komponent už bol vyvinutý ako výsledok predchádzajúcej diplomovej práce, kde komponent obsahoval funkcionalitu umožňujúcu monitorovať progres úloh púšťaných na klástri. Komponent umožňoval spustiť úlohu na klástri a následne obsahoval monitorovaciu tabuľku s výsledkami priebehu úloh na klástroch. Hlavným cieľom bolo vhodne navrhnúť rozšírenia tohto komponentu tak ,aby bolo

možné jednotlivé komponenty prepoužiť a za týmto účelom boli navrhnuté aj ďalšie vylepšenia a extrahovaná funkcionalita do separátnych knižníc, ako napríklad NDMVR a REACT-NDMSPC-CORE.

Charakter komponentu by mal predstavovať konkrétny vizualizačný komponent navrhnutý podľa konkrétnych požiadaviek pre doménu fyziky. Preto pri vývoji bol kladený dôraz na vytváranie funkcií a vylepšení knižnice NDMVR pre čo najuniverzálnejšie použitie. Rovnako aj v prípade funkcionalít týkajúcich sa len JSROOT-u a iných knižníc, tie boli sústredené v knižnici REACT-NDMSPC-CORE. Funkcionalita v knižniciach NDMVR a REACT-NDMSPC-CORE má v konečnom dôsledku nadobudnúť funkcie a komponenty, s ktorými bude možné vyvinúť komponent pre vizualizácie založené na knižnici JSROOT s podporou *RxJs* a zobrazovaním výsledkov v rozšírenej realite. A to komponent podľa špecifických požiadaviek, ktoré bude vyžadovať konkrétna doména.

2.2.1 Architektúra REACT-NDMSPC po aplikovaní koncepčných riešení

Ná základe určených požiadaviek bola navrhnutá nová architektúra 2.2 celého komponentu obsahujúca aj ostatné potrebné komponenty. REACT-NDMSPC komponent je závislý svojou funkcionalitou na knižniciach NDMVR a REACT-NDMSPC-CORE (obsahujúci JSROOT). Všetky potrebné funkcie sa importujú z poskytnutých knižníc a vhodnou implementáciou vznikne funkčný vizualizačný komponent založený n JSROOT-e a Aframe. Požadované funkcionality boli rozdelené podľa ich charakteru na vizualizačné a funkčné.

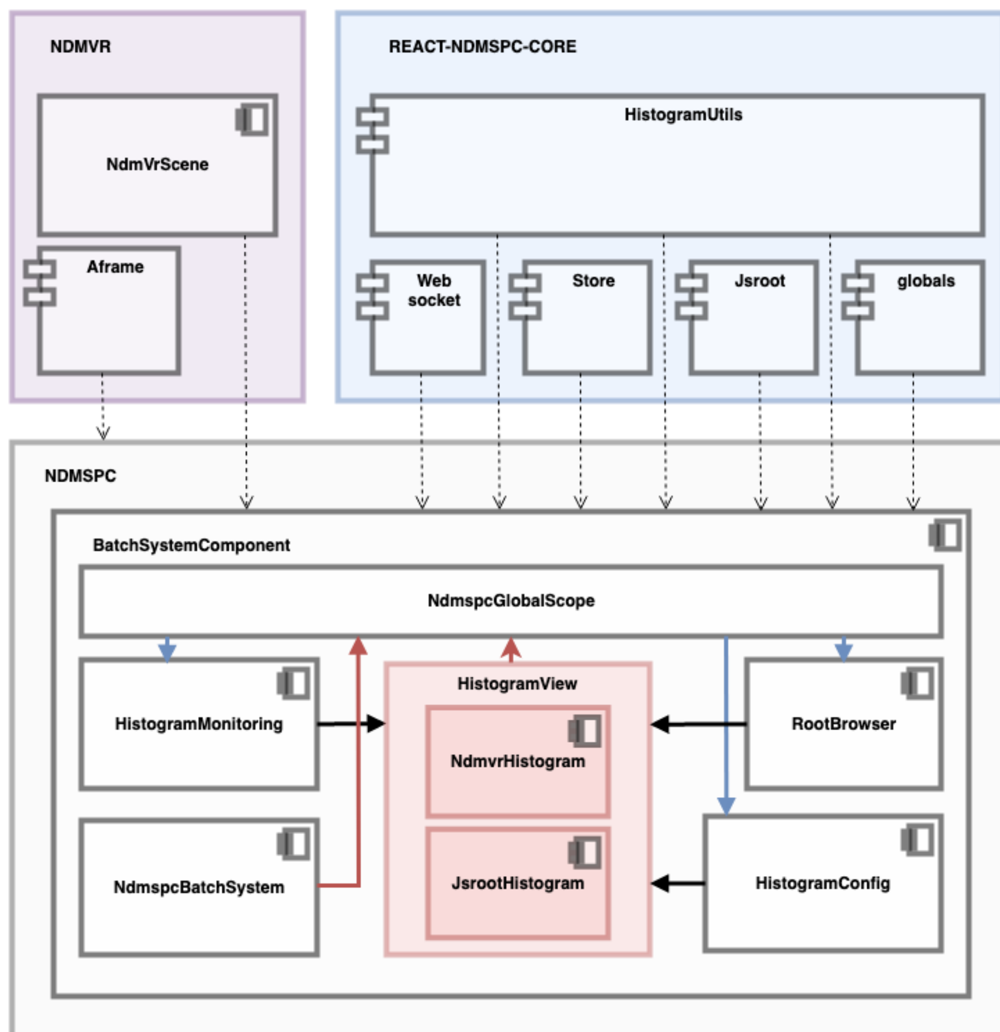
- **Vizualizačná** definuje základný pohľad na vizualizáciu výsledkov a bude využitá vo viacerých komponentoch ako napríklad v prehliadači root súborov, v monitorovacom a konfiguračnom komponente. Komponent prepína vizualizácie z klasického 2D pohľadu na scénu rozšírenej reality a jeho dátovým vstupom majú byť funkcie definované ako parametre spolu s objektom histogramu. Funkcie pre spracovávanie udalostí z vizualizácie a funkcie rôzne projekčné zobrazenia. V architektúre predstavuje komponent **HistogramView** prepínací komponenty **NdmvrHistogram** a **JsrootHistogram**.

- **Funkčná** definuje funkčné komponenty pre naplnenie jednotlivých požiadaviek. V architektúre sú to komponenty **HistogramMonitoring**, **RootBrowser** a **HistogramConfig**.

- **Funkčný komponent NdmspcBatchSystem** komponent bol vyvinutý a preto sa v tejto práci použila len jeho funkcionalita.

V riešení je potrebné zabezpečiť aj zdieľanie stavu medzi komponentami. Komponenty budú zapúzdrovať funkcionalitu a budú zobrazované nezávisle na sebe. **HistogramMonitoring** komponent bude monitorovať stav úloh na klástri ale pri zmene zobrazovaného komponentu na **RootBrowser** bude potrebné vhodne určiť progres a stav daného binu v závislosti od stavu úlohy. Riešenie obsahuje aj globálny komponent alebo objekt **AppGlobalScope** pre uchovávanie globálneho stavu komponentu a zabezpečenie komunikácie medzi komponentami.

Komunikácia medzi komponentami je znázornená šípkami, kde červená šípka predstavuje vykonanie spätne volanej funkcie pre vstup (poslanie údajov do hlavného aplikačného kontextu) a modrá spätne volanú funkciu pre výstup (poslanie údajov z hlavného aplikačného kontextu). Klasická čierna šípka znázorňuje komunikáciu medzi komponentami v aplikácií.



Obr. 2.2: Konceptuálny model architektúry komponentu NDMSPC

2.2.2 Funkčné komponenty REACT-NDMSPC

Ako bolo popísané na obr. 2.2 jedná sa o komponenty:

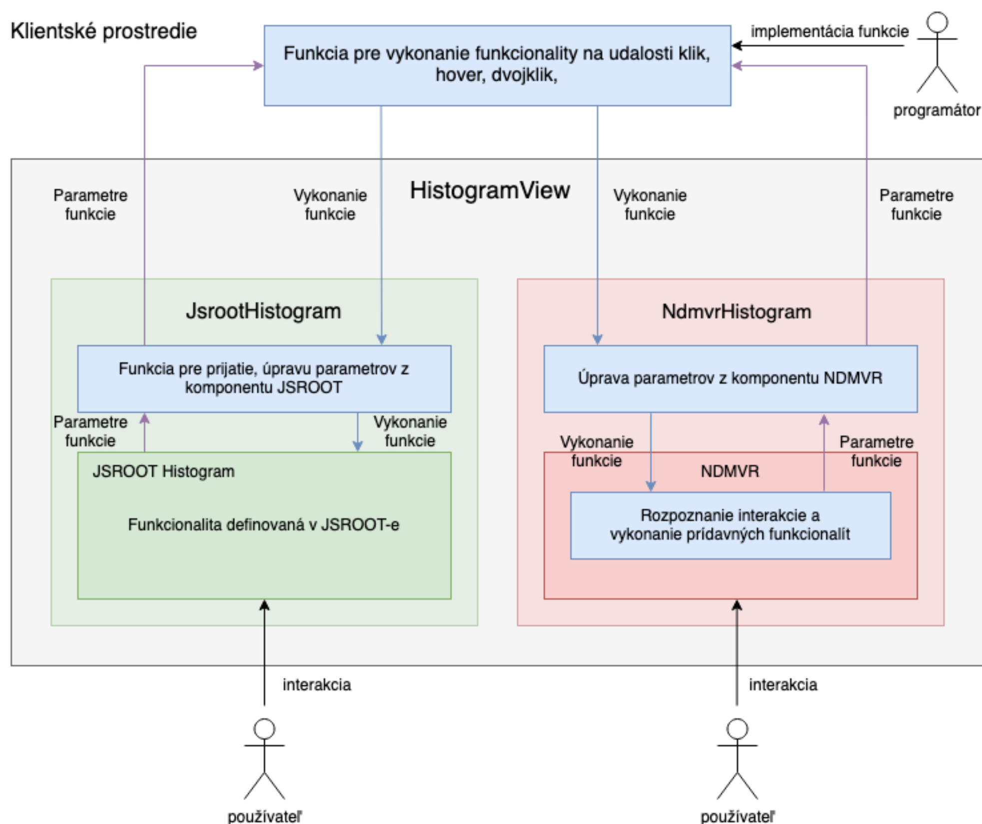
- **NdmspcMonitoring** slúžiaci na spúšťanie úloh a monitorovanie stavu. Zobrazuje výsledky získané zo vzdialeného websocketu a vizualizuje ich v tabuľke alebo formou histogramu použitím komponentu **HistogramView**.
- **RootBrowser** slúžiaci na načítanie root súboru a zobrazenie histogramu taktiež použitím komponentu **HistogramView**.
- **HistogramConfig** slúžiaci na konfiguráciu vlastného histogramu na spúšťanie úloh a monitorovanie ich stavu. Komponent obsahuje komponent **HistogramView** pre zobrazenie histogramu ale obsahuje aj formulár pre vytváranie histogramu. Takisto zahŕňa aj možnosť vytvárať viacero histogramov a ich administráciu.

Pre menežment vizualizácií nie je postačujúce len zobraziť výsledný histogram ale aj zabezpečiť možnosti definícií zmien a interakcií. Tento účel je náročný vzhľadom na použitie univerzálnych komponentov s svojou vlastnou funkcionalitou, ktorá nie je dostupná v našom rozsahu. V komponente, ktorý je vyvíjaný je potrebné zabezpečiť zobrazenie stavu úlohy na klástri, ktorý by mala reprezentovať výška binu a farba rovnako ako je potrebné aj zabezpečiť aby sa po interakcií používateľa s konkrétnym binom vykonala funkcia. Funkcia definuje konkrétnu funkcionalitu, ktorú definujú požiadavky a vzhľadom na komponent NDMVR, ktorý je vyvinutý pre univerzálne vizualizácie, preto je potrebné vytvoriť princíp, ktorý zabezpečí vhodnú separáciu týchto funkcionalít tak, aby vizualizačný komponent vykonával len svoju funkcionalitu týkajúcu sa vizualizácie dát a interakcií vo vnútri svojho kontextu (v tomto prípade ide o scénu). Všetká funkcionalita definujúca naplnenie požiadaviek musí byť definovaná v rozsahu komponentu REACT-NDMSPC a musí spracovávať parametre získané z vizualizačného komponentu, ktorých distribúcia musí byť zabezpečená.

2.2.3 Koncept spracovania udalostí

Pre riešenie prvého koncepčného problému sme stavali na existujúcej implementácii existujúcej v knižnici JSROOT, kde je možné definovať vlastné spätne volané funkcie, ktoré sa potom použijú pri udalostiach vzniknutých interakciami používateľov s binmi histogramu. Tento princíp bol použitý aj pre vyriešenie nášho

problému, kde bola implementovaná podpora priamo v komponente NDMVR pre sprostredkovanie udalostí a parametrov mimo rozsah NDMVR.

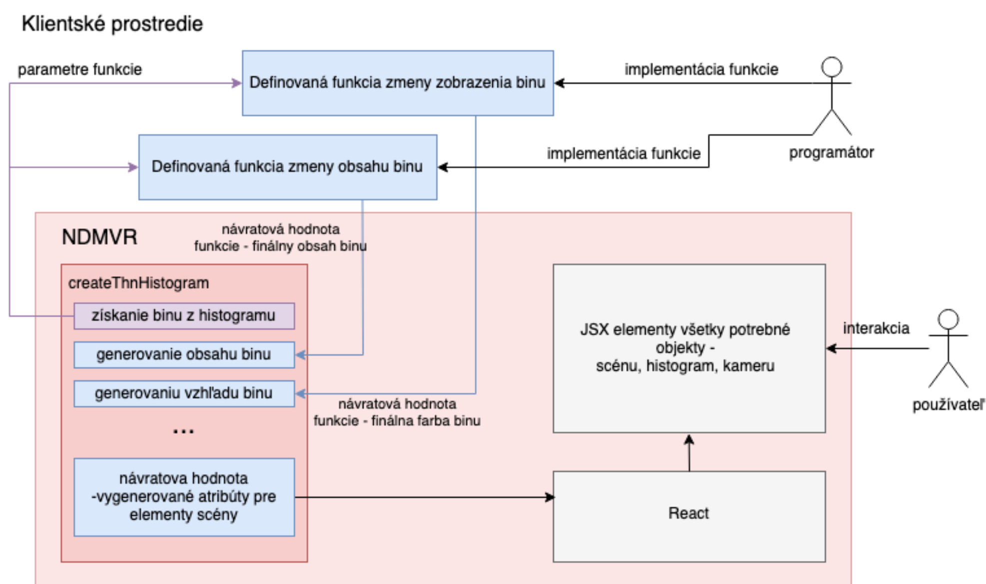


Obr. 2.3: Model popisujúci koncept spracovávania udalostí mimo vizualizačný komponent

Model takejto interakcie je znázornený na obr. 2.3 kde je zobrazený návrh vizualizačného komponentu, ktorý zlučuje oba vizualizácie do jediného komponentu, ktorý manažuje vizualizácie. V definovanom kontexte NDMSPC je podľa znázorneného modelu 2.3 funkcionalita implementovaná ako spätne volaná funkcia, ktorá sa pošle ako parameter do vnútra komponentu. Funkcia sa následne volá už v bežiacej scéne pri vyvolaní udalosti používateľom. NDMVR následne pri vykonaní tejto funkcie poskytne parametre priamo z vnútra kontextu NDMVR a definovaná funkcionalita sa vykoná spolu s poskytnutými parametrami. Implementácia komponentu NDMVR určí moment vykonania tejto definovanej funkcie práve pri vyvolaní udalosti po **kliku**, **hoveri** alebo **dvojkliku** na bin. Spomínanými parametrami sú v tomto prípade údaje dôležité pre správnu klasifikáciu danej udalosti vzhľadom na vizualizované dáta a miesto vyvolania udalosti. Spätne volané funkcie neposkytujú žiadnu návratovú hodnotu.

2.2.4 Koncept modifikácie vizualizácie na základe vonkajších parametrov

V tomto prípade ide o podobný problém ako v predchádzajúcej sekcii ale finálny stav predstavuje zobrazované dáta, ktoré zohľadňujú stav NDMSPC kontextu. Vzhľadom na vykonanie požadovaných funkcionalít je potrebné premietnuť ich výsledok do vizualizovaného histogramu vo forme zmeny zafarbenia alebo hodnoty binu.

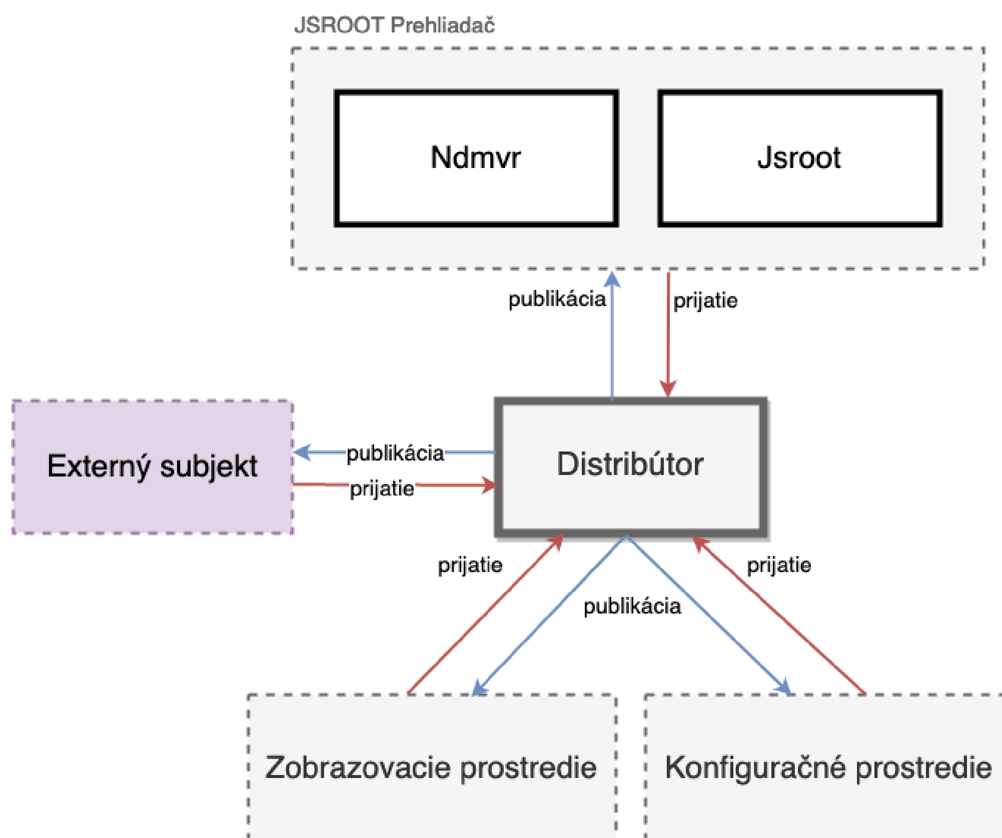


Obr. 2.4: Model popisujúci koncept modifikácie vizualizácie na základe parametrov mimo vizualizačný komponent

Model tohto typu interakcie popisuje obr. 2.4, kde spätne volané funkcie sú opäť ako aj v prípade predošlého riešenia definované na úrovni kontextu NDMSPC. Funkcie sú zaslané ako parametre do komponentu NDMVR a volané sú v určitom bode vo fáze generovania celej vizualizácie. Ako parametre sa taktiež poskytnú dáta definujúce vizualizované dáta a generovanú entitu. V ľavej časti modelu 2.4 je zobrazená vizualizácia funkcie createThnHistogram, ktorá názorne popisuje poradie pri generovaní atribútov pre entít. V prvej fáze sa získajú potrebné údaje o generovanom bine, s ktorými sa následne vykoná používateľská spätne volaná funkcia. Funkcia vráti výslednú hodnotu, ktorá bude predstavovať podľa charakteru modifikovaného atribútu farbu alebo výšku binu. Spätne volané funkcie v tomto prípade poskytujú návratovú hodnotu, ktorá definuje aj hodnotu atribútu pre vizualizáciu.

2.2.5 Komunikácia medzi nezávislými komponentami a zdieľanie stavov

V poslednej časti návrhu je dôležité vyriešiť najpodstatnejší problém, ktorý spočíva v možnosti zdieľať údaje medzi viacerými komponentami zahrnutými v REACT-NDMSPC. Pre prípad selekcie binu v komponente prehliadača následne potrebujeme selektovaný bin zohľadniť aj v komponente pre monitorovanie stavu úlohy na klástri.



Obr. 2.5: Zobrazenie konceptu distribútora dát medzi rôznymi komponentami v aplikácii

Na obr. 2.5 je zobrazený detailný princíp fungovania takéhoto objektu. Distribútor obsahuje subjekt knižnice RxJS, ktorý umožní publikovať dáta a poskytne inštanciu na ktorú potom vieme nastaviť funkciu, ktorá bude vykonaná vždy ako získame nejaké dáta, ktoré boli publikované z ľubovolnej časti aplikácie. Skrátene distribútor je rozšírenie tohto subjektu, ktoré obsahuje metódu pre získanie sledovateľnej inštancie ale aj funkcie na publikácie údajov v požadovanej štruktúre. Konceptuálny model, zobrazený na obr. 2.5 zobrazuje príklad, kedy distribútor je centrálny bod medzi komponentami, ktoré potrebujú komunikovať medzi sebou. Ako príklad je zobrazený už konkrétny koncept distribútora vzhľadom na po-

užitie v REACT-NDMSPC v ktorom je potrebné zabezpečiť komunikáciu medzi zobrazovacím prostredím, konfiguračným prostredím a prehliadačom.

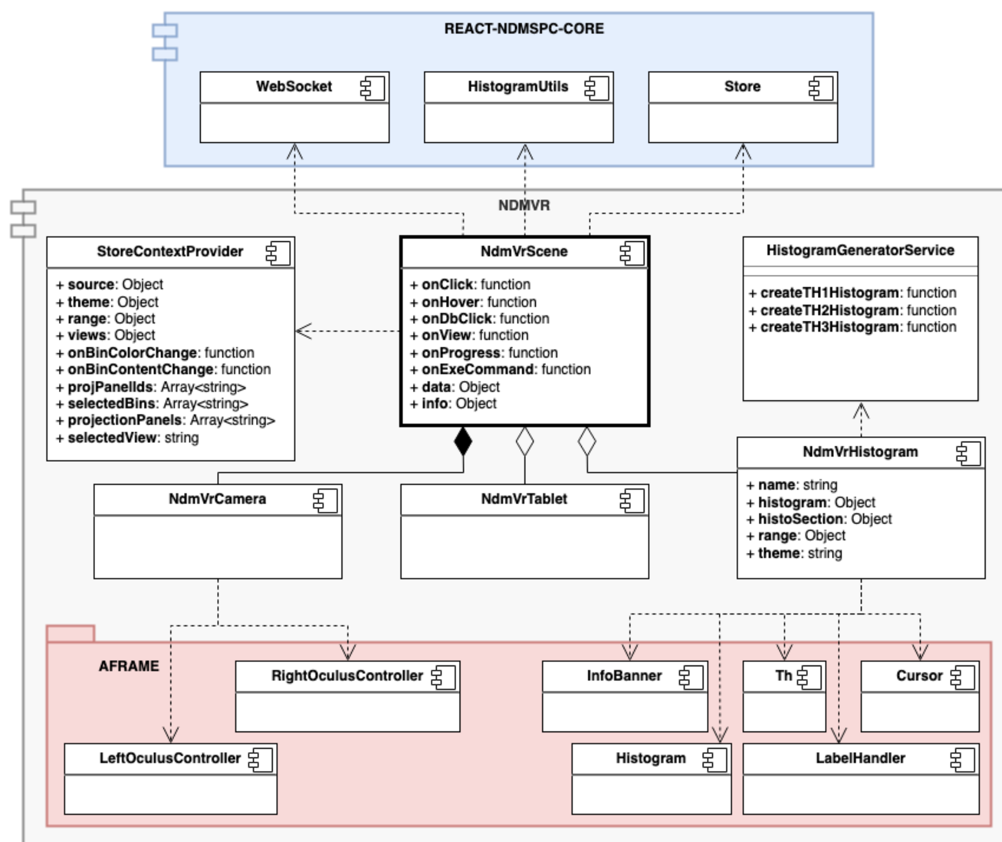
2.3 Návrh vylepšení NDMVR

Na základe špecifikácie pre vyvinutie komponentu pre vizulizáciu je nutné určiť vylepšenia a upraviť tak existujúci komponent NDMVR aby poskytoval všetky potrebné možnosti interakcie. V prvej fáze bol komponent otestovaný na príkladoch histogramov, kde sa pri testovaní bral ohľad hlavne na aspekty, ktoré sme získali pri špecifikácií požiadaviek. Následne boli definované body, ktoré reprezentujú oblasti, ktoré bude potrebné prepracovať aby komponent vo finálnej fáze spĺňal všetky potrebné požiadavky. Spomínané oblasti vylepšení NDMVR boli definované v týchto bodoch:

- globálna refaktorizácia zastaraných štýlov a riešení spojených s asynchrónnymi fragmentami kódu ale aj podpora modularity projektu.
- odstránenie chýb, ktoré vznikli testovaním na väčšej škále dát s rôznym charakterom.
- refaktorizácia a zjednodušenie štruktúry súborov projektu, efektívnejšia a logickejšia organizácia adresárov a súborov.
- vylepšenie generátora pre podporu vizualizácie dvojrozmerných histogramov.
- doplnenie nástrojov pre väčšiu modularitu v prípade zobrazovania histogramov a zložitejších viac. histogramových vizualizácií (viac histogramov a prepojenia priamo v komponente NDMVR) rovnako aj distribúcia potrebných údajov o binoch mimo komponent.
- implementovanie rozšírenia pre vizualizáciu jednorozmerných histogramov TH1 vo virtuálnej realite.
- snaha o zjednotenie komponentových štruktúr spolu s JSROOT. Zabezpečiť aby štruktúry dát o binoch a možnosti pre definovanie funkcionalít boli čo najviac podobné tým v knižnici JSROOT-e.

2.3.1 Nová architektúra NDMVR po aplikovaní koncepčných riešení

Na základe definovaných oblastí bolo potrebné aplikovať viacero úprav architektúry celej knižnice, aby sme funkcionality vedeli neskôr do komponentu dodať. Základný koncept architektúry je značne podobný ako prvá verzia vyvinutá počas implementácie bakalárkej práce, kde bola definovaná architektúra a následne podľa nej bol vyvinutý prvý komponent. Základné koncepčné prvky zostali, ako napríklad komponenty pre AFRAME a aj základný 2 vrstvový model A-Frame a React, kde A-Frame definuje funkcionalitu na úrovni Aframe komponentov a teda interakcie s elementami histogramu a funkcionalita na úrovni React-u pre zobrazovanie a aktualizáciu elementov.



Obr. 2.6: Konceptuálny model novej architektúry knižnice NDMVR

Rozdiel oproti pôvodnej architektúre spočíva teda v spôsobe generovania jednotlivých elementov, kde ako je zobrazené v diagrame 2.6 je definovaná servisná trieda **HistogramGeneratorService**. Trieda definuje funkcie pre generovanie údajových štruktúr určených pre vytvorenie potrebných entít pre histogram a to v podobe dátovej štruktúry a nie konkrétnych elementov. Trieda nevráti žiadne konkrétne elementy iba atribúty nutné pre definovanie potrebných vlastností a

vzhľadu elementov. Konkrétne elementy už dodá technológia na vyššej úrovni v našom prípade React zobrazí potrebné*JSX* elementy s požadovanými atribútmi.

Knižnica prišla o radu funkcionalít v porovnaní s predchádzajúcou verziou. Funkcie súvisiace s JSROOT a komponent slúžiaci na vizualizáciu klasických histogramov na panel boli presunuté do knižnice REACT-NDMSPC-CORE. Zjednodušili sa aj potrebné závislosti, kde v novej verzii je knižnica závislá iba na knižnici REACT-NDMSPC-CORE a závislosť JSROOT už nie je potrebná. Všetky funkcionality už sú sústredené práve do knižnice REACT-NDMSPC-CORE. Príklad je popísaný v diagrame 2.6 kde je zobrazený modul REACT-NDMSPC-CORE obsahujúci potrebné komponenty, ako napríklad **WebSocket**, **HistogramUtils** a **Store** tieto komponenty budú popísane v neskorších kapitolách venujúcim sa návrhom a implementáciou REACT-NDMSPC-CORE.

Najväčšia zmena oproti predchádzajúcej verzii je spôsob definovania funkcionalít z vonkajšieho prostredia. Komponent **NdmspcVrScene** obsahuje atribúty pre definovanie funkcií, ktorými bude umožnené pristupovať k potrebným údajom o binoch a histograme. Rovnako bude možné aj modifikovať vzhľad histogramov poprípade zaznamenávať udalosti, ktoré nastali vo vnútri scény. Rovnako sa zmenila aj štruktúra vstupných atribútov, ktoré sú poskytnuté komponentu. Zdrojové dáta boli umiestnené do osobitného objektu s názvom **data** a údaje nesúvisiace priamo s obsahom vizualizácie ale iba so stavmi zobrazenia a scény boli umiestnené do objektu **info**.

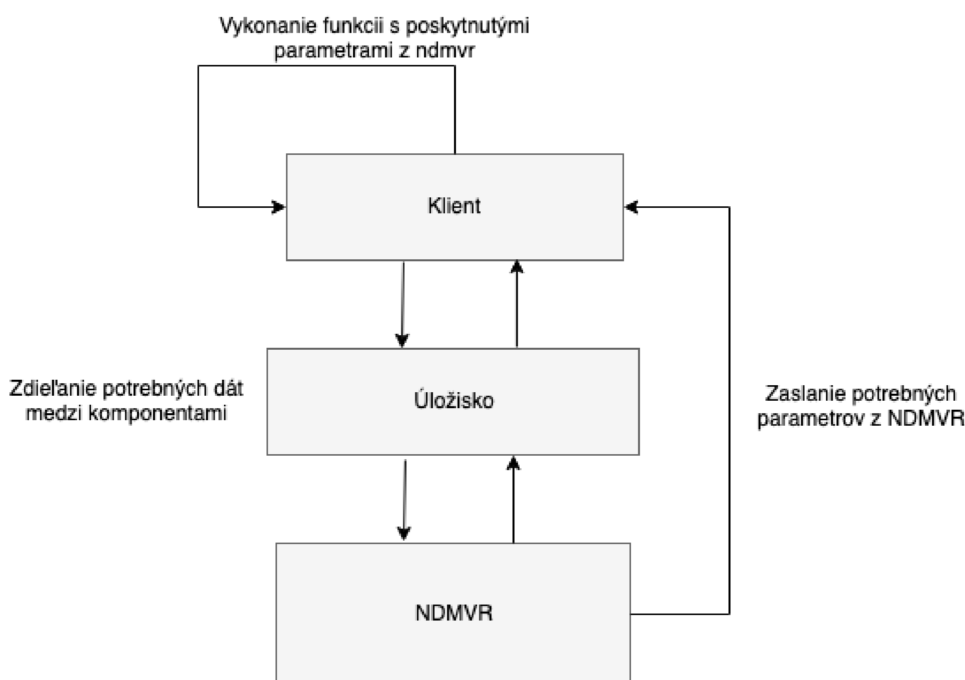
2.4 Návrh vylepšení REACT-NDMSPC-CORE

Knižnica obsahuje komponent **JsrootProvider**, ktorý je používaný v aplikácii na zabezpečenie načítania knižnice JSROOT. Okrem tohto komponentu sú k dispozícií aj ďalšie funkcie **localStore** a **redirectStore**. Pri vývoji riešenia potrebujeme vytvoriť obdobné funkcie, a rozšíriť knižnicu o radu funkcionalít, ktoré budú využité neskôr pri implementácií ďalších knižníc založených na JSROOT-e.

2.4.1 Subjekt pre vykonávanie funkcií

Problém spočíva v oblasti komplexnejších dátových analýz, kde je potrebné manažovať zobrazovanie viacerých histogramov na front-ende aplikácie, ktoré sú primárne zodpovedné za získavanie dát a vizualizovanie ich pomocou NDMVR. Problém nastáva hlavne v potrebe aktualizácie histogramov vo vnútri scény, kde z pohľadu možných interakcií neexistuje možnosť prispôsobiť komponent, aby univerzálne reagoval a menil aktuálny zobrazený histogram.

Pre tento účel komunikácie bol navrhnutý koncept, ktorý predstavuje exekútor funkcií. Tento subjekt je založený na *RxJs*, ktorý umožní formou publikácií signálov z vnútra NDMVR na základe bežných udalostí z ovládačov publikovať správy, ktoré sa zachytia v komponentoch a následne sa vykoná požadovaná funkcia, ktorá vykoná žiadané zmeny na opačnom konci, v komponentoch na rôznych úrovniach z hľadiska architektúry. Ná základe publikovaného signálu sa vykoná funkcia spolu s parametrami, ktoré taktiež poskytne používateľ z NDMVR. Pre univerzálnosť bol subjekt navrhnutý, aby vzhľadom na variabilitu front-endu bolo možné definovať funkcie pre jednotlivé manažovanie vizualizácií.



Obr. 2.7: Návrh kompozície pre riešenie exekútora funkcií

Na obrázku 2.7 je popísaný koncept takéhoto subjektu, ktorý predstavuje 2 komponenty bez potrebnej komunikačnej väzby. Front-end obsahuje histogramy, ktoré vizualizuje pomocou NDMVR. NDMVR je všeobecný komponent určený na vizualizáciu histogramu, ktorý vizualizuje iba jeden jediný histogram v danom čase. Riešením tohoto problému je možnosť definovania vlastných funkcií, ktoré vykonajú potrebné úpravy na front-ende, podľa toho, ako to programátor front-endu naprogramuje. Funkcie sa poskytnú subjektu, ktorý bude prihlásený na získavanie dát z NDMVR. Na základe publikácie signálov a parametrov subjekt vykoná danú funkciu a tým sa prejaví následne zmena vo vizualizácii.

Pre publikáciu signálov by malo byť možné definovať tieto signály a zdieľať ich medzi obidvoma komponentami pre lepšiu vizualizáciu aj v NDMVR kde je problém zobrazovať konkrétnejšie dáta z front-endu aplikácie. Na tento účel bolo

vytvorené úložisko využívajúce localstorage pre uloženie týchto dát. Dáta budú následne dostupné pre ktorýkoľvek komponent v danom čase.

2.4.2 Import knižnice JSROOT v7.2.1 a vytvorenie súvisiacich funkcionalít

V predošlej verzii knižnice NDMVR a NDMSPC bolo nutné zabezpečiť načítanie objektu JSROOT na ktorom bolo neskôr možné zavolať potrebné funkcie pre prácu so súbormi a vizualizáciami. Knižnica sa pre aplikácie založené na React inštalovala ako npm balík a bolo nutné pre správne importovanie použiť komponent *JsrootContext*, ktorý zabezpečil načítanie celého objektu skôr, než softvérový rámec vytvorí dokument. Následne sa k objektu pristúpilo cez inštanciu *window.JSROOT* a bolo možné používať všetky funkcionality.

V najnovšej verzii sa tento problém vyriešil na strane tímu vyvíjajúceho knižnicu a vo verzii 7.2.1 je možné importovať priamo potrebné funkcie bez akýchkoľvek ďalších krokov. Z minulých verzií sme JSROOT inštalovali v každej knižnici a preto sme pri úpravách inštalovali JSROOT v knižnici NDMSPC-CORE, ktorá sa stala neskôr súčasťou NDMVR a NDMSPC. Knižnica po úprave poskytne priamo okrem svojich vlastných funkcionalít aj všetky potrebné funkcie priamo z knižnice JSROOT, aby sme odbremenili ostatné knižnice od potreby mať túto knižnicu nainštalovanú. Nutnosť bude v novej verzii len inštalovať NDMSPC-CORE a ten poskytne všetko potrebné.

Import JSROOT-u a jeho funkcií však nestačí pre vhodné zapuzdrenie komponentov do jednotných celkov a odstránenie zavislostí sme čo najlepšie potrebovali extrahovať funkcionality, ktoré sa priamo netýkajú charakteru komponentov v danej knižnici.

V prípade NDMVR bol komponent *JsrootHistogram* implementáciou pre vizualizáciu jednorozmerných histogramov len pomocou využitia JSROOT-u. Komponent nemá nič spoločné s A-Frame, vytváral projekcie a aktuálny náhľad z div elementu na panel vo VR na základe identifikátora elementu v DOM. Funkcionalita bola z NDMVR odstránená, knižnica už obsahuje len komponenty súvisiace s vizualizáciou histogramu v rozšírenej realite a distribúciou dát medzi komponentami. Všetky funkcie, ktoré zahŕňal komponent *JsrootHistogram* boli extrahované do knižnice REACT-NDMSPC-CORE. Takisto by knižnica nemala byť závislá na knižnici JSROOT ale priamo na REACT-NDMSPC-CORE.

Knižnica po aplikovaní všetkých vylepšení a úprav importuje knižnice JSROOT, REACT a RxJs a poskytuje všetky potrebné komponenty pre menežment dáto-

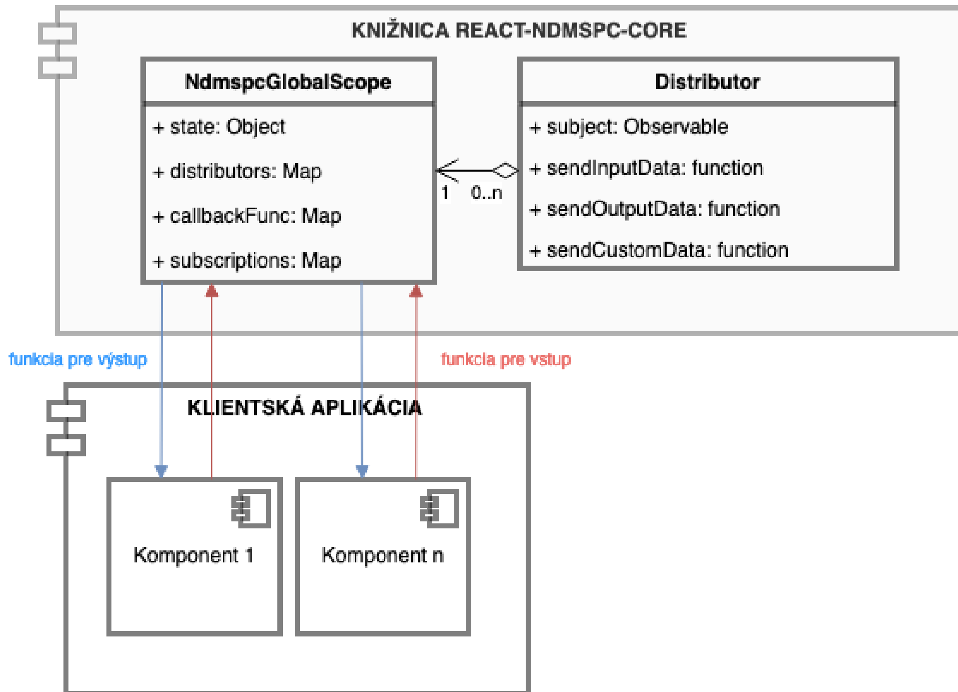
vých analýz a vizualizácií. Knižnica takisto aj presmeruje základné funkcie JSROOT-u a exportuje ich bez zmeny , týmto prístupom neskôr nie je potrebné v knižniciach používajúcich REACT-NDMSPC-CORE inštalovať aj JSROOT a dochádza tak k redukcii závislostí. Viac o riešení s redukciou závislosti v sekcii2.5.

2.4.3 Návrh systému pre distribúciu údajov medzi komponentami

Ako bolo spomenuté skôr pri návrhu REACT-NDMSPC, vizualizačné prostredie pozostáva z viacerých komponentov, ktoré predstavujú väčšie funkčné celky zapúzdrujúce určitú funkcionalitu. Pre vytvorenie komplexného vizualizačného prostredia potrebujeme zabezpečiť aj istú komunikáciu medzi komponentami a to je vzhľadom na rôznorodosť používaných technológií náročne. Pre tento účel bola definovaná nová architektúra, ktorá definuje hlavný objekt v klientskej aplikácii, nazývaný **AppGlobalScope** 2.8, ktorého úlohou je držať stav celej aplikácie.

Tento stav je súbor dát, ktoré je nutné zdieľať medzi komponentami tvoriacimi vizualizačné rozhranie a definovaný objekt predstavuje unikátnu inštanciu v rámci aplikácie a hlavne nezávislú na rôznych moderných webových technológiách. Objekt ďalej obsahuje aj pole komunikačných objektov nazvaných distribútory, ktoré slúžia na nadviazanie spojenia medzi hlavným objektom a konkrétnym komponentom. Každý distribútor umožňuje publikovať dáta a následne ich zachytávať v komponentoch a vykonávať funkcionalitu, ktorú sám používateľ definuje pri dizajne tohoto komponentu.

Z hľadiska typu komunikácie môžeme považovať komunikáciu za obojsmernú čo znamená, že dáta je možné publikovať aj z hlavného objektu **AppGlobalScope** ale aj z koncových komponentov a rovnako je možné aj definovať funkcie, ktoré sa majú vykonať po prijatí týchto dát.



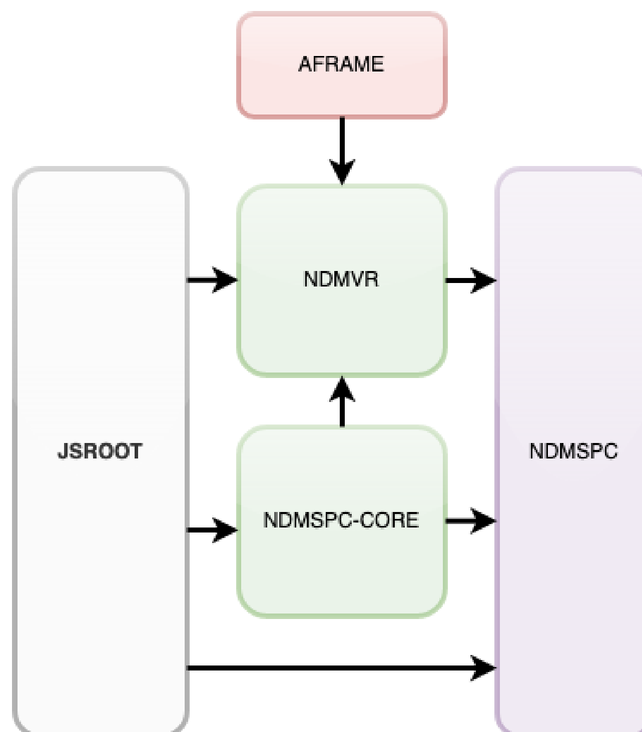
Obr. 2.8: Zobrazenie konceptu hlavného objektu **AppGlobalScope** pre distribúciu dát do vzdialených komponentov

Objekt **AppGlobalScope** pozostáva z hlavných komunikačných objektov, distribútorov. Ako je zobrazené v konceptuálnom návrhu architektúry hlavného objektu 2.8, objekt obsahuje okrem objektu obsahujúceho stav aplikácie aj kolekciu pre evidenciu všetkých distribútorov. Okrem distribútorov si objekt uchováva aj všetky funkcie pre výstup, ktoré sa majú vykonávať pri zachytení údajov z komponentov a v neposlednom rade aj referencie na všetky vytvorené subscriptions.

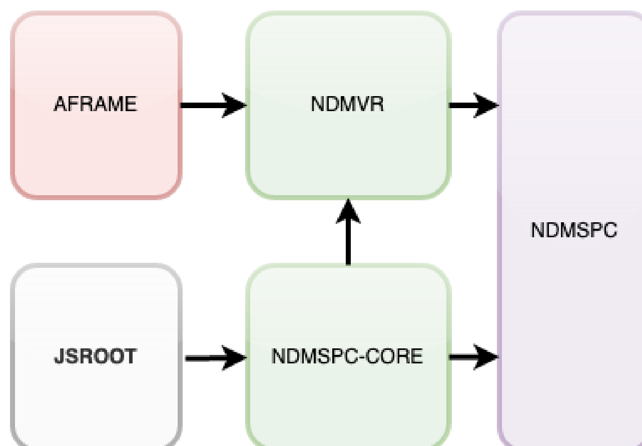
Vzhľadom na to, že objekt môže obsahovať ľubovoľný počet takýchto distribútorov, sú distribútory evidované v mape kde je každý distribútor jednoznačne identifikovaný svojím identifikátorom. Ako aj distribútory, rovnako sú v mapách uložené aj funkcie a takisto aj referencie na komunikačné spojenia. Taktiež sú identifikované identifikátorom, ktorý identifikuje aj samotný distribútor. Na základe tohto prístupu je jednoznačne definovaný distribútor a všetky spojené referencie a aj súvisiace funkcie, čo vedie k prehľadnému a efektívnemu manažmentu týchto atribútov v rámci objektu.

2.5 Usporiadanie funkcionalít a minimalizácia závislostí medzi knižnicami

V tejto kapitole sa na základe predchádzajúcich vylepšení knižníc definujú charaktery prídavných ale aj už existujúcich funkcionalít. Cieľom je vhodne extrahovať funkcionality a umiestniť ich do knižnice, ktorá obsahuje sémanticky funkcie s rovnakým charakterom a ktoré vyžadujú obdobné závislosti. Príklad je možné vymedziť komponent *JsrootHistogram* v knižnici NDMVR, ktorý z hľadiska charakteru, sémantiky a závislostí nemá veľa spoločného s komponentom *NdmVrHistogram3D*. Preto je možné v knižniciach určiť základné závislosti, ponechať komponenty závislé na týchto knižniciach a odčleniť funkcionality tak, ako v prípade NDMVR a komponentu *NdmVrHistogram3D*, ktorý je len implementáciou klasického JSROOT-u a nemá nič spoločné s rozšírenou realitou.



Obr. 2.9: Zobrazenie závislosti medzi knižnicami pred refaktorizáciou



Obr. 2.10: Zobrazenie závislosti medzi knižnicami po preusporiadaní a refaktorizácii

Ako je znázornené na obrázkoch 2.10 a 2.9 najväčší rozdiel je v eliminácii závislostí na knižnici JSROOT, ktorá bola inštalovaná do takmer všetkých knižníc. Ako riešenie bola knižnica JSROOT importovaná do knižnice REACT-NDMSPC-CORE, kde presmerovaním všetkých potrebných objektov boli poskytnuté okrem vlastných definovaných modulov aj modely JSROOT-u. Po úprave už postačí importovať knižnicu REACT-NDMSPC-CORE a tá poskytne aj všetky potrebné moduly JSROOT-u.

3 Implementácia

Z návrhu aplikácie vyplynulo viacero úloh, ktoré je potrebné implementovať pre zabezpečenie požadovanej funkcionality. Pre vytvorenie finálneho riešenia bolo potrebné pracovať na 3 rozličných knižniciach, ktoré obsahovali rozličné funkcionality. Radu bolo potrebné implementovať, nejakým spôsobom upraviť alebo refaktorizovať kvôli lepšej kvalite kódu.

- v prvej fáze boli implementované kritické funkcionality vzhľadom na naše požiadavky v knižnici NDMVR
- v druhej fáze boli implementácie zamerané na knižnicu NDMSPC-CORE, v ktorej boli doplnenie funkcionality z NDMVR závislé iba na knižnici JS-ROOT
- v poslednej fáze boli implementácie zamerané na vývoj knižnice NDMSPC, ktorá obsahuje finálny komponent, ktorý nám zabezpečí splnenie všetkých požiadaviek. Knižnica pre svoje fungovanie priamo využíva vyššie spomínané knižnice NDMVR a NDMSPC-CORE ako závislosti.

3.1 Implementácia vylepšení knižnice NDMVR

Pre to aby sme docielili požadované ciele je nutné vylepšiť a prispôsobiť tak knižnicu NDMVR požiadavkám. V neskoršej fáze pri implementácií NDMSPC kde je predpoklad použitia tejto knižnice ako závislosti tak sme museli vhodne rozšíriť knižnicu o potrebné funkcionality a odstrániť nežiadúce správanie komponentov a chyby.

3.1.1 Refaktorizácia NDMVR na moduly

V počiatočnej fáze návrhu a implementácie bolo nutné refaktorizovať daný komponent a vyriešiť všetky problémy, ktoré boli zistené počas testovania a používania aplikácie. Najdôležitejšie bolo odstrániť problémy vznikajúce prevažne pri

zmene modov ale aj pri zmene zobrazovaných histogramov, ako aj upraviť samotné komponenty do modulov. Lepšia podpora pre moduly hlavne generátor atribútov pre vizualizované entity oddelený od zobrazovača pre lepšiu prehľadnosť a orientáciu v kóde ale aj pre lepšiu škálovateľnosť.

Jednou z prvých častí analýzy je návrh riešení pre odstránenie nedostatkov knižnice NDMVR. Riešenie tohoto problému spočíva v dôkladnej refaktorizácii celého projektu a štruktúry súborov a adresárov, aby sa v projekte dalo ľahšie orientovať a aby bolo následne možné jednoducho projekt rozširovať. Za týmto účelom sme zvolili väčšiu modularizáciu komponentov v projekte. Ako hlavný bod v tomto prípade kompletne bolo oddeliť zobrazovanie histogramov od logiky generovania samotných histogramov. Aktuálne knižnica obsahovala generátor entít, ktoré sa v zobrazovači (na úrovni React-u) vkladali do scény. Všetky entity boli uchovávané v stavoch čo nebolo najideálnejšie riešenie.

Cieľom bolo zabezpečiť, aby sa v generátore vygenerovala štruktúra so všetkými potrebnými atribútmi pripravená na vytvorenie všetkých potrebných entít a aby sa generovala na separátnom mieste od zobrazovača. Štruktúru predstavuje objekt, ktorý bude uložený v stave React komponentu. Následne sa budú v react komponente zobrazovať potrebné entity pomocou parametrov, čo umožní lepšiu orientáciu v kóde rovnako, ako aj lepší výkon.

Medzi ďalšie úpravy patrí aj redukcia a refaktorizácia nepotrebných stavov v komponente, keďže podľa logiky React-u majú byť v stavoch ukladané iba premenné, od ktorých zavisí stav zobrazovaného komponentu. Dáta, ktoré prichádzajú z vonkajšku komponentu ako *Props* atribúty, nemajú byť ukladané v stavoch ale majú byť súčasťou výpočtov v životnom cykle komponentu. V prípade ukladania obrazu komponentu je vhodné uchovávať tieto obrazy dát, ktoré pri zmenách určitých stavových premenných nemajú vplyv na daný obraz ale iba na vybrané časti finálnej podoby komponentu.

Medzi ďalšie vhodné vylepšenia patria aj členenie objektov na funkcionálne komponenty a komponenty obsahujúce iba vizuál.

3.1.2 Rozšírenie vizualizácie TH1 histogramu

V neposlednom rade zabezpečiť vizualizovať aj dvojrozmerné histogramy a prispôsobiť tomu charakter, umiestnenie objektov. V prípade dvojrozmerného histogramu nebude efektívne použiť priamo funkciu pre generovanie entít 3 rozmerného histogramu.

Pre implementáciu podpory pre zobrazovanie bola vytvorená funkcia na vygenerovanie dátovej štruktúry pre definíciu všetkých kľúčových atribútov dôle-

žitých pre správne umiestenie entít aby to zodpovedalo správnemu rozmiestneniu binov. Funkcia je podobná, ako aj ostatné funkcie pre vygenerovanie viacrozmerných histogramov avšak s menšími rozdielmi. Každý typ histogramu má svoje špeciálne charakteristické prvky vo vizualizácii, ktoré ho odlišujú od ostatných. Medzi špecifické charakteristické prvky jednotlivých vizualizácií patrí napríklad počet generovaných objektov, pridávaním rozmerov nám počet generovaných objektov stúpa exponenciálne a preto jednorozmerný histogram môže byť navrhnutý odlišne, keďže nemusíme riešiť obmieňanie sekcií a vieme zobraziť aj všetky biny. Z toho dôvodu je nutné pristupovať ku každému typu histogramu osobitne a generovať tak dátovú štruktúru v osobitnej funkcii so všetkým potrebným čo k tomu patrí. V prípade histogramov TH1 išlo hlavne o zanedbanie rozmeru definovaného osou y, ktorý bol zanedbaný len po stránke zobrazovania údajov, z hľadiska fyziky a 3D prostredia teda os y zostáva bez pridaného označenia s nastavenou pevnou šírkou. Rovnako je iné aj prostredie zahrňujúce panely a umiestnenie histogramu vo svete.

3.1.3 Rozšírenia distribúcie dát o binoch mimo komponent

JSROOT pri vykreslení histogramu poskytuje možnosť definovania funkcií pre spracovanie udalostí, ako sú klik, dvojklik a hover na bin histogramu. Rovnakú funkcionalitu bolo potrebné zabezpečiť aj v prípade vizualizácie histogramu v rozšírenej realite. Ako bolo spomenuté v analýze 1.2.4 tak po vykreslení histogramu sa vráti objekt kde je možné nastaviť funkcie, ktoré sa majú vykonať následne pri vyvolaní konkrétnej udalosti. Rovnaký princíp bol použitý aj v tomto komponente kde sme do komponentu *NdmVrHistogramScene* pridali vstupné atribúty (V React-e Props), ktoré očakávajú implementované spätne volané **callback** funkcie. Tie sa vykonajú pri vyvolaní udalosti.

V tejto fáze bola prerobená podpora pre kliknutie na bin, ktorá sa musela pomerne zložito implementovať. Knižnica NDMVR obsahovala subjekt **distributor**, na ktorý bolo potrebné implementovať funkciu s potrebnou funkcionalitou a následne nastaviť načúvač (subscription) na dáta, ktoré sa publikujú po kliku na bin a následne zachytia už v nadradenom komponente v našom prípade mimo komponentu *NdmVrHistogramScene*. Následne sa vykoná implementovaná funkcionalita.

Tento systém bol vložený o úroveň nižšie. Keďže sa jedná o takzvaný priestupný bod. Tento načúvač po získaní údajov z vnútra vykonával funkciu, ktorá už konkrétne bola implementovaná. Po presunutí do *NdmVrHistogramScene* je jeho úlohou prijať príslušné údaje po vyvolaní udalosti, zistiť o aký typ udalosti

ide a následne vykonať požadovanú funkciu, ktorá sa posunie do komponentu ako parameter z vonkajšieho prostredia.

Zdrojový kód 3.1: Fragment kódu zobrazujúci proces detekcie údajov po vyvolaní udalosti, rozoznanie typu udalosti a vykonanie prislusnej funkcie.

```
let subscription = binDataDistributor
 .getBinDistributor()
 .subscribe( (binData) => {
   if (binData.data.histogramName === info.name) {
      switch (binData.event) {
           case 'CLICK':
                  onClick(dataForResponse)
                  break;
           case 'HOVER':
                  onHover(dataForResponse)
                  break;
           case 'DB_CLICK':
                  onDbClick(dataForResponse)
                  break;
           default:
                  break;
           }
        } else {
        onExeCommand(command)
        }
})
```

V implementácii 3.1 sa na subjekte **distributor** vytvorí načúvač (subscription), ktorý pri každom prijatí údajov o vytvorenej udalosti vykoná *callback* funkciu. Prijaté dáta reprezentuje objekt **binData**, ktorý funkcia získa ako parameter. Overí sa charakter udalosti, kde overujeme či sa jedná o udalosť vzniknutú pri nejakej interakcii s binom alebo sa jedná o vytvorenie príkazu ,takisto sa porovnáva aj atribút *histogramName*, či ide o histogram na ktorom sa vykonala interakcia.

V prípade interakcie s binom sa konvertujú dáta do objektu v tvare pre vykonanie cieľovej funkcie. V *switch* klauzule sa následne určí typ udalosti a vykoná sa príslušná funkcia aj s príslušnou dátovou štruktúrou.

V opačnom prípade ak ide o príkaz tak sa zase vykoná príslušná funkcia s parametrom pre príkaz (v tomto prípade ide len o názov príkazu, aby následne *callback* funkcia vedela rozoznať o aký typ príkazu ide a vykonať patričnú funkcionalitu). Konkrétne sme implementovali zatiaľ podporu pre príkaz **REFRESH**

(obnovenie) kde sa vykoná funkcionalita pre aktualizáciu pohľadov na paneloch. Tento princíp tak umožňuje jednoduchú škálovateľnosť pre iné príkazy ,nutné je len pridať funkciu pre vytvorenie udalosti a publikáciou údajov pre daný príkaz na hociktorom mieste v komponente.

Na opačnej strane komunikácie sa na rôznych koncoch už len vytvárajú udalosti tak, ako je zobrazené na príkladoch 3.2.

Zdrojový kód 3.2: Fragment kódu zobrazujúci príklad vytvorenia udalosti.

```
// parse intersection data
val binData = {
        event:          // CLICK, DB_CLICK, HOVER
        data:             // data from focused bin
        intersect:  // Z, X , Y
        histogramname:  // current histogram name
}

// click, db_click, hover  event in cursor aframe component
binDataDistributor.sendDataOfSelectedBin(binData)

// command event in keyboard Controller on press Enter
if (event.key === 'Enter') {
  binDataDistributor.emmitCommand('REFRESH')
}
```

Prídavok v knižnici NDMVR je aj rozoznávanie strany binu na ktorú používateľ namieri alebo klikne. Pri vytvorení udalosti, ako je zobrazené vo volaní funkcie *sendDataOfSelectedBin* sa vytvorí údajová štruktúra v závislosti od týchto faktorov. Zistí sa strana, na ktorú sa vytvára udalosť a to na základe osi kde je situovaná strana binu a určí sa aj typ udalosti, keďže funkcia sa vykonáva pre viac typov udalostí. Spolu s údajmi o bine sa vytvorí udalosť a publikujú sa dáta, ktoré následne zachytí načúvač na opačnej strane komunikácie v našom prípade načúvač v komponente *NdmVrHistogramScene* 3.1. Takisto v *keyboardController* je príklad pre vytvorenie udalosti pre príkaz. Po stlačení tlačidla sa vykoná funkcia na subjekte a pošle sa parameter indikujúci názov príkazu.

3.1.4 Rozšírenie pre definovanie vzhľadu binov na základe externej funkcie

Podobne ako v predchádzajúcej kapitole je potrebné zabezpečiť aj úpravu vizualizácie na základe vonkajších premenných mimo vizualizačný komponent *Ndm-*

VrHistogramScene. Na tento účel bol použitý opäť podobný koncept ako v prípade zachytávania udalostí a vykonávania príslušných funkcií definovaných mimo kontext s rozdielom, že v tomto prípade funkcia na základe parametrov a definovanej funkcionality vráti príslušnú návratovú hodnotu pre vizualizačný atribút entity. Volanie spätne volajúcej funkcie nastáva priamo vo fáze generovania, kde funkcia prepíše predvolenú hodnotu atribútu určenú z reálneho objektu histogramu tak, ako je to zobrazené vo fragmente 3.3.

Zdrojový kód 3.3: Fragment kódu zobrazujúci príklad volania funkcie na mofifikáciu atribútov generovaných entít.

```
const defaultBinContent =
    this.#histogram.getBinContent(ix, iy);
const content = this.onBinContentChange
? this.validateProgress(this.onBinContentChange({
  histogram: this.#histogram,
  histogramid: this.#id,
  x: ix,
  y: iy
}))
    : defaultBinContent;
```

V prvej fáze dochádza k overeniu existencie funkcie pre aplikovanie hodnoty, ak funkcia je definovaná, použije sa jej návratová hodnota, ako hodnota obsahu pre bin. V opačnom prípade sa použije predvolená hodnota pre obsah získaná z histogramu. Ako parametre sa použijú:

- objekt histogramu súradnice binu, pre ktorý sa obsah zobrazuje
- poloha binu v rámci histogramu (zložená zo súradníc na definovaných osiach podľa rozmeru)
- identifikátor histogramu pre odlíšenie histogramov

Poskytnuté parametre umožňujú programátorovi mimo kontext NDMVR vytvárať projekčné pohľady závislé na vizualizovaných údajoch a polohy binu separátne. Pomocou unikátneho identifikátora je možné definovať každému binu unikátny identifikátor v širšom kontexte.

3.2 Implementácia vylepšení knižnice REACT-NDMSPC-CORE

V tejto kapitole implementácie bolo cieľom vyvinúť všetky potrebné komponenty a funkcie pre prácu s projekciami týkajúcimi sa JSROOT-u ale aj rôzne iné funkcionality pre distribúciu dát a komunikáciu medzi komponentami. Predovšetkým knižnica poskytuje základné funkcie knižnice JSROOT a funkcie spojené s administráciou root súborov, ako je otváranie súborov, následné čítanie objektov, ako aj vytváranie projekčných pohľadov na základe špecifikovaných atribútov. Okrem funkcií obsahuje aj rôzne objekty určené pre distribúciu údajov z websocketov, presmerovačov ale aj objekty pre distribúcie údajov medzi komponentami.

3.2.1 NdmspGlobalScope - charakteristika a atribúty

NdmspGlobalScope je trieda definujúca objekt, ktorý slúži za účelom vytvorenia globálneho kontextu pre určitý komponent alebo funkcionalitu. Trieda špecifikuje súbor premenných, atribútov, ktoré slúžia pre ukladanie globálneho stavu v rámci kontextu ale aj kolekciu objektov pre vytváranie komunikačných kanálov medzi komponentami. Objekty takéhoto charakteru nazývame dátové distribútory a pri vytvorení spojenia je nutné vytvoriť načúvač na sledovateľnej inštancii objektu, ktorý poskytuje tento distribútor a implementovať funkciu, ktorá bude vykonaná vždy pri zachytení dát na tomto komunikačnom kanáli. Pri nastavení funkcie na sledovateľnom objekte získavame referenciu na túto inštanciu a vždy pri zmene komponentu je potrebné dbať na jej uvoľnenie (dealokáciu) aby bolo zabránené zahĺteniu.

Zdrojový kód 3.4: Štruktúra objektu AppGlobalScope.

```
class AppGlobalScope {
  state = {
    jobs: new Map(),
    bins: new Map()
  };
  distributors = new Map();
  handlerFunc = new Map();
  subscriptions = new Map();
  // funkcie pre menezment atributov objektu
}
```

Ako je definované vo fragmente 3.4 trieda obsahuje objekt **state** so stavom kontextu a následne kolekcie pre ukladanie a manažment distribútorov, funkcií, ktoré sa budú vykonávať pri zachytávaní dát na distribútoroch a vzniknuté inštancie pre zabezpečenie korektnej správy alokácie a dealokácie. Aby sa predišlo voľnému vytváraniu inštancií tohto objektu, je exportovaná jediná inštancia, ktorú je možné použiť pre vytvorenie kontextu objektu. Týmto prístupom bola zabezpečená unikátnosť takéhoto objektu v rámci jedného klientského prostredia, kde bude knižnica nainštalovaná.

3.2.2 NdmspGlobalScope - spracovávanie získaných údajov

Pre spracovávanie údajov bol použitý koncept distribútorov založených na subjektoch knižnice *RxJs*, ktoré ponúkajú možnosť publikácie údajov a následné zachytenie vyslaných dát a vykonanie definovanej funkcie s prijatými dátami. Z hľadiska **NdmspGlobalScope** sú rozlišované 2 typy funkcií pre spracovanie prijatých údajov, a to:

- vstupná funkcia - definuje funkciu, ktorej charakter je prijatie dát ich následné spracovanie a modifikácia stavu kontextu. Referencie pri vytvorení su manažované v rámci samotného objektu.
- výstupná funkcia - definuje funkciu, ktorej charakter spočíva v prijímaní dát vo vzdialených externých komponentoch mimo objekt **NdmspGlobalScope**. Pri vytvorení načúvača na prijaté dáta sa vytvára externá referencia, ktorá je manažovaná už v externom komponente.
- doplnková funkcia - definuje doplnkovú funkciu, ktorá nemá presne stanovený charakter ako vstupná a výstupná funkcia. Jej použitie by nemalo obnášať zložité funkcionality ale iba doplnkové, ako napríklad sústredenie údajov a notifikácie.

Údajové štruktúry, keďže ide o manažment viacerých distribútorov ku ktorým sú viazané funkcie a následne aj referencie, boli zvolené dátové mapy kde každý objekt je asociovaný k istému identifikátoru.

Zdrojový kód 3.5: Nastavenie funkcie pre spracovávanie získaných dát pre vonkajší komponent.

```
createExternalSubscription(
    id,
    callbackOutputFunc = null,
```

```
    callbackCustomFunc = null
  ) {
    return this.distributors
      .get(id)
      .getObservableInstance()
      .subscribe((receivedData) => {
        if (receivedData.flag === 'OUT') {
          if (callbackOutputFunc) {
            callbackOutputFunc(receivedData.data)
          }
        } else {
          if (callbackCustomFunc) {
            callbackCustomFunc(receivedData.data)
          }
        }
      });
  }
```

Fragment 3.6 znázorňuje funkciu, ktorá príjme parametre pre definovanie funkcií pre konkrétny distribútor identifikovaný parametrom *id*. Následne funkcia vykoná ak prijaté dáta obsahujú flag **OUT** a ako parameter sa poskytnú dáta získané z prijatej správy.

Zdrojový kód 3.6: Nastavenie funkcie na spracovávanie získaných dát objekt z vonkajších komponentov.

```
createSubscription(id) {
    const handlerFunc = this.handlerFunc.get(id)
    return this.distributors
      .get(id)
      .getObservableInstance()
      .subscribe((receivedData) => {
        if (receivedData.flag === 'IN') {
          if (handlerFunc?.callbackInputFunc) {
            handlerFunc.callbackInputFunc(receivedData.data)
          }
        } else {
          if (handlerFunc?.callbackCustomFunc) {
            handlerFunc.callbackCustomFunc(receivedData.data)
          }
        }
      })
```

```
}
```

V tomto prípade 3.6 ide o funkciu, ktorá príjme konkrétny identifikátor distribútora na ktorom sa má vytvoriť načúvač na dáta. Pri vytváraní načúvačov pre vstupné funkcie na modifikácie stavov globálneho kontextu sú distribútory uložené v samotnom objekte, aby sa dali následne importovať na ľubovoľnom mieste v aplikácii. Následne sa funkcia vykoná ak prijaté dáta obsahujú flag **IN** a ako parameter sa poskytnú dáta získané z prijatej správy v opačnom prípade sa zavolá doplnková funkcia.

3.2.3 NdmspGlobalScope - dátový distribútor

Zdrojový kód 3.7: Funkcie pre publikáciu dát medzi komponentami kontextu.

```
  sendOutputData(data) {
    this.subject.next({ data: data, flag: 'OUT' })
  };

  sendInputData(data) {
    this.subject.next({ data: data, flag: 'IN' })
  };

  sendCustomData(data) {
    this.subject.next({ data: data })
  };
}
```

V tomto prípade je dôležité zabezpečiť, aby sa dáta publikovali v požadovanej štruktúre a to vo forme objektu obsahujúceho vyššie definovanú štruktúru 3.7. Štruktúra musí okrem samotných dát obsahovať aj flag špecifikujúci charakter prijímajúcej funkcie na opačnej strane komunikačného kanálu.

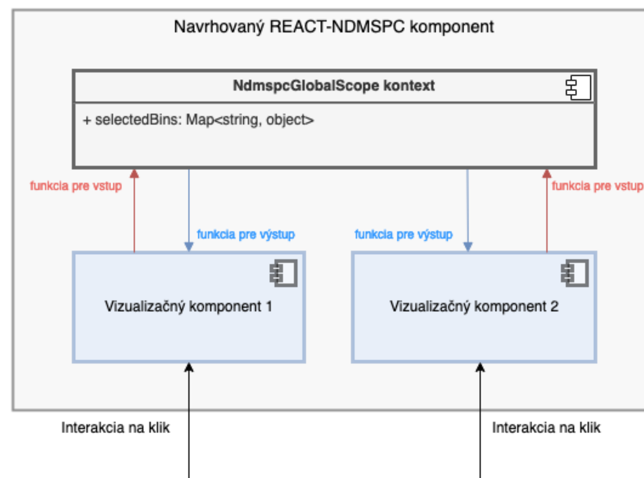
3.3 Implementácia komponentu knižnice NDMSPC

Definícia komponentu knižnice NDMSPC je konkrétny komponent slúžiaci na riešenie analýz a vizualizácií. Zohľadnením používateľskej domény a požiadaviek je možné pomocou vhodných komponentov knižníc REACT-NDMSPC-CORE a NDMVR implementovať takíto komponent.

Keďže implementácia spočíva, ako bolo spomínané z dizajnu a správneho skladania už implementovaných funkcionalít, bude táto kapitola rozdelená do dvoch častí. V prvej bude opísaný príklad definície jednoduchého komponentu na spracovanie jednoduchej interakcie používateľa, uloženie stavu do globálneho kontextu s následným aktualizovaním vizualizovaných scén. Následne v druhej časti bude opísaná už konkrétna implementácia komponentu, ktorý bol vyvíjaný v tejto práci a to komponent na spúšťanie úloh na klástroch klikom a následne monitorovanie týchto úloh, kde sa vizualizuje progres a stav úloh.

3.3.1 Jednoduchý komponent pre selekciu binov

V tejto časti bude popísaný implementačný postup vývoja komponentu na selektovanie binov. Komponent predstavuje 2 vizualizačné komponenty, ktoré vizualizujú 2 dvojrozmerné histogramy typu TH2, každý histogram vizualizuje iný charakter dát ako finálny obsah, avšak ide o totožné histogramy s rovnakým identifikátorom a biny predstavujú rovnakú množinu dát. Používateľ klikom na bin selektuje bin, ktorý sa následne zobrazí s polovičným obsahom oproti bežným obsahom a zmení farbu na farbu selekcie. Všetky selektované biny v histograme zobrazenom vizualizačným **komponentom 1** musia byť zobrazené aj po zobrazení **komponentu 2**.



Obr. 3.1: Zobrazenie základného konceptu vyvíjaného komponentu

V diagrame 3.1 je zobrazená štruktúra celého komponentu, kde môžeme vidieť oba zobrazované komponenty a globálny kontext na zabezpečenie ukladania stavu selekcie binov a komunikácie medzi komponentovými objektami. Komunikácie medzi komponentami sú riešené na základe distribútorov, kde v prípade

toku dát z komponentov 1 a 2 sú prijaté dáta v globálnom kontexte spracovávané **funkciou pre vstup** (červená šípka), kde v opačnom prípade ak je nutné , finálne dáta poskytnúť komponentom pre ich aktualizáciu použitím **funkcie pre výstup** (modrá šípka). Modifikácie stavu kontextu a komponentov by sa nekonali bez interakcií používateľa, preto požadovaná interakcia na selekciu a deselekciu bude **klik** na bin v histograme. Hlavný stav aplikácie obsahuje ukladanie binov do množiny pre naše účely jednoduchého manažmentu binov a administrácie sme zvolili mapu kde každý bin bude obsahovať svoj kľúč v podobe identifikátora a hodnotu, ktorá bude predstavovať ďalšie informácie v našom prípade obsah binu.

3.3.2 Implementácia hlavného kontextu a distribútorov

Po inicializácii webovej aplikácie založenej na softvérovom rámci React a inštalovaní potrebných knižníc je potrebné v prvej fáze vytvoriť globálny kontext pre dané komponenty. Pre kontext bol vytvorený samostatný súbor, ktorý bude exportovať všetky tieto potrebné objekty a funkcie definujúce hlavnú logiku. Súbor bol nazvaný **AppGlobalContext.js** a obsahuje definíciu hlavného objektu **AppGlobalScope** so špecifikovanou štruktúrou stavu tak, ako je to zobrazené vo fragmente 3.8.

Zdrojový kód 3.8: Fragment popisujúci vytvorenie inštancie globálneho objektu spolu s distribútorom.

```
import {
  AppGlobalScope,
  Distributor
} from "@ndmspc/react-ndmspc-core";

// vytvorenie instancie s definiciou struktury stavu
const appGlobalScope = new AppGlobalScope({
    bins: new Map()
})

// vytvorenie a pridanie distributora
appGlobalScope.addDistributor(new Distributor("bin"));
```

Po zahrnutí hlavných tried z knižnice REACT-NDMSPC-CORE bola vytvorená inštancia objektu spolu s inicializáciou stavu danej inštancie. Stav predstavuje mapa kde sa budú ukladať selektované biny histogramu. Následne bolo potrebné vytvoriť aj komunikačný distribútor a pridať ho do objektu pre jednodu-

chý import v kontexte a aby sme vedeli kdekoľvek nadviazať spojenie a vytvoriť komunikačný kanál. Distribútor bol vytvorený a bol mu priradený aj unikátny identifikátor "bin"pomocou ktorého bude možné pristupovať k tejto inštancii.

3.3.3 Implementácia interakcie používateľa

Ďalšou dôležitou časťou je implementácia používateľskej interakcie a to čo sa vykoná a na akú udalosť vyvolanú používateľom. Keďže ide o selekciu binov tak modelová situácia je, že používateľ klikne na bin a v prípade, že bin nie je označený tak ho označí, uloží identifikátor do hlavného objektu, v opačnom prípade sa bin odstráni a odznačí. Ako bolo spomenuté v predchádzajúcich častiach tejto práce tak komponenty na vizualizáciu histogramov JSROOT a NDMVR podporujú možnosti definícií funkcionalít na vyvolávané udalosti na binoch a to **klikmi**, **dvojklikmi** a **hoverom**. Z tohto dôvodu bola implementovaná spätne volaná funkcia, ktorá sa vykoná po kliku na bin v histograme.

Zdrojový kód 3.9: Fragment popisujúci implementáciu spätne volanej funkcie, ktorá sa vykoná po kliku na bin.

```
const handleClick = (data) => {
  appGlobalScope.getDistributorById("bin")?.sendInputData({
     id: `${data.id}-${data.binx}-${data.biny}`,
     data: data
  })
}
```

Sémantika funkcie vo fragmente 3.9 definuje spracovanie dát získaných o bine po vytvorení udalosti klik a následne ich spracovanie, vytvorenie identifikátora binu vzhľadom na identifikátor histogramu (histogram, v ktorom bola udalosť vytvorená) a súradníc binu v danom histograme. Po prijatí dát je potrebné získať z hlavného objektu distribútor a publikovať dáta v požadovanej štruktúre do hlavného objektu **AppGlobalScope** pre spracovanie.

Pre publikáciu na distribútore tohto typu rozlišujeme 3 druhy funkcií, viac bolo spomenuté v sekcii 3.2.2. V tomto prípade je potrebné publikovať dáta z externého prostredia do hlavného objektu, aby sa upravila štruktúra predstavujúca stav označenia binov. Použije sa funkcia *sendInputData*().

Hlavný objekt je možné získať importom zo súboru **AppGlobalContext.js**, kde sme tento objekt implementovali a distribútor následne vieme získať pomocou náležitej funkcie a identifikátora.

3.3.4 Implementácia logiky komponentu

Po implementácii hlavného kontextu a funkcie pre zabezpečenie interakcie zo strany používateľa je potrebné pridať potrebnú funkciu, ktorá spracuje dáta prijaté po vytvorení interakcie kliku na bin a publikácií údajov o bine. V predošlej sekcii 3.3.3 bola popísaná funkcia, ktorá spracuje prijaté dáta do požadovaného tvaru a publikuje ich na danom distribútore. Z tohto dôvodu je potrebné vytvoriť funkciu, ktorá spracuje tieto publikované dáta na opačnom konci spojenia a to priamo v hlavnom objekte **appGlobalScope**. Funkcia sa vykoná iba ak sa zachytia dáta publikované pomocou funkcie *sendInputData*() na danom distribútore.

Zdrojový kód 3.10: Fragment popisujúci implementáciu funkcie pre vstup.

```
// vstupna funkcia pre modifikaciu kontextu
const handleInputViewFunc = (binData) => {
  if (binData) {
      // ziskanie binu z mapy
      const selectedBin =
        appGlobalScope.state.bins.get(binData.id);

      if (selectedBin) {
          // ak existuje odstranenie
          appGlobalScope
               .state
               .bins
               .delete(binData.id);
      } else {
          // ak neexistuje pridanie
          appGlobalScope
               .state
               .bins
               .set(binData.id, binData.data);
      }
      // publikacia oznacenych binov von
      appGlobalScope
          .getDistributorById("bin")
          ?.sendOutputData(
              Array.from(
                appGlobalScope.state.bins.keys()
              )
          );
```

```
  }
};
```

Pre tento účel je nutné zadefinovať **funkciu pre vstup**, ktorá predstavuje funkcionalitu, ktorá sa vykoná po prijatí publikovaných dát. Sémantika funkcie spočíva hlavne v prijatí dát z externého prostredia a vykonanie funkcionalít spojených s modifikáciou svojho stavu, ktorý predstavuje aj stav celého kontextu. Definovaná funkcia prijme ako parameter spracované dáta o bine, overí dostupnosť binu v mape a následne ho tam pridá alebo odstráni.

Po úprave stavu sa následne publikuje aktuálna štruktúra s označenými binmi do externého prostredia. Publikácia modifikovanej štruktúry označených binov bude pomocou funkcie *sendOutputData*.

Hlavný objekt **AppGlobalScope** obsahuje možnosť manažovať komunikačné spojenia pomocou funkcií, preto je možné ukladať tieto distribútory, funkcie a vytvorené referencie na spojenia. Implementovaná **funkcia pre vstup** 3.10 musí byť zaregistrovaná v objekte **AppGlobalScope** a následne musí byť objekt naštartovaný, kde v tomto stave sa vytvoria všetky komunikačné spojenia a spracovávajú sa dáta. Tento proces je názorne zobrazený vo fragmente 3.11.

Zdrojový kód 3.11: Fragment popisujúci implementáciu registrácie funkcií pre vstup a naštartovanie komunikačných spojení.

```
// registracia funkcie pre vstup
appGlobalScope.addHandlerFunc(
  "bin",
  handleInputViewFunc,
  null
);

// nastartovanie komunikacnych spojeni
appGlobalScope.addAllSubscriptions();
```

3.3.5 Implementácia vizualizačných komponentov 1 a 2

Po implementácii všetkých potrebných funkcionalít spojených s interakciami a úpravou stavu je potrebné ešte aplikovať tieto zmeny vo finálnej vizualizácii. Je potrebné zabezpečiť, aby sa štruktúra označených binov zohľadnila pri zobrazovaní histogramu pre používateľa. To zabezpečíme implementáciou vizualizačných komponentov 3.12.

Zdrojový kód 3.12: Fragment popisujúci implementáciu vizualizáčného React komponentu.

```
// import potrebnych objektov a funkcii
import {
 appGlobalScope,
 handleClick
} from "../AppGlobalContext";

// react komponent
const Component1 = () => {
  const [bins, setBins] = useState(
     Array.from(
        appGlobalScope
          .state
          .bins
          .keys()
     )
  )

  // spracovanie dat a editacia stavu komponentu
  useEffect(() => {
      const subscription = appGlobalScope
        .createExternalSubscription(
          'bin',
          (data) => { setBins(data) },
          null
        )
      return () => subscription
                     .unsubscribe()
  })

  // dalsia implementacia
}
```

V komponente je nutné po zahrnutí hlavného objektu **AppGlobalScope** zabezpečiť aktualizáciu stavu komponentu (stav React komponentu, nie stav celého kontextu). To je implementované vytvorením externého komunikačného spojenia s hlavným objektom na distribútore, kde distribútor je získaný z **AppGlobalScope** pomocou identifikátora. Ako spätne volaná funkcia, ktorá bude vykonávaná

vždy pri zaznamenaní dát je jednoduchá funkcia na nastavenie stavu externého komponentu *setBins*, hodnota sa získa publikovaním poľa označených binov v **AppGlobalScope** funkciou **sendOutputData**. Po vytvorení takéhoto spojenia sa uchová referencia na toto spojenie v premennej *subscription* pre manažment tohto spojenia výlučne v externom komponente narozdiel od funkcií pre vstupy, kde sú referencie manažované v objekte **AppGlobalScope**.

Po nastavení stavu komponent uchováva vždy aktuálny stav označených binov a zobrazuje histogram na základe tohto stavu. Na základe stavu komponentu sú definované funkcie, ktoré zabezpečia zmenu požadovaných atribútov binu vo fáze generovania. Ide o spätne volané funkcie ,ktoré sú implementované a zaslané ako parameter do komponentu **NdmVrScene**, ktorý následne funkcie použije v procese generovania entít.

Zdrojový kód 3.13: Fragment popisujúci implementáciu funkcií pre modifikáciu atribútov vizualizovaných komponentov na základe stavu externého komponentu.

```
// funkcia definuje finalny obsah binu
const handleBinContent = (data) => {
  let progress = 4
  bins?.forEach((x) => {
    if (x === `${data.histogramId}-${data.x}-${data.y}`) {
       progress = 2
    }
  })
  return progress
}

// funkcia definuje finalnu farbu binu
const handleBinColor = (data) => {
  let color = null
  bins?.forEach((x) => {
    if (x === `${data.histogramId}-${data.x}-${data.y}`) {
        color = 'red'
    }
  })
  return color
}
```

Sú to funkcie pre nastavenie obsahu a farby 3.14 zohľadnením stavu jednotlivých binov. Funkcie ako parametre získavajú dáta, ktoré definujú objekt histo-

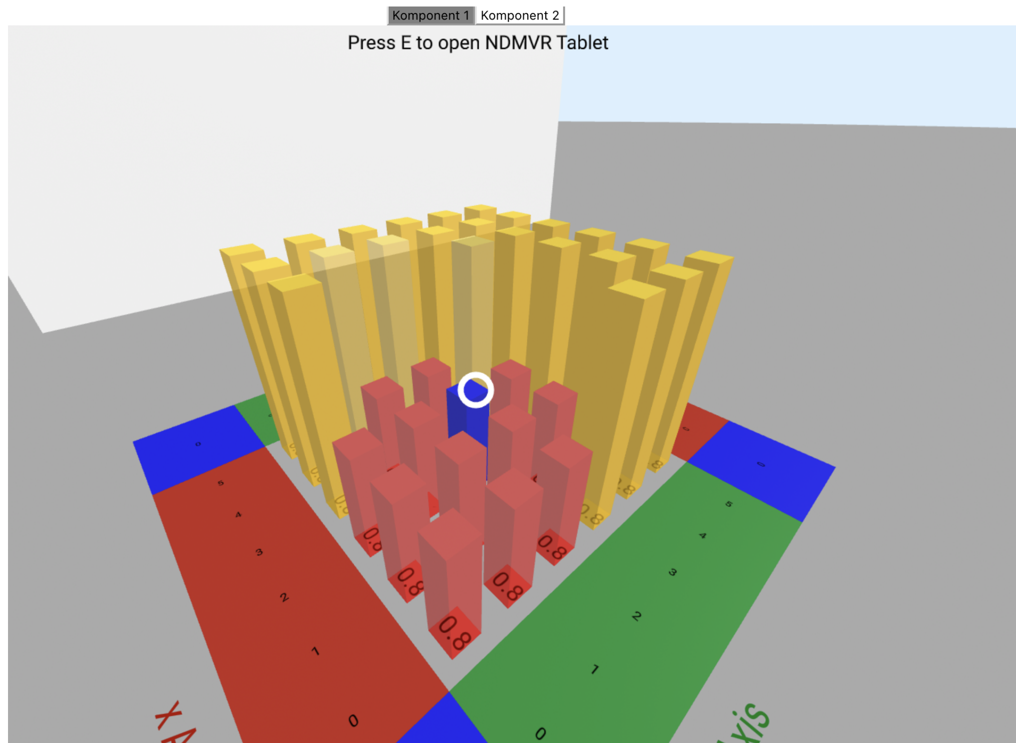
gramu, ktorý sa vizualizuje a aktuálny bin, pre ktorý sa ide vytvoriť vizualizačná entita. Funkcionalita overí na základe týchto získaných parametrov existenciu binu s určitým identifikátorom v stave externého komponentu, ktorý sa získava z **AppGlobalScope** a následne určí hodnotu atribútu.

Ako posledný krok je nutné poskytnúť všetky parametre vizualizačnému komponentu **NdmVrScene**. Rovnako aj funkciu pre spracovávanie používateľských interakcií *handleClick*.

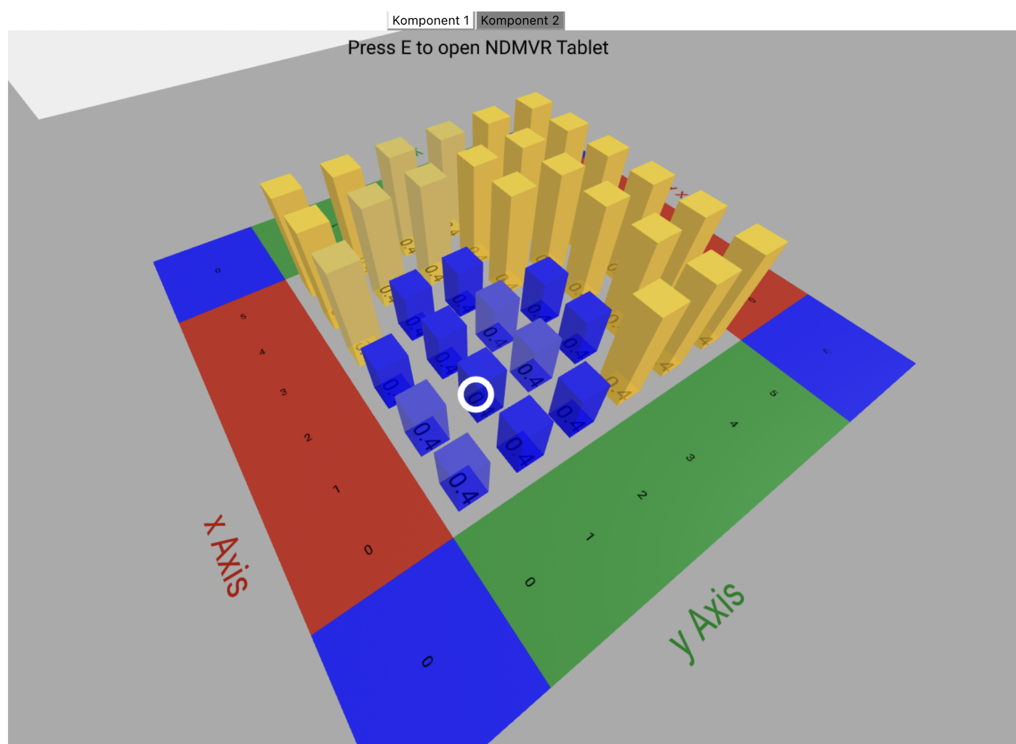
Zdrojový kód 3.14: Fragment popisujúci implementáciu funkcií pre modifikáciu atribútov vizualizovaných komponentov na základe stavu externého komponentu.

```
 // komponent ndmvr
return <NdmVrScene
        data={{
            histogram: histogram,
            id: 'example',
            projectionsNames: [],
            projPanelIds: []
        }}
        onClick={handleClick}
        onBinContentChange={handleBinContent}
        onBinColorChange={handleBinColor}
    />
```

Následne na rovnakom princípe bol implementovaný aj druhý komponent. Histogram má rovnaký identifikátor, čím zabezpečíme rovnaké identifikátory pre biny ale objekt histogramu je rozdielny.



Obr. 3.2: Vizualizácia komponentu 1 s označením binov



Obr. 3.3: Vizualizácia komponentu 2 s označením binov

Ako je viditeľné na obrázkoch 3.2 a 3.3, stav označených binov je zohľadnený v oboch vizualizáciách, rozdiel je však v zobrazovaných dátach, kde sú obsahy

binov rozdielne. Rozdiel je rovnako aj v implementáciách funkcií na modifikáciu týchto atribútov binov.

3.3.6 Implementácia logiky komponentu REACT-NDMSPC vyvíjaného v práci

Rozdiel v tejto implementácii oproti tej prezentovanej, ako príklad je predovšetkým v logike. Okrem faktu, že komponent je zložený z viacerých vizualizačných komponentov a obsahuje aj komponenty pre pripojenie k websocketom pre zabezpečenie získavania informácií o stavoch úloh na klástroch.

Stav reprezentuje uchovávanie mapy binov identifikované taktiež kľúčom (identifikátor binu) a objektom, ktorý už je zložitejší a obsahuje údaje o stave úlohy ako aj progres. Interaktivita pozostáva z možnosti selekcie binu a zobrazenie projekcie na postranný panel avšak iba v prípade ak je dostupná v definovanom súbore na konkrétnej adrese v adresároch s projekciami, ako napríklad pre bin so súradnicami $x=1$ a $y=3$ bude projekcia v príslušnom adresári. V opačnom prípade ak bin nie je na danej adrese, tak používateľ si dvojklikom na daný bin spustí úlohu, ktorá mu vytvorí potrebné projekcie.

Celý proces je následne monitorovaný, kde hlavný kontext spracúva dáta prijaté z websocketu. Prijaté dáta obsahujú údaje o úlohách, ako identifikátor úlohy a údaje o stave a progrese danej úlohy. Tieto dáta sú spracovávané, uložené v hlavnom kontexte a rozposlané v požadovanej štruktúre všetkým komponentom pre aktualizáciu ich vizualizácií.

Používateľ tak vždy po prijatí aktuálnejších dát z websocketu vidí požadovanú zmenu vo vizualizácii, teda progres binu na ktorom spustil úlohu a aj stav ak úloha je už ukončená. V prípade úspešného ukončenia úlohy sa bin patrične zafarbí a používateľ vie klikom zobraziť dané projekcie na bočných paneloch.

3.3.7 Implementácia kontextu

V prvej fáze bolo nutné vytvoriť globálny kontext pre vyvíjaný komponent a počiatočná inicializácia štruktúry stavovej premennej.

Zdrojový kód 3.15: Fragment popisujúci vytvorenie globálneho kontextu.

```
// Vytvorenie globalneho kontextu
const appGlobalScope = new AppGlobalScope({
  jobs: new Map(),
  bins: new Map()
});
```

```
// pridanie distributorov
appGlobalScope
  .addDistributor(new Distributor("view"));
appGlobalScope
  .addDistributor(new Distributor("projection"));
```

Kontext je definovaný ako inštancia objektu **AppGlobalScope** so stavom na selekciu binov a spúšťanie úloh. Spúšťanie úloh je cieľom práve tejto práce kde selekcia binov bude riešená v iných prácach v tejto oblasti vývoja. Komunikácia bola založená na distribútoroch s identifikátormi *"view"* a *"projection"*, kde takzvaný distribútor *"view"* slúžiaci na komunikácie ohľadom zmien pohľadov a vizualizácií dát v hlavnej scéne a teda histogramu. Distribútor s identifikátorom *"projection"* definuje komunikačné kanály za účelom distribúcie a manažment názvov zobrazovaných projekcií na paneloch.

3.3.8 Implementácia používateľských interakcií

V komponente NDMSPC sú rozlišované zatiaľ 2 interakcie a to klik na bin so sémantikou zobrazenia projekcií a dvojklik pre spustenie úlohy na klástri. Pri spustení úlohy sa extrahuje dátová štruktúra priamo z lokálneho úložiska prehliadača kde je uložená, keďže príkazy je možné definovať aj v iných komponentoch aplikácie, tie majú spoločné, že aktuálnu štruktúru uložia na lokálne úložisko prehliadača. Následne pri spustení úlohy sa zavolá potrebná žiadosť, pripoja sa všetky potrebné konfiguračné dáta, získané z lokálneho úložiska (funkcia *getCommand*) a zavolá sa HTTP žiadosť (funkcia *runJobOnSalsa*) tak, ako je to znázornené vo fragmente 3.16. Dátová štruktúra týchto konfiguračných dát bola definovaná podrobnejšie v analytickej časti venujúcej sa tejto oblasti 1.3.

Zdrojový kód 3.16: Fragment popisujúci implementáciu funkcie pre spustenie úlohy.

```
// funkcia na spustenie ulohy
const onDbClick = (data) => {
  const jobAttr = getCommand("ndmspcApp");
  runJobOnSalsa(jobAttr.executorUrl,
  {
      ...jobAttr.curlCommand,
      args: `${data.binx} ${data.biny}`
  })
    .then((res) => {
```

```
        if (res) {
          const job = appGlobalScope
            .state
            .jobs
            .get(res.job);
          if (!job) {
            appGlobalScope
              .state
              .jobs
              .set(res.job, {
                 progress: 0,
                 state: "STARTED",
                 bin: `${data.id}-${data.binx}-${data.biny}`
             });
          }
        }
      })
      .catch((error) => {
        console.log(error);
      });
  }
```

Publikujú dáta o zvolenom bine, z ktorých sa vytvorí identifikátor binu. Identifikátor spolu s dátami zachytí **AppGlobalScope** a vykoná funkciu 3.16, ktorá zavolá http žiadosť, ktorej poskytne dáta potrebné pre spustenie úlohy na klástri. Spustenie úlohy nastane len ak štruktúra ešte neobsahuje úlohu priradenú k zvolenému binu. V opačnom prípade sa neinicializuje keďže mapa ma ako kľúč definovaný identifikátor úlohy, ktorú získa po spustení úlohy z websocketu, ktorý distribuuje dáta o stave všetkých úloh.

3.3.9 Implementácia spracovania dát o stavoch úloh

Po prijatí dát o stavoch úloh z websokcketu sa dáta spracúvajú do určitej formy pre ďalšie použitie, ako napríklad zobrazenia v tabuľke alebo histograme. Táto upravená štruktúra je následne publikovaná na distribútore a spracovaná vo **funkcii pre vstup** viac 3.17. Funkcia prijaté dáta pretransformuje do mapy, uloží do stavovej premennej a následne publikuje funkciou *sendOutputData*().

Zdrojový kód 3.17: Fragment popisujúci implementáciu funkcie pre vstup po prijatí údajov o stavoch úloh z websocketu.

```
 // spracovanie prijatych dat o ulohach
const handleInputViewFunc = (jobs) => {
 if (jobs) {
   const bins = [];
   jobs.forEach((x) => {
     const job = appGlobalScope
       .state
       .jobs
       .get(x[0]);
     if (job) {
       const bin = {
         ...job,
         state: x[4] === x[6]
           ? "SUCCESS"
           : x[5] === x[6]
              ? "FAILED"
              : "PENDING",
         progress: calculateProgress(x)
       };
       appGlobalScope
         .state
         .jobs
         .set(x[0], bin);
       bins.push(bin);
     }
   });
   // publikacia vyslednej struktury ako vystup
   appGlobalScope
     .getDistributorById("view")
     ?.sendOutputData(bins);
   }
 };
```

Mapa sa vytvorí na základe prijatých dát a to tak, že sa k identifikátoru binu priradí identifikátor úlohy, ktorý pridelí monitorovací uzol pri inicializácií úlohy. Následne sa definuje status úlohy ako aj progres úlohy, keďže tieto údaje su dôležité pre správnu vizualizáciu.

3.3.10 Komponent pre vizualizáciu

Princíp je identický ako bol opísaný v sekcii 3.3.5 aj s implementáciou funkcií pre zmenu atribútov. Na základe mapy s binmi sa binom nastaví farba podľa stavu binov a obsah na základe progresu danej úlohy na klástri.

4 Vyhodnotenie

Táto kapitola má za úlohu vyhodnotiť stav dosiahnutého riešenia a naznačiť ďalšie smery, kam je možné posunúť danú implementáciu. Vyvinuté boli predovšetkým funkcie a komponenty v knižniciach **NDMVR**, REACT-NDMSPC-CORE a **REACT-NDMSPC**. Všetky dostupné funkcie je možné použiť pre implementáciu vlastného vizualizačného komponentu, ktorý dokáže splňať všetky požiadavky na vizualizácie v danej doméne dátových analýz. Z tohto dôvodu bude popísaná účelovosť zvolených postupov a architektonických riešení uplatnených pri implementácií funkcií a komponentov.

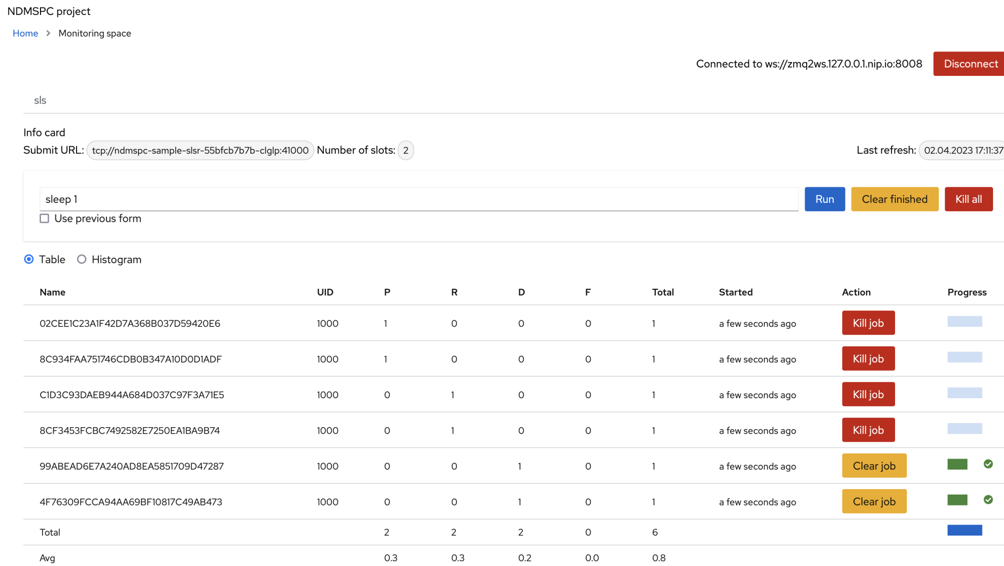
4.1 Vývoj vizualizačného komponentu

V knižnici bol vyvinutý komponent [1] pre jednoduché použitie v ktorejkoľvek klientskej aplikácií za účelom dátovej vizualizácie. V počiatočnej fáze tejto diplomovej práce bol už komponent vyvinutý a obsahoval možnosti pripojenia k vzdialenému websocketu a vizualizáciu údajov získaných z monitorovania úloh na klástri. Následne v tejto práci bola implementovaná podpora pre vizualizáciu týchto dát aj vo virtuálnej realite s dostupnou podporou pre spúšťanie úloh na klástri priamo z prostredia VR.

4.1.1 Monitorovanie úloh na klástri

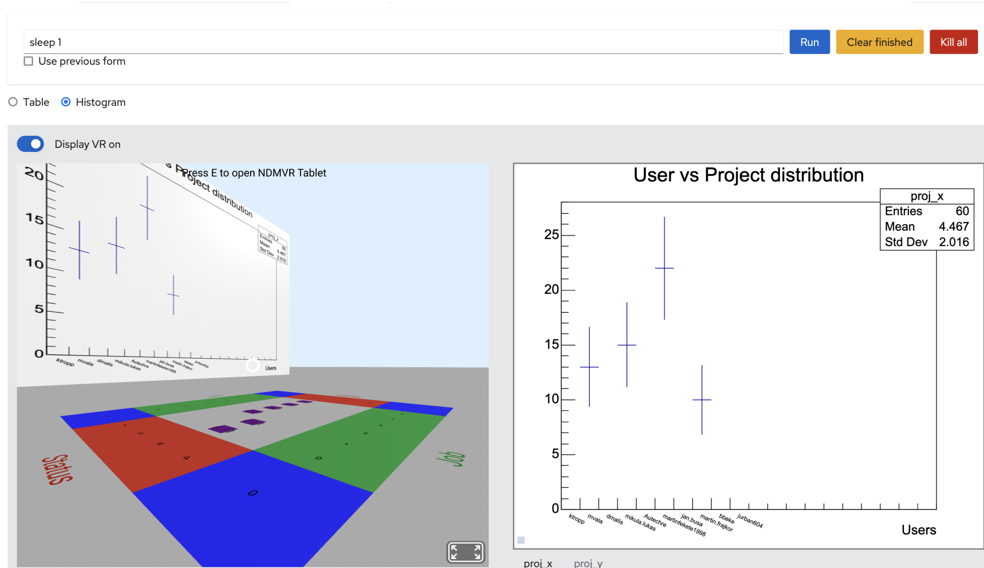
Tento pohľad na stavy úloh 4.1 bol implementovaný v diplomovej práci [32].

[1] `https://ndmspc.gitlab.io/react-ndmspc/`



Obr. 4.1: Zobrazenie stavu úloh vykonávaných na klástri.

Prijaté dáta z websocketu sú transformované a zobrazené v tabuľke. V tejto diplomovej práci bola vizualizácia rozšírená o komponent pre vizualizáciu týchto dát vo virtuálnej realite s využitím komponentu *NdmvrScene*.

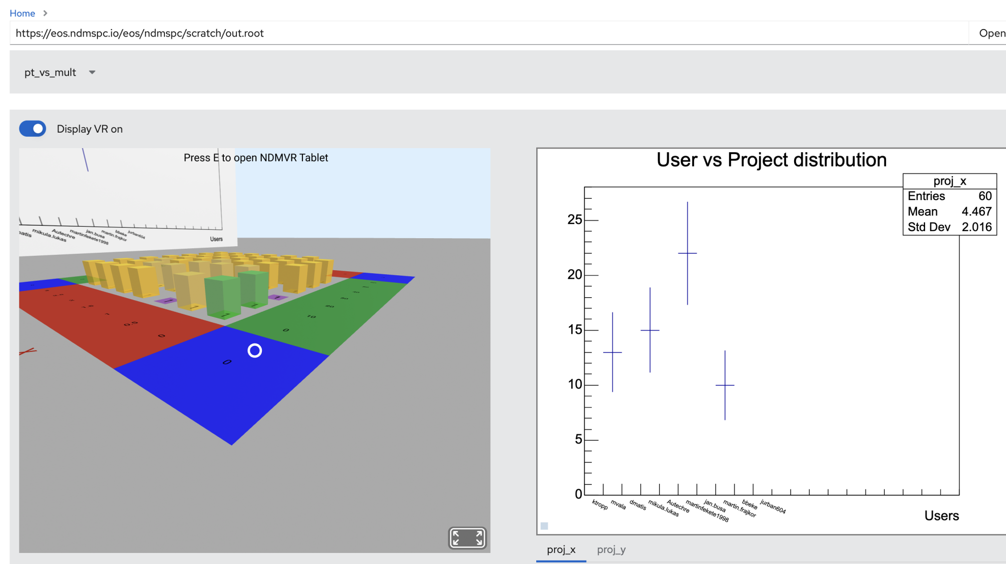


Obr. 4.2: Zobrazenie stavu úloh vykonávaných na klástri vo virtuálnej realite, implementované v tejto diplomovej práci.

Obraz dát z tabuľky na obr. 4.1 je ekvivalentný, ako aj na obr. 4.2, kde sú dáta zobrazené v histograme. V prípade zobrazenia histogramu je v paneli na pravej strane obrazovky priestor pre zobrazenie projekcie.

4.1.2 Prehliadač ROOT súborov

Prehliadač ROOT súborov bol implementovaný za účelom otvárania súborov s určitou štruktúrou, ktorá predstavuje objekty s vizualizovanými dátami (histogramy) a adresárovú štruktúru s projekciami. V tomto prípade projekcia predstavuje taktiež objekt s vizualizovanými dátami (histogram) určený pre každý bin, teda dátovú jednotku z hlavného vizualizovaného objektu.



Obr. 4.3: Zobrazenie vizualizácie histogramu z otvoreného root súboru. V histograme sú zobrazené aj stavy bežiacich úloh reprezentujúce biny na ktorých boli spustené.

Na obr. 4.3 je zobrazené používateľské rozhranie aplikácie prehliadača, kde je zobrazený dvojrozmerný histogram TH2. Zobrazený histogram umožňuje klikom na bin zobraziť projekciu zo súborovej štruktúry pre daný bin a dvojklikom spustiť úlohu, ktorá vytvorí projekciu pre zvolený bin. Na pravej strane je zobrazená projekcia pre zvolený bin. Požadovaná funkcionalita bola implementovaná a je možné spúšťať úlohy z prostredia virtuálnej reality s následným aktualizovaním vzhľadu binov v závislosti na stave úloh. Biny s ukončenou úlohou a úspešne vytvorenou projekciou sú zafarbené zelenou farbou, ako je to zobrazené na obr. 4.3. Biny so stavom **"bežiaci"** predstavuje fialové zafarbenie s 0 výškou, kde výška predstavuje aktuálny progres úlohy.

4.1.3 Konfigurátor histogramov

Nie vždy je možné jednoduché vytvorenie root súboru s požadovanou štruktúrou. Pre prípad, že používateľ si vytvorí vlastné makro, ktoré sa následne spustí

na klástri si používateľ dokáže v tejto karte vytvoriť vlastný histogram z ktorého bude môcť spúšťať následne tieto úlohy aj s patričnými parametrami, ktoré budú predstavovať biny. Tab obsahuje tlačidlo pre otvorenie konfigurácie kde je možné definovať identifikátor histogramu, dôležitý pre správne identifikovanie histogramu v aplikácii a rovnako dôležitý pre správne identifikovanie binov, ktoré su viazané s identifikátormi úloh spustených na klástroch. Takisto obsahuje možnosť správy takýchto histogramov a ich histórie, kde sú uložené všetky potrebné informácie o histograme, ako identifikátor a biny, ktoré boli označené. Táto funkcionalita obsahuje aj možnosti pre správu tejto histórie, a to možnosti triedenia, pridávania, mazania a pod.

V tejto karte však nebola implementovaná funkcionalita spúšťania úloh na klástroch hlavne z dôvodu, že ide o funkcionalitu, ktorá je závislá na ďalšej časti funkcionality konfigurácie spúšťaných makier ako úloh na klástri. Táto funkcionalita bude vyvinutá v ďalších prácach.

4.1.4 Zobrazovanie projekcií

Pre zobrazovanie projekcií bol použitý postup, ktorý bol vyvinutý v predchádzajúcej bakalárskej práci [23]. Zobrazenie JSROOT vizualizácie na postrannom paneli funkciou *draw* alebo *redraw* a následne získanie náhľadu vo forme PNG obrázku a následné zobrazenie textúry na entite vo virtuálnej realite.

Pre prípad, že bin obsahuje viacero projekcií je nutné ich taktiež zobraziť. V tejto práci bol vyvinutý komponent ktorý rieši tento problém použitím tabov, kde je v danom čase zobrazená len jedna projekcia na paneli a používateľ si môže prepínať tieto projekcie. Nevýhoda je hlavne v absencii všetkých zobrazených projekcií v rovnakom čase, čo môže spôsobiť problém v istých analýzach, kde je potrebné porovnávať údaje na projekciách. Ďalšou nevýhodou je aj implementácia komponentu NDMVR a zobrazovanie týchto projekcií na entitách vo virtuálnej realite. Komponent **NdmvrScene** obsahuje viacero entít na to určených (konkrétne 4 entity). Nutnosť je však mať všetky projekcie zobrazené na webe v príslušných elementoch, aby bolo možné získať textúru náhľadu.

V tejto sekcii je viacero možností na zlepšenie tohto vizualizačného komponentu projekcií ale takisto aj zlepšenie komponentu **NdmvrScene**, keďže v aktuálnej dobe softvérové rámce neponúkali potrebné prostriedky a nástroje na vyriešenie interaktivít na komplexných elementoch v scéne virtuálnej reality.

4.2 Vývoj vizualizačného komponentu knižnice REACT-NDMSPC-CORE

Viaceré funkcionality z NDMVR ale aj REACT-NDMSPC boli vyčlenené a presunuté do knižníc podľa ich vzťahu k funkcionalitám. Funkcionality netýkajúce sa virtuálnej reality boli presunuté do REACT-NDMSPC-CORE, ktorá predstavuje takzvané jadro obsahujúce všetky funkcie knižnice JSROOT, funkcie spojené s komunikáciou, websocketmi ale aj rôzne iné údajové štruktúry. Knižnica obsahovala implementované funkcie a komponenty spojené s websocketmi.

- **createTH1Projection**
- **openTH1Projection**
- **displayImageOfProjection**
- **readProjectionFromFile**
- **AppGlobalScope**
- **Distributor**
- **StreamBroker**
- **WebSocketStreamBroker**

Počas tejto práce boli implementované alebo presunuté funkcie pre prácu s histogramom, projekciami a ich manažmentom.

Vyvinutá bola aj funkcia pre prácu s kontextovým menu poskytujúca možnosť implementácie vlastného kontextového menu pri udalostiach na binoch, kde sa ako akcie na voľbu položky definujú vlastné spätne volané funkcie.

V poslednej fáze bol vyvinutý aj aplikačný globálny objekt pre vytváranie kontextov pre komponenty. Tento objekt slúži pre vytváranie globálnych stavov, umožňuje spravovať komunikačné distribútory. Pri posielaní dát z komponentov a ich prijímaní umožňuje vykonávanie spätne volaných funkcií, ktoré si môže definovať samotný developer daného komponentu podľa potreby v danej oblasti analýz. Komponent je univerzálny a pri vývoji bol kladený dôraz hlavne na to, aby bolo možné použiť tieto objekty a funkcie v rôznych oblastiach analýz, kde sú vizualizácie založené na funkciách JSROOT. Objekt nie je priamo závislý na knižnici JSROOT, preto je využiteľný v rôznych projektoch s nutnosťou inštalácie REACT-NDMSPC-CORE.

Aj keď je aktuálne knižnica založená na softvérovom rámci React, tak knižnica obsahuje predovšetkým implementované hooky, ktoré vyžadujú práve závislosti tejto knižnice. Viaceré funkcie definujúce funkcionality s vizualizáciami dát a globálny kontext, tak nie sú závisle priamo na knižnici React, čo bol taktiež zámer pri vývoji týchto funkcionalít.

4.3 Vylepšenia komponentu NdmvrScene

Komponent bol vyvinutý v bakalárskej práci [23], kde boli položené základy celej vizualizácie histogramov vo virtuálnej realite. V tejto práci bola aplikovaná refaktorizácia knižnice, kde bol upravený koncept generovania komponentu.

4.3.1 Generovanie entít histogramu

Cieľom bola separácia funkcionalít generovania atribútov histogramu a viazanie týchto hodnôt do entít, čo v predošlej verzii knižnice nebolo takto implementované. Nevýhoda bola hlavne v uchovávaní stavov, kde sa ukladali celé elementy vo VR, čo nebolo ideálne a absentovala aj univerzálnosť tohto riešenia. Po refaktorizácii bola zabezpečená separácia, kde inštancia generátora atribútov entít vytvára dátovú štruktúru s týmito atribútmi pre entity. Po vygenerovaní sa aktuálna štruktúra uchováva ako stav komponentu a React komponent už len vygeneruje potrebné entity s atribútmi zo stavu. Inštancia generátora je nezávislá na technológii, predstavuje ju inštancia triedy s atribútmi a implementovanými funkciami pre generovanie.

4.3.2 Modifikovanie atribútov z vonkajšieho kontextu

Pre zabezpečenie zmeny vybraných atribútov pri vizualizácii histogramu bol použitý princíp implementovanej spätne volanej funkcie vo vonkajšom kontexte a jej volanie v určitej fáze generovania. Pri modifikáciách atribútov ako sú výšky a farba binov na základe hodnôt získaných z websocketov boli implementované spätne volané funkcie, ktoré ako parametre poskytovali objekt histogramu spolu s aktuálnymi súradnicami binov v histograme. Na základe tejto implementácie funkcie vrátia adekvátnu hodnotu farby a obsahu pre daný bin.

Nevýhodou je hlavne volanie pomerne náročnej operácie pri definovaní atribútu každého binu, keďže sa funkcia vykonáva pre každý bin. Z opačného hľadiska je tento princíp univerzálnejší, keďže developer nie je viazaný, a pre definovanie finálneho atribútu má k dispozícií všetky dáta obsiahnuté v histograme.

Ako ekvivalentom tejto implementácie by mohlo byť aj definovanie poľa obsahov alebo farieb binov, kde by sa komponentu poskytlo len pole s označenými binmi a v generátore by sa tak overila existencia binu a určila špecifická hodnota atribútu. Hodnotu tohto atribútu by sme museli definovať už v samotnom generátore a bola by rovnaká pre každý bin, čo nám značne uberá možnosti pre prispôsobovanie ale ušetril by sa výpočtový výkon, keďže by nebolo nutné vykonávať toľko funkcií pri jednom cykle generovania entít.

Použité riešenie bolo zvolené na základe faktu, že celkovo sme boli obmedzení v počte binov, ktoré sú zobrazované. Pri vizualizácii histogramu nie je možné zobraziť veľké množstvo binov v rovnakom čase, preto bolo implementované rozdeľovanie do menších rozsahov. Je zabezpečené zobrazenie menšieho počtu binov v danom čase a bolo možné preto poskytnúť väčšiu voľnosť pri implementácii projekčnej funkcie pre daný atribút binu.

V tejto oblasti je obrovský potenciál aj pre budúce práce, napríklad v definicii projekčných funkcií pre dané atribúty rovnako, ako aj implementácie podpory pre modifikácie ďalších atribútov.

4.3.3 Definovanie spätne volaných funkcií pre spracovanie udalostí

Možnosť definovať vlastné spätne volané funkcie na spracovanie udalostí vyvolaných používateľom. Použitý je prístup identický, ako bol implementovaný už v knižnici JSROOT pre definíciu spracovania udalostí nad binmi.

V knižnici **NDMVR** bola vytvorená podpora pre spracovanie udalostí, ktoré podporuje aj JSROOT. Naviac boli implementované aj udalosti vyvolané po stlačení špecifických kláves nazvaných, ako **príkazy**, kde je aktuálne podporovaný *"REFRESH"* po stlačení klávesy **ENTER**, ktorý vyvolá udalosť a vykoná sa spätne volaná funkcia pre tento príkaz.

V prípade implementácie týchto spracovaní udalostí v knižnici **NDMVR** bolo možné využiť špecifický charakter a výhody ponúkané prostredím virtuálnej reality. 3D prostredie nám umožní rozčleniť klasický **klik**, **hover** a **dvojitý klik** na bin do 3 stavov na základe kurzora a strany ktorú pretína. To nám poskytuje viac interakčných možností a absentuje to v klasickej implementácii JSROOT-u, kde je možné na jednu udalosť vygenerovať iba jeden stav.

4.4 Zhrnutie prínosu riešenia a ďalšie rozšírenia

V porovnaní s predchádzajúcimi stavmi knižníc a komponentov, ktoré boli vyvíjané v skupine **NDMSPC**, bola v tejto práci vyriešená hlavná otázka, ktorá bola definovaná už v prvej fáze vzniku celej myšlienky a následne skupiny **NDMSPC**. Otázka, ktorej znenie bolo hlavne možnosť spúšťať úlohy jednoduchým klikom na entitu alebo element a následne vidieť všetky potrebné progresy a stavy daných úloh a to všetko len jednoduchým **klikom**, **hoverom** alebo **dvojklikom**. To, že hlavná funkcionalita bola vyriešená a funguje, predstavuje len začiatok implementácií. Najväčší potenciál spočíva vo vytváraní vizualizačných komponentov pre konkrétne domény využívajúce dátovú analýzu na základe funkcionalít vyvinutých v tejto práci ale rovnako aj vylepšovanie a rozširovanie funkcionalít, určených na implementáciu a dizajn týchto komponentov.

Celkovo boli v tejto práci vyvinuté funkcionality, ktoré umožňujú dizajnovať vizualizačné komponenty pre konkrétne domény, preto medzi oblasť pre pokračovanie prác patrí rozširovanie týchto knižníc o iné funkcionality, ktoré neboli odkryté pri implementáciách v tejto práci, ako aj rozširovanie knižnice **NDMVR** o ďalšie funkcionality JSROOT-u. V spojení so spúšťaním úloh je tam potenciál na ďalší vývoj a ladenie štruktúry ukladania údajov o úlohách, takzvané rozdelenie úloh na podúlohy s ich následným spúšťaním vo väčších celkoch rovnako, ako aj vývoj ďalších funkcionalít spojených so spúšťaním úloh na viacerých binoch naraz a aj s prípadným plánovaním samotného spúšťania. Množstvo vylepšení vyvinutých v knižnici **NDMVR** bolo v tejto práci testovaných iba na režimoch spúšťaných na klasických počítačoch, čo otvára aj priestor pre detailnejšiu prácu na UX pri použití špeciálnych zariadení (VR okuliare a podobne).

5 Záver

Cieľom tejto práce bolo preskúmať možnosti spúšťania úloh na klástroch, priamo z vizualizačného prostredia jednoduchými udalosťami, vyvolanými používateľom v procese dátovej analýzy v prostredí virtuálnej reality. Vizualizačné prostredie predstavovalo virtuálnu realitu, v ktorej boli analyzované dáta prezentované vo forme jednorozmerných a dvojrozmerných histogramov, kde základná požiadavka predstavovala možnosti inicializácie úloh ale aj správne zobrazenie ich výsledkov pre možnosti ďalšieho postupu v analýze dát. K dispozícii boli už vyvinuté riešenia pre komunikáciu s dávkovacími systémami, komponenty určené pre vizualizáciu dát vo virtuálnej realite vo forme histogramov a rovnako aj ekvivalent tejto vizualizácie v dvojrozmernom prostredí (JSROOT).

Dôkladná analýza spočívala v získaní dostatočných poznatkov o existujúcich vizualizáciách vedeckých dát vo virtuálnej realite, predovšetkým ich forma, riešenie interakcií a porovnanie výhod a nevýhod, aby bolo neskôr vo fáze návrhu riešenia možné, čo najlepšie navrhnúť architektúru riešenia. V ďalšej fáze boli podrobne analyzované knižnice NDMVR, REACT-NDMSPC-CORE a REACT-NDMSPC obsahujúce už implementované množstvo komponentov a funkcií vyvinutých v predošlých prácach v tejto oblasti. NDMVR obsahujúci komponent pre vizualizáciu histogramu vo virtuálnej realite, REACT-NDMSPC-CORE obsahujúci funkcie a nástroje potrebné k manipulácií s vizualizáciami s integráciou JSROOT-u a nakoniec aj vizualizačný komponent NDMSPC obsahujúci prepojenie s dávkovacím systémom, spolu so zobrazením údajov o stavoch spustených úloh na klástri v klasickom prostredí webovej aplikácie. V závere analýzy bolo analyzované rozhranie na inicializácie úloh na klástroch a nástroje pre optimálne zostavovanie knižníc, kde bolo potrebné vhodne reagovať na nekompatibilitu s knižnicou JSROOT.

Výsledky analýzy odkryli viaceré problémy vrátane spomínanej nekompatibility technológie pre zostavovanie knižníc s novšími verziami JSROOT-u ale aj problémy s distribúciou údajov mimo vizualizačné komponenty a komunikácie medzi komponentami na rôznych architektonických úrovniach. Požiadavkou

bolo vytvorenie komponentu, ktorý umožní inicializovanie úloh na vzdialenom klástri, správne spracovanie údajov o spustených úlohách, získaných z monitorovacej aplikácie, spolu s ich vizualizáciou (v tejto práci bola rozšírená o vizualizácie pomocou JSROOT-u a NDMVR) a nakoniec aj poskytne možnosti pre konfigurácie vlastných dátových štruktúr pre počiatočnú definíciu vizualizácie.

Zohľadnením analyzovaných poznatkov a veľkosti problému, ktorému sme čelili, navrhnuté riešenie predstavovalo 2 časti. Prvá časť definovala početné vylepšenia jednotlivých knižníc a to v podobe úpravy existujúcich funkcionalít (zmena technológie pre zostavovanie aplikácií, integrácia JSROOT-u do REACT-NDMSPC-CORE pre redukciu závislostí) a implementácia nových komponentov a funkcií (AppGlobalScope na vytvorenie kontextu aplikácie, distribútory na zabezpečenie komunikácie ale aj iné funkcionality spojené s vizualizáciami v JSROOT). Druhá časť predstavovala implementáciu konkrétneho vizualizačného komponentu knižnice REACT-NDMSPC podľa definovaných požiadaviek s využitím implementovaných funkcionalít z prvej časti implementácie. V konečnom dôsledku sa nám podarilo vytvoriť takýto komponent, ktorý umožní spustiť úlohu na klástri po kliku na bin v histograme, inicializuje úlohu a následne po pripojení na vzdialený komunikačný kanál (websocket) získava monitorovacie údaje o úlohe, na základe ktorých aktualizuje histogram a vytvorí projekciu, ktorú je možné zobraziť na postranných paneloch.

V porovnaní s počiatočným stavom sa nám podarilo naplniť hlavný cieľ tejto práce, ktorý spočíva v inicializácii úlohy po kliku používateľa na objekt v prostredí virtuálnej reality a následne získanie spätnej väzby z dávkovacieho systému bez nutností opustenia pohlcujúceho režimu virtuálnej reality, čo nebolo možné v predošlej aplikácii bez nutnosti zadávania príkazov mimo tento režim. Takisto sa nám podarilo napĺniť aj druhý cieľ, spočívajúci v monitorovaní týchto úloh a adekvátnej aktualizácií zobrazovaných dát vzhľadom na spracovávané výsledky. Pri naplnení týchto cieľov vzniklo množstvo nástrojov a komponentov, ktoré sú využiteľné pri implementácii iných vizualizačných komponentov.

Prístupy použité pri vývoji funkcionalít boli v záverečnej časti zhrnuté a porovnané s ostatnými možnými riešeniami, kde boli zdôvodnené voľby. Napriek splneniu požadovaných cieľov sa odomklo širšie spektrum možností pre dizajn vizualizačných komponentov pre konkrétne domény, čo môže byť v konečnom dôsledku predmetom ďalších prác. Rovnako vzniká aj potenciál pre dopĺňanie knižníc o nové funkcionality, ktoré vyriešia množstvo problémov a uľahčia tak implementácie v budúcnosti.

Literatúra

1. VALA M., NDMSPC. *salsa*. 2023. Dostupné tiež z: https://gitlab.com/ndmspc/salsa.
2. VALA M., NDMSPC. *ndmvr*. 2023. Dostupné tiež z: https://gitlab.com/ndmspc/ndmvr.
3. VALA M., NDMSPC. *react-ndmspc-core*. 2023. Dostupné tiež z: https://gitlab.com/ndmspc/react-ndmspc-core.
4. VALA M., NDMSPC. *ndmspc*. 2023. Dostupné tiež z: https://gitlab.com/ndmspc/react-ndmspc.
5. LINEV, Sergey. *JavaScript Root github project*. 2023. Dostupné tiež z: https://root.cern/js/latest/examples.htm.
6. MATIS, Dominik. *Webové rozhranie pre systém SALSA*. 2020. Dipl. pr. Technická unierzita v Košiciach. bakalárska práca.
7. MARCOS D. McCurdy D., Ngo K. *Introduction - A-Frame*. 2023. Dostupné tiež z: https://aframe.io/docs/1.1.0/introduction/.
8. SERGEY LINEV, Manraj Singh. *JavaScript Root github documentation*. 2023. Dostupné tiež z: https://github.com/root-project/jsroot/blob/master/docs/JSROOT.md.
9. DANYLUK, Kurtis; ULUSOY, Teoman Tomo; WEI, Wei; WILLETT, Wesley. Touch and Beyond: Comparing Physical and Virtual Reality Visualizations. *IEEE Transactions on Visualization and Computer Graphics*. 2020.
10. VAN DAM, Andries; LAIDLAW, David H; SIMPSON, Rosemary Michelle. Experiments in immersive virtual reality for scientific visualization. *Computers & Graphics*. 2002, roč. 26, č. 4, s. 535–555.
11. MCCORMICK, Bruce H. Visualization in scientific computing. *Acm Sigbio Newsletter*. 1988, roč. 10, č. 1, s. 15–21.

12. RIBARSKY, William; BOLTER, Jay; DEN BOSCH, A Op; VAN TEYLINGEN, Ron. Visualization and analysis using virtual reality. *IEEE Computer Graphics and Applications*. 1994, roč. 14, č. 1, s. 10–12.
13. EL BEHEIRY, Mohamed; DOUTRELIGNE, Sébastien; CAPORAL, Clément; OSTERTAG, Cécilia; DAHAN, Maxime; MASSON, Jean-Baptiste. Virtual reality: beyond visualization. *Journal of molecular biology*. 2019, roč. 431, č. 7, s. 1315–1321.
14. KOUTEK, Cover Design Michal; KOUTEK, Michal. Scientific visualization in virtual reality: Interaction techniques and application development. 2003.
15. BLANC, Thomas; EL BEHEIRY, Mohamed; CAPORAL, Clément; MASSON, Jean-Baptiste; HAJJ, Bassam. Genuage: visualize and analyze multidimensional single-molecule point cloud data in virtual reality. *Nature Methods*. 2020, roč. 17, č. 11, s. 1100–1102.
16. FOUNDATION, Python Software. *Python*. 2023. Dostupné tiež z: `https://docs.python.org/3/`.
17. THE MATHWORKS, Inc. *MATLAB*. 2022. Dostupné tiež z: `https://www.mathworks.com/products/matlab.html`.
18. BEHEIRY, Mohamed El; DAHAN, Maxime; MASSON, Jean-Baptiste. InferenceMAP: mapping of single-molecule dynamics with Bayesian inference. *Nature methods*. 2015, roč. 12, č. 7, s. 594–595.
19. GANGUAGE, ThBlanc. *Ganuage*. 2023. Dostupné tiež z: `https://github.com/Genuage/Genuage`.
20. SCIENCE, CERN accelerate. CERN official site. 2022. Dostupné tiež z: `https://home.cern/about`.
21. MRDOOB. *Three.js*. 2023. Dostupné tiež z: `https://threejs.org/`.
22. MARTIN VAĽA Matin Fekete, Štefan Korečko. VISUALIZATION OF EXPERIMENTAL DATA IN WEB-BASED VIRTUAL REALITY. In: ZAIKINA, Vladimir Korenkov * Andrey Nechaevskiy * Tatiana (ed.). Joint Institute for Nuclear Research, Dubna, Moscow Region, Russia: Proceedings of the 9th International Conference "Distributed Computing, Grid Technologies in Science a Education" (GRID'2021), Dubna, Russia, July 5-9, 2021, 2021, s. 4. 9.
23. FEKETE, Martin. *Vizualizácia experimentálnych údajov v zdieľanej rozšírenej realite a jej používateľské rozhranie*. 2021. Dipl. pr. Košice: Technická univerzita v Košiciach, Fakulta elek- trotechniky a informatiky,

24. TRONCONE, Brian. *rx.js*. 2023. Dostupné tiež z: `https://www.learnrxjs.io/`.
25. INC., Meta. 2023. Dostupné tiež z: `https://reactjs.org/`.
26. JORDAN, Papaleo; SOPHIA, Shoemaker. *React and WebVR using A-Frame*. 2023. Dostupné tiež z: `https://www.newline.co/fullstack-react/articles/react-and-webvr-using-aframe/`.
27. SERGEY LINEV, Manraj Singh. *JavaScript ROOT*. 2023. Dostupné tiež z: `https://root.cern/js/dev/jsdoc/JSROOT.html`.
28. FERRAIOLO, Jon; JUN, Fujisawa; JACKSON, Dean. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington, 2000.
29. *jsroot context menu example*. 2023. Dostupné tiež z: `https://root.cern.ch/js/latest/api.htm#custom_html_context_menu_src`.
30. FOSTER, Ian; KESSELMAN, Carl. Computational grids: Invited talk. In: *Vector and Parallel Processing—VECPAR 2000: 4th International Conference Porto, Portugal, June 21–23, 2000 Selected Papers and Invited Talks*. 2001, s. 3–37.
31. VALA M., NDMSPC. *zmq2ws*. 2023. Dostupné tiež z: `https://gitlab.com/ndmspc/zmq2ws`.
32. MATIS, Dominik. *Riadenie získavania experimentálnych údajov na účely vizualizácie*. 2022. Dipl. pr. Technická unierzita v Košiciach. diplomová práca.

Zoznam skratiek

3D Trojrozmerný.

API Application Programming Interface.

CSS Cascading Style Sheets.

DOM Document Object Model.

HTTP Hypertext Transfer Protocol.

JSON Java Script Object Notation.

REST Representational state transfer.

SVG Scalable Vector Graphic.

UX User Experience.

VR Virtual Reality.

Zoznam príloh

Príloha A CD médium – záverečná práca v elektronickej podobe,

Príloha B Používateľská príručka

Príloha C Systémová príručka