SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 1346

# VIZUALIZACIJA VELIKIH MREŽNIH GRAFOVA POMOĆU WEBGL-A

Oleksandr Ichenskyi

Zagreb, lipanj 2026.

SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 1346

# VIZUALIZACIJA VELIKIH MREŽNIH GRAFOVA POMOĆU WEBGL-A

Oleksandr Ichenskyi

Zagreb, lipanj 2026.

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

Zagreb, 2. ožujka 2026.

# DIPLOMSKI ZADATAK br. 1346

| | |
|---|---|
| Pristupnik: | **Oleksandr Ichenskyi (0036537370)** |
| Studij: | Računarstvo |
| Profil: | Računarska znanost |
| Mentorica: | prof. dr. sc. Željka Mihajlović |

Zadatak: **Vizualizacija velikih mrežnih grafova pomoću WebGL-a**

Opis zadatka:

Jedan od ključnih izazova u grafičkom prikazu podataka je prikaz grafova, posebice velikih mrežnih grafova. Cilj ovog rada je istražiti postojeću programsku podršku za prikaz grafova u web okruženju te razviti vlastito rješenje. Posebno obratiti pažnju na postizanje interaktivnih performansi pri radu s velikim brojem čvorova. Temeljem proučenih tehnologija razviti programsko rješenje prikaza velikih grafova u web okruženju. Razmotriti mogućnosti korištenja GPU jedinice za ubrzavanje prikaza. Na nizu primjera ostvariti ispitivanje dobivenih rezultata. Analizirati i ocijeniti postignute rezultate u odnosu na slične tehnologije. Diskutirati upotrebljivost rješenja kao i moguća proširenja. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 26. lipnja 2026.

*Hvala na podršci i idejama ekipi iz Memgrapha.*

# Sadržaj

# 1. Uvod

Iako malo ljudi razmišlja o grafovima u svojoj svakidašnjici, grafovi su posvuda - jedan od najmoćnijih matematičkih modela koji se koristi za opisivanje odnosa između entiteta se koristi u pozadini mnogih mehanizama i sustava zahvaljujući kojima naše društvo funkcionira. Primjene se protežu od znanstvenih grafova koji čine temelj modernih sustava za pretraživanje i semantičko zaključivanje do bioinformatike gdje čvorovi predstavljaju gene ili proteine, rješavajući problem sinteze novih lijekova i antibiotika. Današnji razvoj interneta i digitalnih platformi obilježava eksponencijalan rast količine podataka koji se prirodno modeliraju grafovima, što nameće zahtjeve za boljim i efikasnijim alatima za vizualizaciju i analizu tih podataka.

Ključni mehanizam za razumijevanje bilo kojih podatkovnih struktura, a tako i grafova, jest vizualizacija. Iako postoji bezbroj alata i algoritama za automatiziranu analizu podataka i detekciju anomalija, vizualizacija omogućava stjecanje intuitivne slike o lokalnoj i globalnoj strukturi grafa, što pomaže u boljem razumijevanju domene problema. Međutim, kako skupovi podataka rastu, vizualizacija većeg broja čvorova i bridova postaje sve zahtjevniji zadatak. U kontekstu interaktivnih prikaza u web okruženju, grafovi s više od nekoliko tisuća čvorova već predstavljaju ozbiljan tehnički izazov i to ne samo po pitanju brzine izrade prikaza, već i po pitanju responzivnosti sučelja pri interakciji s korisnikom.

Odgovor na ove izazove leži u iskorištavanju masivne paralelizacije koju nude grafički procesori (GPU). Za standardiziran pristup GPU hardveru iz preglednika koristi se *WebGL API* ili njegova nasljednička verzija, *WebGL2*, koja donosi niz naprednih mogućnosti bez kojih implementacija mnogih mehanizama potrebnih za rješavanje problema vizualizacije bila bi izrazito otežana. Kombinacija te tehnologije s postojećim mehanizmima za izvršavanje općih matematičkih operacija na grafičkom procesoru (engl. *GP-*

*GPU*) te s algoritmima raspoređivanja čvorova poput Barnes-Hut algoritma je ono što moderne tehnologije nude za efikasnu vizualizaciju velikih grafova direktno u preglednicima.

Kako već postoje brojna rješenja i biblioteke otvorenog kôda koje rješavaju isti problem, ovaj je diplomski rad razvijen u sklopu jedne od takvih biblioteka, Memgraph Orb[1], koja nudi opširan i funkcionalno bogat mehanizam za vizualizaciju grafova na CPU strani. Implementacija svih tih funkcionalnosti na strani GPU bi trebala ponuditi taj isti širok spektar funkcionalnosti uz bolje korisničko iskustvo na velikim skupovima podataka, što je ujedno i cilj ovog rada.

# 2. Pregled područja

Kako vizualizacija grafova nije samo zanimljivo i vizualno fascinantno istraživačko područje nego i interdisciplinaran kompleksan alat koji spajanjem teorije grafova, računalne grafike i drugih disciplina pruža rješenja za mnogobrojne znanstvenoanalitičke izazove, vrlo je važno razumjeti ključne koncepte i pojmove za razumijevanje ovog područja. Kako je vizualizacija grafova sučelje između čovjeka i računala, pregled potencijalnih primjena i korištenih apstrakcija pojednostavljuje probleme i izazove koje ovaj alat namjerava riješiti.

## 2.1. Grafovi i njihova primjena

### 2.1.1. Teorijska osnova

Iz teorije grafova imamo definiciju grafa kao matematičke strukture $G = (V, E)$ koja se sastoji od nepraznog skupa čvorova V i potencijalno praznog skupa bridova $E \subseteq V \times V$ koji predstavljaju odnose između čvorova. Iako je već i ovaj matematički model izuzetno moćan i pogodan za rješavanje širokog spektra problema, ništa nas ne sprečava da bridovima i čvorovima pridružujemo neka dodatna svojstva i atribute što čini ovu apstrakciju još izražajnijom. Na primjer, bridove možemo učiniti usmjerenima ili neusmjerenima, a čvorovima pridružiti težine. Upravo takav model grafa s proizvoljnim skupom parova ključ-vrijednost na svakom čvoru i bridu se koristi u sustavima za upravljanje mrežnim podacima kao što su graf baze podataka, na primjer Memgraph.

### 2.1.2. Domene primjene

Nakon definicije ovog jednostavnog ali stoga ne manje zanimljivog matematičkog modela postavlja se pitanje - zašto to postoji? Jedina granica odgovora na to pitanje je ljudska

imaginacija jer se domene primjene protežu od problema koji se prirodno nameću poput transportne i infrastrukturne mreže sve do računalne sigurnosti i biomedicine[2].

Dok je korištenje takvih struktura u transportnim mrežama trivijalno jer se modeliraju direktnim mapiranjem lokacija u čvorove te cesta u bridove, problemi poput kibernetičke sigurnosti nisu toliko intuitivni i zahtijevaju dublje razumijevanje domene primjene. U nastavku su upravo opisane neke od najvažnijih kategorija primjene relevantne za vizualizaciju velikih mrežnih grafova.

1. Analiza društvenih mreža - iako jedan od najjednostavnijih i najraširenijih primjera korištenja grafova, dobro se generalizira na analizu podataka kao područje. Mapiranjem entiteta (koji mogu predstavljati fizičke osobe, organizacije, objave i slično) na čvorove i mapiranjem interakcija (prijateljstva, praćenja i slično) nastaju grafovi koji se mogu koristiti za analizu tendencija i trendova na kojima se mogu temeljiti algoritmi preporučivanja sadržaja u društvenim mrežama. Dodatno, kvalitetnom vizualizacijom takvih mreža možemo jednostavno uočiti trendove u podacima zahvaljujući prirodnoj strukturi takvih skupova podataka gdje mali broj čvorova ima visok stupanj (puno susjednih čvorova), a velik broj čvorova ima mali stupanj.

2. Modeliranje složenih protein-protein interakcija unutar stanica u bioinformatici i biomedicini pomaže u efikasnijem otkrivanju lijekova te se često koristi kao memorijski sloj za učenje modela strojnog učenja. Kako grafovi u ovoj domeni znaju sadržavati stotine tisuća čvorova uz veliku gustoću bridova, efikasna vizualizacija ovih mreža postaje ključna za uočavanje i analizu uzoraka koji mogu pomoći u boljoj segmentaciji podataka za ih daljnje korištenje.

3. Otkrivanje anomalija u mrežnom prometu i vizualizacija napada na infrastukturu su još jedan primjer problema koji se prirodno formuliraju grafovskom strukturom. Intuitivno, čvorovi odgovaraju uređajima ili domenama, dok bridovi predstavljaju tokove podataka između entiteta čime se ostvaruju efikasne i ljudski čitljive vizualizacije koje pomažu u analizi sigurnosnih incidenata i poboljšavanju računalne sigurnosti sprečavanjem DDoS i drugih vrsta napada.

### 2.1.3. Izazovi

Iako rješenja ovih i brojnih drugih problema u velikoj mjeri beneficiraju korištenjem grafova i njihove vizualizacije, ovo je područje dovoljno kompleksno da ima brojne i složene izazove. Definicija velikog grafa nije apsolutna te ovisi o raspoloživoj računalnoj snazi, ciljevima vizualizacije te gustoći grafa. Generalno, grafovi s više od nekoliko stotina čvorova i bridova već zahtijevaju kvalitetne algoritme raspoređivanja te druge kanale prijenosa informacija poput boje i/ili veličine jer inače u većini slučajeva nemaju perceptivnu vrijednost. Grafovi još veći od toga, više od tisuću čvorova pa sve do gigantskih grafova od desetaka tisuća do milijuna čvorova zahtijevaju GPU podržanu vizualizaciju te odgovarajući hardver kako bi se postigla interaktivna brzina prikaza. Važno je imati na umu da se za kvalitetan i udoban prikaz svaki čvor i brid treba iscrtati 60 puta u sekundi - samim time, osim hardvera, oslanjamo se i na tehnike poput level-of-detail, klasteriranja i fokus+kontekst filtriranja. Neke od ovih tehnika ćemo ne samo spomenuti, već i objasniti princip njihova rada i implementacije u idućim poglavljima.

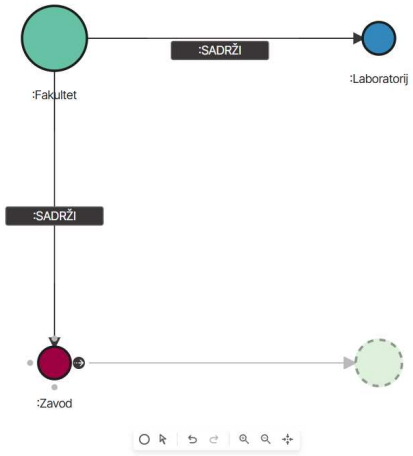
Temeljan izazov u vizualizaciji velikih mrežnih grafova je balansiranje između čitljivosti i skalabilnosti. Čitljiv prikaz zahtijeva da čvorovi budu prostorno ili vizualno (na primjer, bojom ili veličinom) odvojeni, da se bridovi sijeku što rjeđe, jer je u praksi skoro nemoguće dobiti graf bez presijecanja bridova, te da cjelokupan prikaz odražava tražene topološke karakteristike grafa. Kako raste veličina grafa, ovaj kriterij je sve teže udovoljiti, pa je s povećanjem skalabilnosti alata za vizualizaciju važno očuvati i adekvatnu čitljivost grafa korištenjem prije spomenutih tehnika. Stoga, suvremeni pristupi vizualizacije kombiniraju GPU-akceleriran prikaz sa složenijim tehnikama i algoritmima raspoređivanja kako bi se postigao traženi balans.

## 2.2. Vizualizacija grafova u web okruženju

Činjenica da se preglednici danas koriste za kompleksne stvari poput 3D grafike i vizualizacije podataka je rezultat tehnološke evolucije koja je prošla kroz nekoliko jasno razgraničenih generacija izrada prikaza. Svaka generacija, iako i je donosila brojne benefite i rješenja za određene probleme, istovremeno je unosila i svoju specifičnu arhitekturnu kompleksnost. Jasno je da se te tehnike mogu primijeniti i u kontekstu vizualizacije grafova što ih nama čini zanimljivima.

### 2.2.1. *SVG*

Prva tehnika koju ćemo razmotriti je *Scalable Vector Graphics (SVG)*, format za opis dvodimenzionalne vektorske grafike koji je od 2011. godine nativno podržan u svim glavnim web preglednicima. Taj se model temelji na *Document Object Model*-u (*DOM*), odnosno na stablu elemenata stranice, što znači da svaki vizualni element grafa, odnosno svaki čvor i svaki brid, svaka tekstualna oznaka postoji kao zaseban *DOM* element kojim upravlja preglednik. Iako ova metoda ima brojne prednosti zahvaljujući nativnoj podršci *CSS* stiliziranja i *JavaScript* događaja, što omogućuje nativno korištenje vizualno skalabilnih elemanata u toku stranice, poznata je po svojim optimizacijskim problemima. Kako svaki *DOM* element ima određeni memorijski trag, a svako, čak i najmanje, ažuriranje *DOM* stabla može izazvati ponovni izračun raspoređivanja elemenata i ponovno iscrtavanje dijela ili cijele stranice, tako ova metoda postaje neupotrebljiva za vizualizaciju velikih grafova, iako se još uvijek može koristiti za manje interaktivne prikaze.




**Slika 2.1.** Interaktivni SVG editor graf shema

### 2.2.2. *Canvas 2D API*

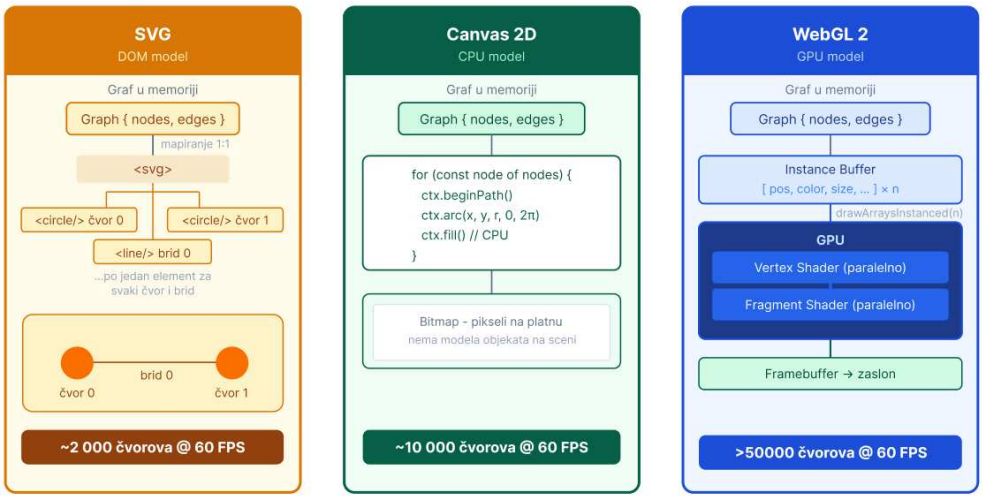
*HTML5 Canvas* element bio je, bez umanjenja, revolucija u crtanju u preglednicima, koja je uvela imperativni model crtanja koji potpuno zaobilazi *DOM*. *Canvas* je zaslon na koji se sadržaj crta piksel po piksel pozivanjem *JavaScript API* funkcija poput `fillRect()`, `arc()` ili `drawImage()`, bez ikakve trajne scene koja bi interagirala sa *DOM*-om. Ovaj pristup izradi prikaza znači da nema potreba da preglednik prati hijerarhiju vidljivih objekata što sugerira da nećemo imati *DOM* troškova ni za koji broj nacrtanih elemenata. Prednost te metode nad *SVG*-om pri većim grafovima se krije upravo u tome: tipična im-

plementacija može bez problema izraditi prikaz grafova s 5000 do 20000 čvorova uz omogućenu (iako i uz malo zahtjevniju implementaciju) interaktivnost, ovisno o složenosti stilova i raspoloživim računalnim resursima. Svaka promjena scene zahtijeva potpuno precrtavanje platna, što znači da pri pomicanju ili zumiranju čitav graf mora biti nanovo nacrtan svaki kadar, što je trošak koji raste linearno s brojem elemenata, iako postoje metode poput virtualnog iscrtavanja koje omogućavaju potpuno zaobići to ograničenje. Naime, kod grafova koji imaju više od 50000 elemenata čak i jednostavno iscrtavanje bez ikakve fizikalne simulacije počinje biti problematično, pogotovo uzimajući u obzir zahtjeve za interaktivnošću.

### 2.2.3. *WebGL* i *WebGL2*

Konačno, dolazimo do naprednijih tehnika koje postaju sve popularniji kako se preglednici iz alata za jednostavno pregledavanje stranice pretvaraju u dostojnu zamjenu za desktop aplikacije. *WebGL (Web Graphics Library)* i *WebGL2* su *API* koji *JavaScript* kodu u preglednicima omogućuju neposredan pristup GPU-u preko *OpenGL ES* grafičkog cjevovoda. Za razliku od prethodno spomenutih *SVG*-a i *Canvas API*-ja, kod kojeg se procesiranje svakog elementa odvija serijski na CPU-u, *WebGL* iskorištava masivno paralelnu arhitekturu GPU-a koji raspolaže stotinama do tisućama malih procesora koji mogu simultano obrađivati geometriju i piksele. Iako to mijenja cijelu paradigmu i pristup tome kako se nešto iscrtava, omogućuje potpuno iskorištavanje potencijala koji se nudi od strane računala. Kako to funkcionira? Geometrija koja se treba iscrtati prenosi se u GPU memoriju kao međuspremnik jednom, a zatim se crtanje može pozivati bez ponovnog prijenosa podataka, sve dok nema promjena geometrije čime se nivelira utjecaj najvećeg uskog grla u modernoj računalnoj grafici, a to je prijenos podataka između CPU i GPU. *WebGL* standardizirala je *Khronos Group* i u verziji 1.0 (baziranoj na *OpenGL ES 2.0*) dostupna je u svim modernim preglednicima od oko 2011. godine. Kako je ta verzija imala svoja ograničenja (iako je i dan danas jedna od najkorištenijih biblioteka za grafiku u preglednicima), napravljen je novi standard u obliku *WebGL 2* (baziran na *OpenGL ES 3.0*[3]) koji donosi niz naprednih mogućnosti ključnih za efikasnu vizualizaciju grafova. Među tim novitetima je instancirano prikazivanje (engl. *instanced rendering*) koje omogućuje crtanje tisuća geometrijski identičnih objekata (na primjer, svih čvorova jednakog oblika) jednim jedinim pozivom za crtanje. Svaka instanca može imati vlastite atribute

poput pozicije, boje i veličine, definirane u odvojenom međuspremniku instanci (engl. *instance buffer*) koji GPU čita paralelno za svaku instancu. Jedna od najvažnijih novih stvari koju je uveo *WebGL2* su teksture s pomičnom točkom (engl. *float textures*, *R32F*, *RG32F*, *RGBA32F*) neophodne za implementaciju *GPGPU* tehnika u *WebGL*-u. Bez mogućnosti pisanja podataka s pomičnom točkom u teksture i čitanja istih, implementacija fizikalne simulacije sila na GPU-u bila bi praktički nemoguća. Zajedno s mehanizmom koji se naziva *transform feedback* (u nekim implementacijama alternativa *ping-pong* tehnici, koju ćemo kasnije opisati) omogućuje bilježenje izlaza sjenčara vrhova (engl. *vertex shader*) nazad u spremnik bez prolaska kroz sjenčar fragmenata (engl. *fragment shader*) i slikovni međuspremnik (engl. *framebuffer*), što je skoro pa nezamjenjivo za određene klase GPU-baziranih simulacija, a tako i onu koju ćemo kasnije implementirati i objasniti. Programabilnost GPU cjevovoda realizira se pomoću sjenčara pisanih u općepoznatom *GLSL (OpenGL Shading Language)*, što olakšava adaptaciju ove tehnologije. Iako se najčešće koriste tri vrste sjenčara - sjenčare vrhova, geometrije i fragmenata, nas najviše zanimaju samo dvije koje će biti srce naše implementacije. Sjenčar vrhova, koji obrađuje svaki vrh geometrije i određuje njegovu poziciju u prostoru zaslona, koristit će se za transformaciju pozicije čvora iz prostora svijeta (engl. *world space*) u prostor odscijecanja (engl. *clip space*), te sjenčar fragmenata, koji određuje boju svakog fragmenta (kandidata za piksel), implementirat će zaglađivanje rubova kružnog oblika čvora i složenije efekte poput sjene.




**Slika 2.2.** Shematski prikaz tri pristupa vizualizaciji podataka u pregledniku

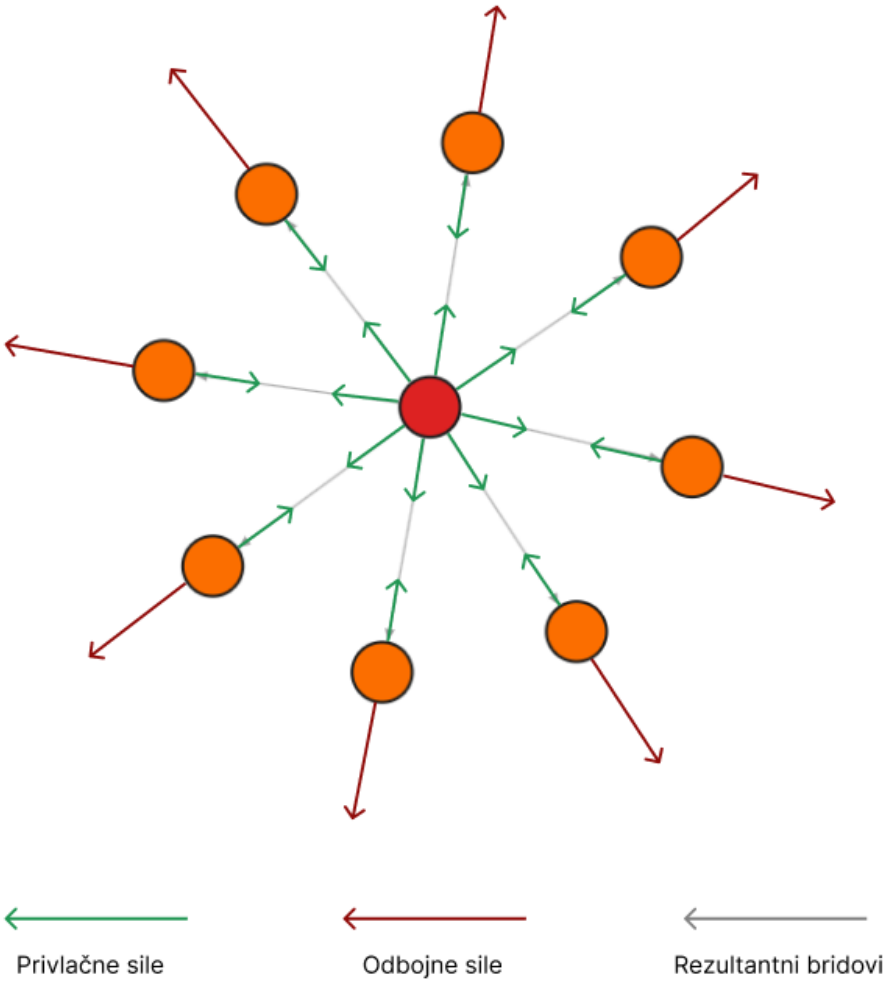
## 2.3. Algoritmi raspoređivanja čvorova

Teško je precijeniti važnost raspoređivanja čvorova (engl. *graph layout*) u kontekstu vizualizacije grafova. Imamo li graf $G = (V, E)$, treba odrediti poziciju $f : V \to \mathbb{R}^2$ svakog čvora u ravnini tako da rezultirajući prikaz bude ne samo vidljiv, ali i jasan i informativan. Problem je u općem slučaju NP-težak jer se sastoji od više problema poput minimizacije broja križanja bridova, koja je već po sebi NP-kompletan problem u slučaju planarnih grafova. Stoga su praktični algoritmi raspoređivanja ili heuristike ili aproksimacije koje optimiziraju jedan ili više kriterija estetske kvalitete poput minimalnosti križanja bridova, uniformnosti duljina bridova, simetričnosti raspoređivanja, refleksije topološke strukture i sličnosti s prethodnim raspoređivanjima pri inkrementalnim promjenama grafa. Iako postoje mehanizmi koji omogućuju jednostavnije rješenje nekih od ovih problema poput virtualnih čvorova u sredini svakog brida u fizikalnoj simulaciji čime se smanjuje broj križanja bridova, algoritmi raspoređivanja ostaju kako glavno rješenje, tako i najveći izvor problema u ovom kontekstu.

### 2.3.1. Algoritmi temeljeni na fizikalnoj simulaciji sila

Prva obitelj algoritama koju ćemo razmotriti, a ujedno i najraširenija, su algoritmi temeljeni na silama (engl. *force-directed layout algorithms*, *FDL*) koji se koriste za automatsko raspoređivanje grafova. Iz naziva je jasno da su ti algoritmi snažno inspirirani fizikalnim modelima gdje se čvorovi ponašaju kao nabijene čestice koje se međusobno odbijaju, dok bridovi djeluju poput opruga koje privlače susjedne čvorove jedan prema drugome. Iterativnom simulacijom ovog fizikalnog sustava (koja je, doduše, poprilično računalno zahtjevna), čvorovi konvergiraju prema ravnotežnom stanju koje inherentno teži minimizirati globalnu energiju sustava, što nas podsjeća na stohastičke optimizacijske modele poput simuliranog kaljenja. Rezultirajuće raspoređivanje često je vizualno prihvatljivo, pogotovo za manje grafove: čvorovi koji su topološki blizu (mali grafovski razmak) smještaju se prostorno blizu zahvaljujući privlačnim silama i obratno, dok grupe gušće međusobno povezanih čvorova (grupe) prirodno tvore prostorno odvojene nakupine. Eadesov model (1984.)[4] bio je prva formalizacija ovog pristupa, koja je modelirala bridove kao opruge čija je idealna duljina konstanta, a odbojnu silu između svih parova čvorova kao logaritamsku funkciju udaljenosti. Simulacija se u tom modelu iz-

vodi iterativno tako da se u svakom koraku za svaki čvor izračunava rezultantna sila svih privlačnih i odbojnih sila te se čvor na odgovarajući način pomiče u smjeru rezultantne sile. Proces se ponavlja sve dok sustav ne konvergira ili dok se ne dostigne maksimalni broj iteracija, što je još jedna sličnost s optimizacijskim algoritmima.




**Slika 2.3.** Primjer grafa dobivenog pomoću FDL algoritma

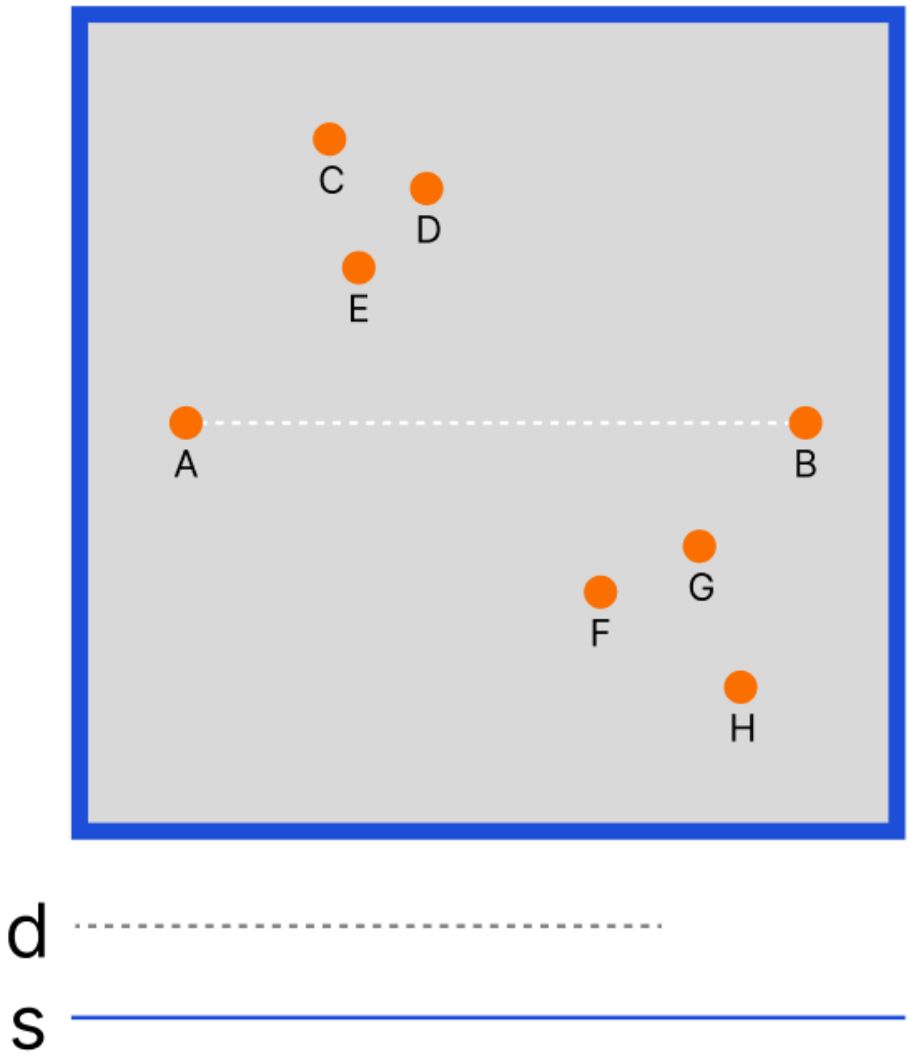
Fruchterman-Reingold model (1991.)[5] bio je pokušaj formalizacije kriterija estetike u kojemu odbojna sila između svakog para čvorova opada s kvadratom udaljenosti (po uzoru na Coulombov zakon), dok privlačna sila bridova raste s kvadratom udaljenosti (po uzoru na Hookov zakon). Uveli su i pojam "temperature" sustava, koji je, opet, sličan algoritmu simuliranog kaljenja, a koji kontrolira maksimalni pomak čvora u jednom koraku i postupno se smanjuje s napretkom simulacije. Ovaj model implementiran je danas u D3-force biblioteci, koja je trenutno jedna od najkorištenijih, i temelj je koji se koristi u velikom broju suvremenih implementacija. Glavna i najveća slabost naivnih FDL im-

plementacija je kvadratna složenost računanja odbojnih sila. U svakom koraku iteracije svaki čvor mora izračunati odbojnu silu od svakih drugog čvora, što rezultira $O(n^2)$ operacija po iteraciji. Za grafove s $n = 10000$ čvorova, to je $10^8$ izračuna sile po iteraciji, što je, očigledno, predugo za interaktivnu simulaciju (pogotovo na CPU-u). Privlačne sile bridova imaju složenost $O(m)$ po iteraciji gdje je m broj bridova, što je prihvatljivo budući da je m tipično znatno manji od $n^2$.

### 2.3.2. Barnes-Hut aproksimacija

Trenutno najefikasnije rješenje problema algoritamske složenosti naivnih implementacija je Barnes-Hut algoritam[6], koji su 1986. za simulaciju n-body problema razvili Josh Barnes i Piet Hut, koji vremensku složenost FDL simulacije smanjuje s $O(n^2)$ na $O(nlogn)$ po iteraciji. Osnovna ideja algoritma se temelji na aproksimaciji čvorova koji su dovoljno daleko od promatranog čvora jednim kompozitnim čvorom čija je masa (ili električni naboj) jednaka zbroju masa svih čvorova grupe, a čija je pozicija težište grupe. Kako je "dovoljno daleko" poprilično apstraktan pojam, definiramo ga parametrom $\theta$ na sljedeći način: ako je omjer duljine stranice promatrane regije $s$ (svejedno koje jer je promatrana regija uvijek kvadrat) i udaljenosti $d$ do promatranog čvora manji od $\theta$ (tj. $s/d < \theta$), cijela ćelija aproksimira se jednim čvorom.

Ovaj algoritam počinje podjelom prostora simulacije rekurzivnom kvadrantnom dekompozicijom, tzv. četverostablo (engl. *quadtree*, u 2D) ili osmerostablo (engl. *octree*, u 3D). Kako je naš trenutni zadatak ograničen na 2D prostor i vizualizacija 3D podataka je kompleksan zadatak za sebe, razmatrati ćemo samo kvadrantno stablo. Korijenski čvor stabla obuhvaća cijeli prostor simulacije. Ako ćelija sadrži više od jednog čvora, dijeli se na četiri jednaka kvadranta koji postaju djeca u stablu. Proces se nastavlja rekurzivno sve dok svaki list stabla ne sadrži najviše jedan čvor. Svaki unutarnji čvor stabla pamti ukupnu masu (ili naboj) svih čvorova u svom podstablu i koordinate njihovog težišta. Za izračun sile na promatrani čvor $v$, stablo se treba obići od korijena. Za svaki unutarnji čvor stabla koji predstavlja ćeliju sa stranicom $s$ na udaljenosti $d$ od čvora $v$, provjerava se uvjet $s/d < \theta$. Ako je uvjet zadovoljen, cijela ćelija se aproksimira jednim kompozitnim čvorom pa se za nju ne spušta dalje u stablo. Ako uvjet ipak nije zadovoljen (ćelija je preblizu ili prevelika), čvorovi ćelije moraju se obraditi individualno nakon čega se rekurzivno obilaze djeca ćelije. Za listove stabla (pojedinačne čvorove)




**Slika 2.4.** Parametri s i d tijekom prvog koraka Barnes-Hut algoritma

sila se uvijek izračunava egzaktno. Parametar $\theta$ kontrolira kompromis između brzine i točnosti, u slučaju kada je $\theta = 0$, dobivamo egzaktni $O(n^2)$ izračun, a $\theta = 1.0$ tipičan je odabir koji daje dobru ravnotežu između brzine i vizualne kvalitete raspoređivanja kako zahtjevi nisu egzaktni nego estetski. D3-force biblioteka koristi zadanu vrijednost $\theta = 0.9$, pa će i naša implementacija koristiti istu zadanu vrijednost. Dok je Barnes-Hut algoritam izvorno sekvencijski algoritam koji prirodno koristi rekurzivno obilaženje stabla, paradigmu koja se loše preslikava na GPU arhitekturu, nekoliko radova istražilo je mogućnosti paralelizacije[7]. Glavni izazov je što izgradnja i obilazak stabla uključuju uvjetna grananja i rekurziju koji su poprilično problematični na GPU-u gdje sve niti izvršavaju isti jednostavan kod (SIMD model paralelnog računanja). Pristup implementiran u ovom radu razrješava ovaj problem hibridnom strategijom na način da se četverostablo

gradi na CPU-u, a zatim se uzorkovačem tekstura (engl. *texture sampler*) prenosi u GPU memoriju gdje ga sjenčar fragmenata paralelno čita za izračun sile na svaki čvor, što je upravo razlog zašto je za implementaciju odabran *WebGL2*.

### 2.3.3. Ostali algoritmi raspoređivanja

Kao što smo već rekli, korištenje grafova u praksi je izrazito široko i bogato. Samim time, iako su FDL algoritmi opće namjene, za vizualizaciju nekih skupova podataka neće biti najbolji. Iz tog razloga postoji još bezbroj algoritama raspoređivanja. Kružno raspoređivanje (engl. *circular layout*) smješta sve čvorove na kružnicu jednakog polumjera. Izrazito je jednostavno za implementaciju i računski trivijalno, ali čitljivo samo za manji broj čvorova bez guste međusobne povezanosti. Njegova primarna primjena je vizualizacija bipartitnih grafova i mrežnih topologija. Taj je algoritam raspoređivanja implementiran i u sklopu ovog rada kao jedan od testnih algoritama, zajedno sa mrežnim raspoređivanjem (engl. *grid layout*) koje raspoređuje čvorove u pravilnu pravokutnu mrežu. Nema vizualne semantike osim ako su čvorovi sortirani prema nekom atributu, ali iznimno je brz i predvidiv te koristan kao inicijalno stanje prije pokretanja FDL simulacije jer sprječava problem inicijalnog preklapanja čvorova koji usporava konvergenciju. Osim ova dva, u svrhe testiranja implementiran je i hijerarhijski algoritam, koji je računalno kompliciraniji, ali koristan za prikazivanje roditeljskih odnosa između čvorova te za vizualizaciju stabala. Osim spomenutih, postoji još bezbroj algoritama (i, iskreno, nije ni toliko komplicirano napraviti neki svoj, ovisno o zadanom skupu podataka), ali spomenuti ovdje su najčešće korišteni i implementirani u sklopu ovog rada. Kako nisu pretjerano računalno zahtjevni, implementirani su na CPU strani osim GPU implementacije FDL algoritma koja će biti detaljnije opisana u idućim poglavljima.

## 2.4. Pregled postojećih alata

Na tržištu postoji više raznih biblioteka za vizualizaciju grafova u web okruženju. Važno je razumjeti postojeće relevantne alate s obzirom na korištene tehnologije, pristupe izradi prikaza, algoritme raspoređivanja, performanse i rasprostranjenost u praksi kako bi dobili drugu perspektivu na ovo područje. Cilj analize nije sveobuhvatan katalog svih biblioteka, već kontekstualizacija rješenja razvijenog u ovom radu u odnosu na konku-

rente.

### 2.4.1. *D3.js*

D3.js (Data-Driven Documents) Mikea Bostocka[8] univerzalna je biblioteka za vizualizaciju podataka u web okruženju, objavljena 2011. godine i danas jedna od najkorištenijih *JavaScript* biblioteka u cjelini koja se svaki dan razvija i širi. Sama D3 nije namijenjena isključivo vizualizaciji grafova, nego nudi i sloj apstrakcija za opće vezanje podataka uz *DOM* i *SVG* elemente te posbne module za različite izračune korisne u vizualizaciji. Vizualizacija grafova postiže se kombinacijom nekoliko D3 modula, od kojih je najvažniji d3-force. d3-force implementira ranije spomenuti Fruchterman-Reingold model sile s Barnes-Hut aproksimacijom (gdje je parametar $\theta = 0.9$) u čistom *JavaScript*-u koji se izvršava na CPU-u. Biblioteka je dosta dobro dokumentirana i modularna dovoljno da se pojedinačno mogu koristiti sile za odbijanje čvorova (engl. *many-body*), privlačenje bridova (engl. *link force*) te centriranje i sudaranje (engl. *collision*). Izrada prikaza je u praksi gotovo uvijek *SVG*-bazirana kada se koristi uz standardne D3 recepte, premda je *Canvas API* integracija moguća i bila je ostvarena u sklopu biblioteke Memgraph Orb kako bi izbjegli ograničenja nativne *SVG* vizualizacije uz pristup raspoređivanju čvorova generiranom pomoću d3-force. Iako taj sustav dobro funkcionira i dan danas, CPU-bazirana Barnes-Hut simulacija postaje usko grlo za grafove s više od 10000 čvorova gdje svaka iteracija traje nekoliko sekundi. D3 nema ugrađenu podršku za *WebGL* izradu prikaza u kontekstu vizualizacije grafova pa je jedan od ciljeva ovog rada bio ostvariti GPU implementaciju algoritma dovoljno blizu d3-force CPU raspoređivanju koje služi kao referenca.

### 2.4.2. *cytoscape.js*

Još jedna opcija je cytoscape.js[9] biblioteka za vizualizaciju grafova otvorenog kôda namijenjena ponajprije bioinformatici i analizi bioloških mreža, a nastala je kao web port originalnog Java Cytoscape alata. Biblioteka je iznimno bogata funkcionalnostima te podržava velik broj algoritama raspoređivanja (temeljeni na silama, hijerarhijski, kružni, spektralni, konzentrilni, *cose-bilkent* i dr.), ima bogat sustav stiliziranja čvorova i bridova CSS-sličnom sintaksom, selekciju i filtriranje elemenata, izvoz u slike i napredne animacije. Izrada prikaza u Cytoscape.js primarna je slabost u kontekstu skalabilnosti

jer biblioteka koristi *Canvas 2D* izradu prikaza bez *WebGL* podrške. Dok bogat skup funkcionalnosti čini cytoscape.js izuzetno moćnim za grafove srednje veličine (do 5000 čvorova), performanse pri interaktivnom radu s grafovima koji broje više od 10000 elemenata postaju neprihvatljive na prosječnom uređaju. Cytoscape.js ima eksperimentalni `cytoscapejs-canvas-webgl` dodatak koji pokušava ubrzati izradu prikaza *WebGL*-om, ali ovaj plugin nije dio jezgrene biblioteke, iako je dobra inspiracija za ostvarivanje sličnih funkcionalnosti.

### 2.4.3. *Cosmograph*

Najzanimljivija od ovih biblioteka je Cosmograph, suvremena biblioteka otvorenog kôda specijalizirana za vizualizaciju masivnih grafova koja prioritet daje performansama i iskorištavanju modernog hardvera. Ključna razlika u odnosu na D3.js i cytoscape.js je potpuna i nativna GPU akceleracija. Grafički procesor se ne koristi samo za izradu prikaza, već i za samu simulaciju sila (GPGPU[10]). Izvršavanjem algoritma temeljenog na silama na GPU-u, omogućen je interaktivni rad s grafovima koji broje stotine tisuća, pa čak i milijun elemenata, što je za CPU-bazirane simulacije nedostižno. Izrada prikaza se oslanja na *WebGL* i optimizirano je za ekstremnu skalabilnost, iako takav fokus na sirove performanse rezultira manjom fleksibilnošću u stiliziranju te je sustav vizualnih pravila znatno jednostavniji nego kod cytoscape.js, a podrška za kompleksne topološke algoritme ili bogate teksturne atlase je ograničena. Cosmograph je stoga primarno alat za istraživanje makroskopskih struktura velikih podataka, a manje za detaljnu stilizaciju pojedinačnih elemenata.

# 3. Arhitektura sustava

Kako se ovaj rad nadovezuje i u velikoj mjeri proširuje postojeću biblioteku za vizualizaciju grafova zvanu Memgraph Orb, potrebno je razumjeti kako ovaj sustav funkcionira i izgleda da se može adekvatno i skalabilno proširiti. Arhitektura ove biblioteke je slojevita te se drži određenih arhitekturnih principa kako bi se osigurala proširivnost, ispitivost i jasno odvajanje odgovornosti te se dodavanjem novih funkcionalnosti u sklopu ovog rada ta tendencija nastojala zadržati.

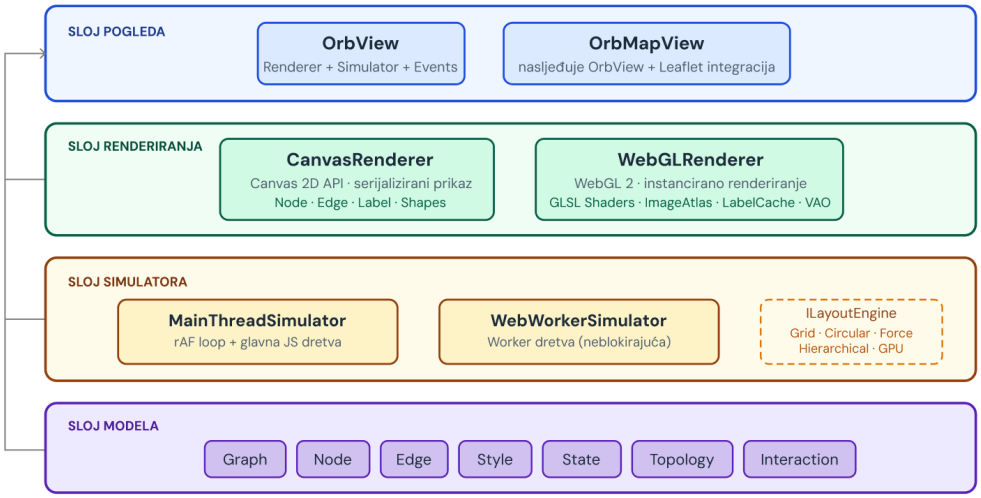
Arhitektura se pridržava tri glavna principa:

1. Najvažniji i najizraženiji je princip inverzije ovisnosti (engl. *Dependency Inversion Principle*), što znači da visoki slojevi (pogledi, aplikacijska logika) ne ovise o konkretnim implementacijama nižih slojeva (specifičan izrađivač prikaza, specifičan algoritam raspoređivanja), nego o apstrakcijama definiranim sučeljima. Konkretne implementacije se uvrštavaju tijekom izvođenja putem tvorničkih funkcija, što omogućuje transparentnu i dinamičnu zamjenu *Canvas* izrađivača prikaza *WebGL* izrađivačem ili CPU simulatora GPU simulatorom bez izmjene klijentskog koda.

2. Ne manje važan i većim djelom odgovoran za čitljiv kod je princip jedne odgovornosti (engl. *Single Responsibility Principle*) koji kaže da svaki modul ima jednu jasno definiranu odgovornost. Model podataka ne zna ništa o izradi prikaza, izrađivač prikaza ne zna ništa o algoritmima raspoređivanja, simulator ne zna ništa o grafičkom cjevovodu i slično. Ova stroga separacija odražava se u organizaciji direktorija izvornog koda i u samoj logici uz rijetke, neophodne za ostvarivanje nekih specifičnih funkcionalnosti, iznimke.

3. Oblikovni obrazac strategija (engl. *Strategy Pattern*) se koristi za sve promjenjive dijelove ponašanja od odabira koji izrađivač prikaza ili algoritam raspoređivanja koristiti

pa sve do odabira kojim se sučeljem prikazuje graf. Svaki odabir izoliran je u strategijske objekte koji implementiraju zajednička sučelja, a klijentski kod, koji koristi biblioteku, može zamijeniti bilo koju strategiju bez poznavanja unutarnjih detalja implementacije što nudi udobno i elegantno korištenje biblioteke.




**Slika 3.1.** Prikaz slojeva arhitekture biblioteke Memgraph Orb

## 3.1. Sloj modela podataka

Vizualizacija grafova je potproblem područja koje se naziva vizualizacija podataka. Stoga je iznimno važno imati dobar model podataka i entiteta kako bi se kod i logika prirodno nadovezivali na tu strukturu. Temeljne apstrakcije nad kojima rade svi ostali slojevi, kako ih nema puno, definirani su u zasebnoj mapi. Ovaj sloj nema ovisnosti o izrađivaču prikaza, *DOM*-u ni *WebGL*-u, nego se radi o čistim *TypeScript* klasama i sučeljima koja modeliraju grafovsku domenu.

### 3.1.1. Graf, čvor i brid

Temeljna apstrakcija našeg sustava je graf, u ovom slučaju definiran klasom `Graph<N, E>` koja je generička nad tipovima atributa čvorova (N) i bridova (E) kako bi se korisnicima omogućilo proširenje modela čvora i brida svojim podacima i atributima. Graf interno čuva čvorove i bridove u mapi indeksiranoj po identifikatoru, što osigurava $O(1)$ pristup po ID-u što je ključno za efikasno ažuriranje stila pojedinih entiteta (bilo u sklopu interakcije ili u sklopu programiranih izmjena) bez linearnog pretraživanja.

Kako se graf sastoji od čvorova i bridova, moramo imati definiranu generičku klasu

`Node<N>` koja enkapsulira:

1. **Podatkovni sloj** - originalni korisnički objekt tipa *N* (npr. rezultat iz Memgrapha), dostupan putem metode `getData()`.

2. **Stilski sloj** - objekt `NodeStyle` koji opisuje vizualne karakteristike čvora (boja, veličina, oblik, oznaka, slika i sl.), odvojen od podatkovnog sloja.

3. **Pozicijski sloj** - koordinate čvora u ptostoru svijeta, koje simulator ažurira pri svakoj iteraciji.

4. **Interakcijski sloj** - zastavice stanja *hover* i *selected*, koje izrađivač prikaza koristi pri stilizaciji.

Analogno, definirana je klasa `Edge<E>` s podatkovnim, stilskim i interakcijskim slojem, uz reference na čvorove krajeva (`startNode` i `endNode`). Iako poprilično jednostavan, ovaj model nudi izuzetno širok spektar mogućnosti za upravljanje entitetima grafa te pri tome ostaje skalabilan.

### 3.1.2. Stil i stilske strategije

Memgraph Orb nudi vrlo širok spektar opcija za stiliziranje grafa preko sučelja `NodeStyle` i `EdgeStyle` s opcionalnim poljima tako da svako polje koje nije postavljeno nasljeđuje zadanu vrijednost iz globalnog zadanog stila. Ovo omogućuje korisniku da postavi samo ona svojstva koja želi izmijeniti bez potrebe za navođenjem cijelog objekta stila te omogućuje korištenje patch metoda za dinamičku promjenu odabranih stilova.

Korisnik biblioteke implementira ovo sučelje predajući objekt s metodama `getNodeStyle` i `getEdgeStyle` pozivu `orb.data.setDefaultStyle(...)`. Metode se pozivaju za svaki čvor i brid neposredno prije izrade prikaza kadra, što osigurava da će izračunati stil odgovarati trenutnom stanju čvora (npr. drugačija boja za selektirani čvor, veličina proporcionalna stupnju čvora i sl.). Ova strategija (engl. *callable style*) fleksibilnija je od statičkog CSS pristupa jer stil može biti funkcija proizvoljno složene logike nad podacima čvora. Valja spomenuti da se u praksi stilovi za Orb često generiraju pomoću jezika Memgraph GSS (Graph Style Script) koji je specifično namijenjen za stilizaciju grafova (čvorova, bridova i pogleda) i razvijen upravo za korištenje zajedno sa Orb-om,

što će biti važno kad ćemo pričati o implementaciji kako je jedan od logičnih zahtjeva taj da naš izrađivač prikaza podržava sve postojeće mnogobrojne stilove.

**Kôd** 3..1: Primjer GSS koda za stilizaciju čvora u običnom i aktivnom stanjima

```
@NodeStyle {
  Define(COLOR, red)

  size: 6
  color: COLOR
  color-hover: Lighter(COLOR)
  color-selected: Darker(COLOR)
}
```

### 3.1.3. Topologija i stanje simulacije

Iako su navedene strukture same po sebi sasvim dovoljne za rješavanje širokog spektra problema, definirano je još nekoliko, ne toliko učestalih, sučelja, koja definiraju pomoćne strukture. Na primjer, strukture za topološku analizu grafa koji su potrebni algoritmima raspoređivanja poput liste susjedstva (engl. *adjacency list*) i liste stupnjeva čvorova. Osim toga, promjenjiva stanja vizualizacije enkapsuliramo sučeljem `GraphState` koji sadrži trenutnu razinu zumiranja i pomak kamere (engl. *pan*), skup selektiranih čvorova/bridova, stanje tekuće simulacije te stanje animacije tranzicije između raspoređivanja. Centralizacija promjenjivog stanja u jednom objektu omogućuje jednostavnu serijalizaciju i obnavljanje stanja (npr. za funkcionalnost povratka na prethodno raspoređivanje), a osim toga olakšava dodavanje novih cjevovoda za izradu prikaza kako je sva logika o stanju već implementirana i enkapsulirana na odgovarajućim mjestima te se ne trebamo o tome brinuti.

### 3.1.4. Interakcijski model

Da bi ostvarili interakciju korisnika s grafom, što je ključan zahtjev takvih biblioteka, trebamo imati opisane tipove za opisivanje interaktivnih događaja, u našem slučaju `NodeClickEvent`, `EdgeClickEvent`, `MouseMoveEvent` i sl. Svaki tip događaja nosi referencu na entitet (čvor ili brid) koji je bio predmet interakcije, koordinate miša u prostoru svijeta i prostoru zas-

lona, te metapodatke događaja (koje tipke su pritisnute i sl.).

## 3.2. Sloj pogleda

Budući da izrađujemo biblioteku koju će netko koristiti, trebamo definirati javno sučelje (engl. *public API*), u našem slučaju preko klasa `OrbView` i `OrbMapView`.

### 3.2.1. `OrbView`

OrbView je temeljna klasa koja implementira interaktivnu vizualizaciju grafa unutar kontejnerskog HTML elementa. Konstruktor prima referencu na kontejnerski *DOM* element i opcionalni konfiguracijski objekt kojim se opisuju inicijalne postavke grafa, što uključuje inicijalno stanje simulatora, korišten algoritam raspoređivanja čvorova te, u budućnosti, način i postavke renderiranja (CPU vs GPU). Interno, OrbView koordinira tri podsustava:

1. **Izrađivač prikaza** koji se instancira pozivom tvorničke funkcije `createRenderer(config.render)` koja vraća ili `CanvasRenderer` ili `WebGLRenderer`, koje ćemo opisati kasnije, ovisno o konfiguraciji.

   OrbView drži referencu tipa `IRenderer` (sučelje) i ne zna koji je konkretan izrađivač prikaza aktivan, što omogućuje njegovu dinamičku promjenu u stvarnom vremenu.

2. **Simulator** koji se instancira pozivom tvorničke metode `createSimulator(config.layout)`.

   Analogno izrađivaču prikaza, `OrbView` drži referencu tipa `ISimulator` i ne zna je li aktivan CPU ili GPU simulator, niti koji algoritam raspoređivanja se trenutno koristi.

3. **Event manager** koji registrira listenere na *Canvas* ili *WebGL* element za mouse i touch događaje te ih prevodi u apstraktne grafovske događaje (`NODE_CLICK`, `EDGE_HOVER` i sl.) koje korisnik biblioteke može slušati.

Životni ciklus jednog kadra u OrbView:

1. **Poziv funkcije** `requestAnimationFrame()` kojom preglednik neposredno prije iscrtavanja sljedećeg okvira (engl. *frame*) poziva funkciju proslijeđenu kao argument.

2. **Ažuriranje pozicija čvorova** pozivom `Simulator.step()`, pod uvjetom da je simulacija aktivna.

3. **Iscrtavanje kadra** pomoću metode `Renderer.render(graph, state)`.

4. **Emitiranje događaja** `RENDER_END`.

Metoda `setSettings()` prima parcijalni konfiguracijski objekt i rekonfigurira aktivni simulator što znači da korisnik može, primjerice, promijeniti jačinu odbojnih sila (`manyBody.strength`) ili parametar theta (`manyBody.theta`) tijekom izvođenja bez ponovnog pokretanja simulacije od nule.

### 3.2.2. `OrbMapView`

`OrbMapView` je proširenje klase `OrbView` integracijom s *Leaflet* bibliotekom koja služi za izradu interaktivnih kartografskih aplikacija. Umjesto slobodnog postavljanja čvorova u prostor svijeta, `OrbMapView` zahtijeva da svaki čvor ima geografske koordinate (`latitude`, `longitude`) koje se mapiraju u koordinate trenutnog pogleda karte. Arhitekturalno, `OrbMapView` interno instancira `Leaflet Map` objekt i postavlja *WebGL* izrađivač prikaza kao sloj iznad *Leaflet* slojeva. Svaki put kada se *Leaflet* karta pomiče ili zumira, `OrbMapView` ponovno izračunava projekcijsku matricu izrađivača prikaza kako bi pozicije čvorova u prostoru svijeta odgovarale novim koordinatama geografskih pozicija. Ova sinkronizacija realizirana je slušanjem *Leaflet* `moveend` i `zoomend` događaja. Ključno je da `OrbMapView` ne koristi simulator za raspoređivanje čvorova, nego su pozicije fiksno određene geografskim koordinatama. Simulator ostaje dostupan za grafove bez geografskog konteksta, ali je za kartografski prikaz zamijenjen direktnim geo-projekcijskim izračunom.

**Slika 3.2.** Primjer vizualizacije geografskog skupa podataka koji sadrži podatke o glavnim gradovima država

## 3.3. Sloj izrade prikaza

Ovaj sloj, što i slijedi iz naziva, implementira sve mehanizme vizualnog prikaza grafa. Kako je ovaj sloj dosta velik, organiziran je u tri odgovarajuća direktorija. `canvas/` za Canvas 2D izrađivač prikaza, `webgl/` za *WebGL2* izrađivač prikaza, te `factory.ts` i `shared.ts` za tvornicu i zajednička sučelja. Ovdje se, kao i na mnogim drugim mjestima u ovoj biblioteci, koristi tvornica kako bi osigurali mogućnost dinamičke promjene izrađivača prikaza tijekom rada programa.

### 3.3.1. *Canvas* izrađivač prikaza

Prvi od dva izrađivača prikaza koja je biblioteka Memgraph Orb nudila i prije naših modifikacija je *canvas* izrađivač prikaza koji, u svrhe funkcioniranja tvornice i strategije, implementira `IRenderer` te koristi *HTML5 Canvas API*. Interna organizacija *canvas* izrađivača prikaza razlaže izradu prikaza po tipu elementa te odvojeno implementira logiku za iscrtavanje čvorova, tekstualnih oznaka te bridova, koji imaju najkompleksniju logiku, uzimajući u obzir da je riječ o tri vrste bridova - obični (ravni bridovi), višestruki (zakrivljeni) bridovi te povratni (također zakrivljeni, ali početak i kraj imaju u istom čvoru) veze. *Canvas* izrađivač prikaza ne koristi instanciranu izradu prikaza što znači da se svaki čvor, brid i oznaka crtaju zasebnim *Canvas 2D API* pozivima u petlji. Ovo je prihvatljivo za manji broj elemenata i, dok dobro funkcionira i obavlja glavne zadatke ove biblioteke, nama će služiti kao referentna implementacija s kojom ćemo uspoređivati našu *WebGL* implementaciju.

### 3.3.2. *WebGL* izrađivač prikaza

WebGL2[11] izrađivač prikaza, koji predstavlja glavni fokus ovog rada, također implementira sučelje `IRenderer` te koristi *WebGL2 API* s instanciranom izradom prikaza i *ping-pong* simulacijom. *WebGL* izrađivač prikaza interno drži zasebne strukture podataka (`VAO` i međuspremnike instanci) za svaki od tri tipa elemenata (čvorovi, bridovi, oznake). Životni ciklus jedne izrade prikaza `render(graph, state)`:

1. `gl.clear()` - brisanje slikovnog međuspremnika

2. Ažuriranje projekcijske MVP matrice (ako se zoom/pan promijenio)

3. `LabelCache.getOrCreate()` za sve vidljive čvorove te rasterizacija oznaka u atlas

4. `ImageAtlas.getOrCreate()` za sve čvorove sa slikom te dohvat/raspoređivanje slika

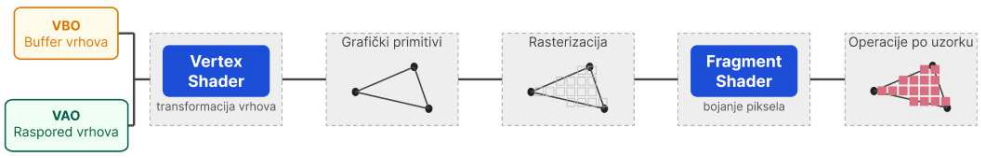
5. `LabelCache.uploadIfDirty()` za prijenos atlasa u GPU teksturu (ako dirty)

6. `ImageAtlas.uploadIfDirty()` za prijenos atlas teksture na GPU (ako dirty)

7. Ažuriranje međuspremnika instanci za promijenjene čvorove

8. `useProgram(nodeProgram)` kako bi se omogućilo korištenje sjenčara za instancirano iscrtavanje preko $drawArraysInstanced(nodes)$

9. Iscrtavanje bridova i oznaka korištenjem istog mehanizma




**Slika 3.3.** Pojednostavljen WebGL cjevovod

Redoslijed izrade prikaza je važan: bridovi se prikazuju ispod čvorova, a oznake iznad svih. U *WebGL*-u, koji nema automatskog upravljanja dubinom za 2D scene, ovaj redoslijed osigurava ispravan vizualni preklapajući redoslijed bez korištenja ispitivača dubine (engl. *depth tester*) i bez gubitka vizualne informacije o čvorovima te oznakama jer su ti entiteti u pravilu važniji od bridova u smislu vizualizacije.

## 3.4. Sloj simulatora

Sve algoritme raspoređivanja čvorova, od jednostavnih statičkih raspoređivanja do GPU-akceleriranog raspoređivanja temeljenog na silama, se nalaze u zasebnom logičkom sloju simulacije, prateći postojeću strukturu projekta.

### 3.4.1. Arhitektura simulatora

Simulator je organiziran u dvostruku hijerarhiju apstrakcija gdje gornja razina razlikuje gdje se simulacija izvodi (na glavnoj niti ili u *WebWorker*-u), a donja razina razlikuje koji algoritam raspoređivanja se koristi. Naime, važno je napomenuti da *WebWorker* okruženje ne podržava korištenje *WebGL*-a. Dok je korištenje *WebWorker*-a neophodno za očuvanje interakcije kad god se za izračun raspoređivanja koristi CPU, moramo imati mogućnost dinamičke promjene okruženja kako bi se omogućilo korištenje GPU u provođenju tih izračuna. To je ujedno i glavni razlog za ovakvu podjelu arhitekture.

### 3.4.2. `MainThreadSimulator` i `WebWorkerSimulator`

Kao što smo već rekli, jedna od opcija je izvođenje simulacije direktno u *JavaScript* glavnoj niti pomoću `MainThreadSimulator` klase. Za upravljanje pojedinačnim iteracijama koristi se metoda `requestAnimationFrame`. Ovo je jednostavniji pristup koji funkcionira u svim okruženjima, ali može uzrokovati kratka zamrzavanja UI-ja za računalno zahtjevne simulacije, što se u praksi često i događalo. S druge pak strane, `WebWorkerSimulator` premješta simulacijski pogon u zasebnu `Web Worker` dretvu, oslobađajući glavnu nit za izradu prikaza i reagiranje na korisničku interakciju. Komunikacija između glavne niti i workera odvija se asinkronom razmjenom poruka definiranih u zasebnoj mapi:

- `worker-input.ts` - definira poruke koje glavna nit šalje workeru (`START`, `STOP`, `UPDATE_SETTINGS`).
- `worker-output.ts` - definira poruke koje worker šalje glavnoj niti (`STEP_DONE`, `END`, `POSITIONS`).
- `worker-payload.ts` - definira tipove korisnih tereta (engl. *payload*) poruka.

### 3.4.3. Hijerarhija pogona za raspoređivanje čvorova

Prethodno smo naveli neke od algoritama koji se najčešće koriste za raspoređivanje čvorova. Ovdje su implementirani neki od tih algoritama te su podijeljeni u dvije skupine. Statički pogoni su najjednostavniji i izračunavaju pozicije čvorova jednom i odmah završavaju bez iterativne simulacije. `GridLayoutEngine` raspoređuje čvorove u pravilnu rešetku, `CircularLayoutEngine` ih postavlja na kružnicu, a `HierarchicalLayoutEngine` implementira Sugiyama metodu za raspoređivanje čvorova u stabala. S druge strane, dinamički pogoni iterativno ažuriraju pozicije čvorova u svakom koraku simulacije sve dok sustav ne konvergira. `BaseLayoutEngine` služi kao osnova za sve druge pogone te implementira zajedničku logiku alpha hlađenja, upravljanja zastavicama zaustavljanja i pozivanja callback-ova na korak i završetak simulacije. Konkretni dinamički pogoni nasljeđuju `BaseLayoutEngine` i implementiraju samo metodu `_step()` koja obavlja jednu iteraciju specifičnog algoritma, što čini ovu strukturu fleksibilnom za bilo kakva proširenja bez potrebe promjene postojećeg koda.

### 3.4.4. `ForceLayoutEngine` i `GpuForceLayoutEngine`

`ForceLayoutEngine` je prvi pogon implementiran u sklopu Memgraph Orb biblioteke te služi kao jednostavan omotač oko biblioteke `d3-force`, koju smo spominjali ranije, kako bi ostatku biblioteke omogućio korištenje pozicija čvorova izračunatih na taj način. Nedostatak ovog pogona jest da se može izvršavati samo na CPU.

Upravo je zato napravljen `GpuForceLayoutEngine` koji implementira isti algoritam na GPU-u korištenjem *ping-pong* sjenčara fragmenata. Svaka iteracija:

1. **Izgrađuje strukturu četverostabla** pozivom metode `QuadtreeBuilder.build(nodes)` na CPU-u.

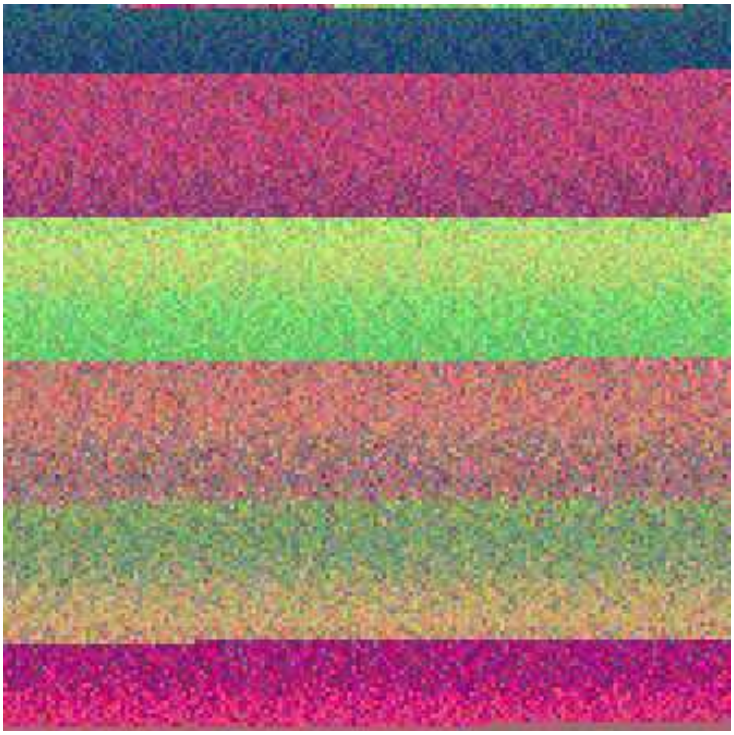
2. **Serijalizira i prenosi podatke** na način da CPU serijalizira četverostablo u `Float32Array` i učitava ga u GPU teksturu pomoću funkcije `gl.texSubImage2D()`.

3. **Izvršava simulacijsku iteraciju** pri čemu GPU paralelno izvršava jednu ping-pong iteraciju simulacijskog sjenčara fragmenata za sve čvorove.

4. **Dohvaća rezultate** tako da CPU čita novu pozicijsku teksturu natrag u radnu me-

moriju pomoću `gl.readPixels()`.

5. **Ažurira graf** tako da se pozicije čvorova ažuriraju u objektu `Graph` otkud postaju dostupne za sve potrošače, uključujući i izrađivače prikaza.



**Slika 3.4.** Stanje grafa veličine oko 100000 čvorova zakodirano kao slika (R = x, G = y, B = vx, A = vy)

### 3.4.5. Sjenčari simulatora

`force.vert` i `force.frag` su GLSL sjenčari koji implementiraju simulacijski korak na GPU-u. Za razliku od sjenčara koji su odgovorni za vizualni prikaz, ovi sjenčari nemaju vizualnog izlaza u tradicionalnom smislu - njihov "vizualni" izlaz (boja fragmenta) zapravo kodira novo stanje simulacije (pozicija i brzina) koje se zapisuje u *ping-pong* slikovne međuspremnike. Fizička separacija koda sjenčara u zasebne GLSL datoteke (`.vert`, `.frag`) umjesto ugrađivanja u obliku *JavaScript* znakovnih nizova predstavlja svjestan arhitekturalni odabir koji poboljšava čitljivost (IDE može primijeniti GLSL sintaksno bojanje), omogućuje linting koda sjenčara i olakšava izoliranu izmjenu sjenčara. Napravljena je i datoteka koja definira *TypeScript* deklaracije koje omogućuju importiranje GLSL datoteka kao *TypeScript* modula čiji sadržaj zatim modul za spajanje koda (`Webpack` s odgovarajućim dodatkom) ubacuje kao konstante tijekom izgradnje, čime

finalni paket sadrži kod sjenčara kao znakovne nizove koji se kompajliraju tijekom izvođenja.

# 4. Implementacija

Ovo poglavlje opisuje implementacijske detalje ključnih komponenti sustava. Dok je prethodno poglavlje opisivalo što svaka komponenta radi i kako se uklapa u cjelokupnu arhitekturu, ovo će poglavlje pokušati opisati kako je svaka komponenta konkretno implementirana s naglaskom na netrivijalne aspekte koji zahtijevaju posebno razmatranje. Krenut ćemo s opisom *WebGL* tehnika koje sad stoje iza cijelog procesa izrade prikaza nakon čega ćemo objasniti neke specifičnosti tog sustava poput LOD (razina detalja) te ćemo priču zaokružiti opisom *WebGL* pristupa računanju Barnes-Hut aproksimacije.

## 4.1. Izrađivač prikaza čvorova

### 4.1.1. Organizacija geometrije i spremnika

Konceptualno se mentalni model programiranja u *WebGL*-u znatno razlikuje od klasičnijih pristupa crtanju stvari u pregledniku poput *SVG* ili *Canvas API*-ja. Dok je *SVG* deklarativan model, *Canvas* i *WebGL* su konceptualno bliži jer su imperativni ali se i dalje jako razlikuju - dok se *Canvas API* više može doživjeti kao platno na kojem direktno crtamo našim programima, *WebGL* je konceptualno bliži tvorničkoj traci u smislu da gradimo sustav koji podatke pohranjene u spremnike obradom kroz više manjih programčića (sjenčara) pretvara u sliku. U našem slučaju, svaki čvor grafa se prikazuje kao instanca jedne bazne geometrije. Odabir vrste bazne geometrije među postojećim primitivima nije trivijalan - premda bi točka (`gl.POINTS`) bila najjednostavnija (i, vjerojatno, najiuntuitivnija) opcija, nakon isprobavanja se uspostavilo da ima ograničenu podršku za veličine veće od 1 px na nekim platformama i ne pruža dovoljno teksturnih koordinata za kompleksne oblike što je jedan od zahtjeva postojećeg sustava za stilizaciju. Stoga je svaki čvor prikazan kao *unit quad* iliti pravokutnik sastavljen od dva trokuta koji obuhvaća prostor $[-1, 1] \times [-1, 1]$ u lokalnom koordinatnom sustavu te se sastoji od šest

vrhova (dva trokuta) koja su pohranjena u spremnike geometrije koji se inicijalizira jednom i nikad više ne mijenja. Međuspremnik instanci organiziran je kao spremnik bez indeksiranja u kojem su svi atributi jedne instance pohranjeni sekvencijalno u memoriji, a ne kao zasebni spremnik po atributu. Takav format daje bolje *cache* performanse jer GPU čita sve atribute iste instance iz jedne *cache* linije. Razmak između početaka susjednih instanci (engl. *stride*) izračunava se kao suma bajtova svih atributa jedne instance. U ovoj implementaciji *stride* iznosi $25 \times 4 = 100$ bajtova po čvoru. Za n = 100000 čvorova međuspremnik instanci zauzima 10 MB GPU memorije, što je zanemarivo za moderne sustave. Parametri povezivanja atributa pojedinih instanci se postavljaju jednom pri inicijalizaciji unutar VAO objekta.


**Slika 4.1.** Izgled spremnika koji se šalje sjenčaru za jednu instancu

### 4.1.2. Sjenčar vrhova čvorova

Do sada smo saznali da je vizualizacija grafova netrivijalan zadatak koji zahtijeva istovremenu obradu i prikaz velike količine konceptualno sličnih skupina elemenata, čvorova i bridova. U mnogim grafičkim bibliotekama se koriste mali specijalizirani programi zvani sjenčari, koje smo već više puta i spominjali, a koji se odlično uklapaju u rješenje ovog zadatka. Stoga krećemo sa opisivanjem sjenčara vrhova čvorova (`node.vert`) koji obavlja dva primarna zadatka - transformaciju pozicije vrhova iz lokalnog prostora u prostor odscijecanja i prosljeđivanje potrebnih atributa dalje sjenčaru fragmenata koji obavlja većinu teškog matematičkog posla. Nekoliko detalja ovdje zaslužuje poseban komentar. Transformacijska matrica `u_transform` je $3 \times 3$ matrica afine transformacije koja u sebi kombinira zoom i pan. Korištenje $3 \times 3$ umjesto $4 \times 4$ matrice, što je pomalo neuobičajen odabir u računalnoj grafici, je svjesna odluka koja štedi uniformni prostor jer je scena isključivo 2D. Matrica se ažurira u *JavaScript* kodu svaki put kad korisnik zumira ili pomiče scenu. Skaliranje quad-a s marginom ($half = a\_size + margin$) je implementacijski trik koji, iako čini mehanizam vizualizacije samog čvora manje intuitivnim, omogućuje prikaz sjene i obruba izvan geometrijskog radijusa čvora. Stoga treba paziti da je quad dovoljno velik da obuhvati ne samo tijelo čvora nego i zonu sjene, što je

i napravljeno u jednoj iteraciji razvoja ovog sjenčara.

### 4.1.3. Sjenčar fragmenata čvorova i zaglađivanje rubova

Ključni element ovog cjevovoda za prikaz čvorova je sjenčar fragmenata čvorova (`node.frag`) koji je odgovoran za vizualni oblik čvora, zaglađivanje rubova i prikaz sjene i obruba. Njegov ključni element jest korištenje *Signed Distance Field (SDF)* funkcija za definiranje oblika. SDF funkcija pruža veoma elegantan način za opisivanje oblika čvora na način da za danu točku $p$ vraća negativnu vrijednost ako je točka unutar oblika, nulu na rubu oblika te pozitivnu vrijednost ako je točka izvan oblika. Pametnim oblikovanjem SDF funkcija možemo iskoristiti oblikovni obrazac strategija kako bi podržali iscrtavanje različitih oblika, od najjednostavnije kružnice do kompliciranih oblika poput zvijezde, što je jedan od zahtjeva sustava za stiliziranje. Iako se korištenje uvjetnih naredbi u sjenčarima smatra lošom praksom zbog jednostavnosti GPU jezgri, to je prihvatljivo u ovom slučaju jer je provjera izrazito jednostavna, ne stvara previše grananja tijekom izvođenja programa, a istovremeno omogućava dinamički prikaz više različitih oblika u stvarnom vremenu.

**Kôd** 4..1: Primjer jednostavnih SDF funkcija za oblike kružnice, kvadrata i romba

```
float sdCircle(vec2 p, float r) {
  return length(p) - r;
}

float sdSquare(vec2 p, float r) {
  vec2 d = abs(p) - vec2(r);
  return max(d.x, d.y);
}

float sdDiamond(vec2 p, float r) {
  return (abs(p.x) + abs(p.y)) - r;
}
```

Reprezentacija koju generira ovaj sjenčar iznimno je pogodna za zaglađivanje rubova jer se umjesto oštrog binarnog odabira "unutra/izvana" korištenjem funkcije `smoothstep(-aaWidth,`

`aaWidth, d)` generira glatki prijelaz u tankom pojasu oko ruba oblika. Širina zone zaglađivanja rubova `u_aaWidth` namještena je u *JavaScript* kodu proporcionalno trenutnom zoomu - pri malom zoomu (čvorovi su mali) zona je šira u normaliziranim koordinatama, čime se sprječava pikselizacija generirane vizualizacije. Isti mehanizam koristi se i za obrub na način da pomicanje praga $d$ prema unutra za `v_borderWidth` daje granicu između zone tijela i zone obruba, a `smoothstep` između te dvije granice generira glatki obrub bez dodavanja eksplicitne geometrije obruba, što dodatno pojednostavljuje proces vizualizacije i povećava performanse.

### 4.1.4. Prikaz slika čvorova

Biblioteka omogućuje postavljanje `imageUrl` parametara čvorovima što omogućava njihovo prikazivanje s teksturom slike umjesto ili iznad standardne boje. Implementacija koristi `ImageAtlas` modul koji ćemo opisati kasnije, ali logika za vizualizaciju i korištenje tekstura nalazi se upravo u sjenčaru fragmenata. U međuspremniku instanci, za čvorove sa slikom dodaju se UV atributi koji označavaju regiju unutar atlasa te se taj podatak koristi u sjenčaru fragmenata tako da ako je $a\_hasImage > 0.5$, tekstura se uzorkuje unutar SDF maske. Omjer stranica (engl. *aspect ratio*) slike ispravlja se u sjenčaru vrhova kako bi se slika prikazala proporcionalno bez deformacije bez obzira na omjer veličina čvora. Za čvorove čija slika još nije učitana (asinkrono dohvaćanje u tijeku), `a_hasImage` ostaje postavljeno na 0 i čvor se prikazuje standardnom bojom. Čim `ImageAtlas` završi pakiranje slike u atlas i postavi $\_dirty = true$, sljedeći kadar automatski učitava novi atlas i čvor počinje prikazivati sliku bez ikakve eksplicitne intervencije korisnika. Asinkrono učitavanje slika je jedna od standardnih praksi u preglednicima pa ovakav dinamički pristup omogućuje dobro korisničko iskustvo.

## 4.2. Izrađivač prikaza bridova

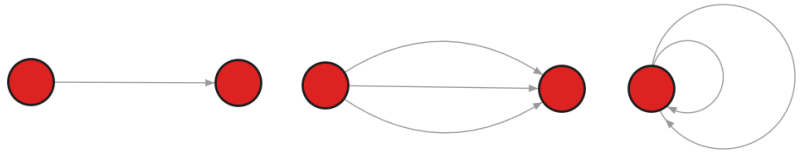
### 4.2.1. Tipovi bridova i njihova geometrija

Grafovi u praksi sadrže tri tipa bridova koji zahtijevaju zasebnu geometrijsku reprezentaciju:

1. Ravni brid (engl. *straight edge*), jednostavna direktna linija između dva čvora.

2. Zakrivljeni brid (engl. *curved edge*) predstavljen Bézierovom krivuljom, koristi se za vizualno razlikovanje višestrukih bridova između istih čvorova.

3. Povratni brid (engl. *loopback/self-loop*) koji počinje i završava u istom čvoru, prikazuje se kao elipsa.



**Slika 4.2.** Primjer jednostavnog grafa sa svim vrstama bridova

Odabir tipa brida provodi se u *JavaScript* kodu pri izgradnji međuspremnika instanci te se za svaki brid provjerava se je li $edge.start \texttt{ === } edge.end$ (povratni brid) i koliko već postoji bridova između istog para čvorova (za odluku o zakrivljenosti). Nakon toga, slično mehanizmu koji se koristi za izradu prikaza različitih oblika čvorova, koristimo strategiju kako bi prikazali sve ove vrste čvorova jednostavnim grananjem varijable `aEdgeType` koja je proslijeđena sjenčaru preko spremnika uz ostale podatke o atributima pojedinih bridova.

### 4.2.2. Ravni bridovi

Najjednostavniji slučaj, ravni brid, na prvu naivno implementiran bez podrške ostalih vrsta bridova, implementira se kao uski pravokutnik orijentiran duž pravca koji spaja čvorove izvora i odredišta. Bazna geometrija je horizontalni unit quad koji sjenčar vrhova rotira i skalira u prostoru svijeta. Bazna geometrija brida je *quad* čije koordinate `aQuadPosition` leže u rasponu $[-1, 1]$ po objema osima, a centriran je na polovištu brida. Sjenčar vrhova gradi lokalnu ortonormiranu bazu koja se sastoji od jediničnog vektora smjera brida i njegovog 2D okomitog vektora, te tim baznim vektorima transformira quad koordinate u prostor svijeta. Zaglađivanje rubova u sjenčaru fragmenata oslanja se na funkciju `fwidth(sdf)` koji automatski prilagođava širinu prijelaznog pojasa trenutnoj razini zumiranja. Zaglađivanje ruba brida provodi se u sjenčaru fragmenata analogno čvorovima funkcijom `smoothstep` koja generira glatki prijelaz na rubovima.

### 4.2.3. Zakrivljeni bridovi

Za kompliciranije slučajeve, odnosno za grafove u kojima između istog para čvorova postoji više bridova, ravni bridovi bi se preklapali i postali nerazlučivi. Rješenje je prikazivanje svakoga od višestrukih bridova kao kvadratnu Bézierovu krivulju s različitim stupnjevima zakrivljenosti. Kvadratna Bézierova krivulja definirana je s tri kontrolne točke $P_0$, $P_1, P_2$, pri čemu se točke određuju prema $B(t) = (1-t)^2P_0 + 2(1-t)tP_1 + t^2P_2, \quad t \in [0,1]$ Za vizualizaciju grafova, $P_0$ i $P_2$ su pozicije čvorova izvora i odredišta, a $P_1$ je kontrolna točka koja određuje zakrivljenost. Kontrolna točka izračunava se u *JavaScript* kodu kao pomak okomito od pravca brida proporcionalan indeksu brida u skupu višestrukih bridova. Brid se prikazuje kao jedan *quad* opisan omeđujućim okvirom (engl. *bounding box*) krivulje, a sjenčar fragmenata analitički računa egzaktnu udaljenost piksela od krivulje rješavanjem kubične jednadžbe bez aproksimacije segmentima, čime se postiže glatkoća neovisna o razini zumiranja.

### 4.2.4. Povratni bridovi

Povratni brid (self-loop) poseban je slučaj koji zahtijeva zasebnu geometriju. Implementiran je kao kružni luk tako da sjenčar vrhova gradi omeđujući pravokutnik oko kružnice definirane centrom (`aControl`) i radijusom (`aLoopbackRadius`), analogno pristupu za Bézierove krivulje. Sjenčar fragmenata potom analitički računa udaljenost piksela od kružnice formulom $d = \left| \|p - c\| - r \right|$, gdje je $r$ polumjer kružnice, $c$ centar kružnice, a $p$ pozicija trenutnog piksela u svjetskim koordinatama, što daje egzaktnu geometriju bez aproksimacije segmentima i neovisno o razini zumiranja.

### 4.2.5. Strelice na usmjerenim bridovima

Za usmjerene grafove (a biblioteka Memgraph Orb samo takve i podržava), bridovi trebaju vizualnu indikaciju smjera. Strelice su implementirane kao proceduralni SDF unutar istog sjenčara fragmenata kao i brid te su opisane atributima `aArrowTip`, `aArrowDir` i `aArrowSize` koje se prenose kao dio iste instance u spremniku. Sjenčar fragmenata računa `sdArrow` neovisno od SDF-a brida te ih kombinira operacijom unije (`min`), što omogućuje neovisno skaliranje strelice od širine brida bez deformacije pri bilo kojoj razini zumiranja te pruža jasno razdvajanje odgovornosti kod iscrtavanja bez gubitka koncep-

tualne i vizualne povezanosti.

## 4.3. Sustav teksturnih atlasa

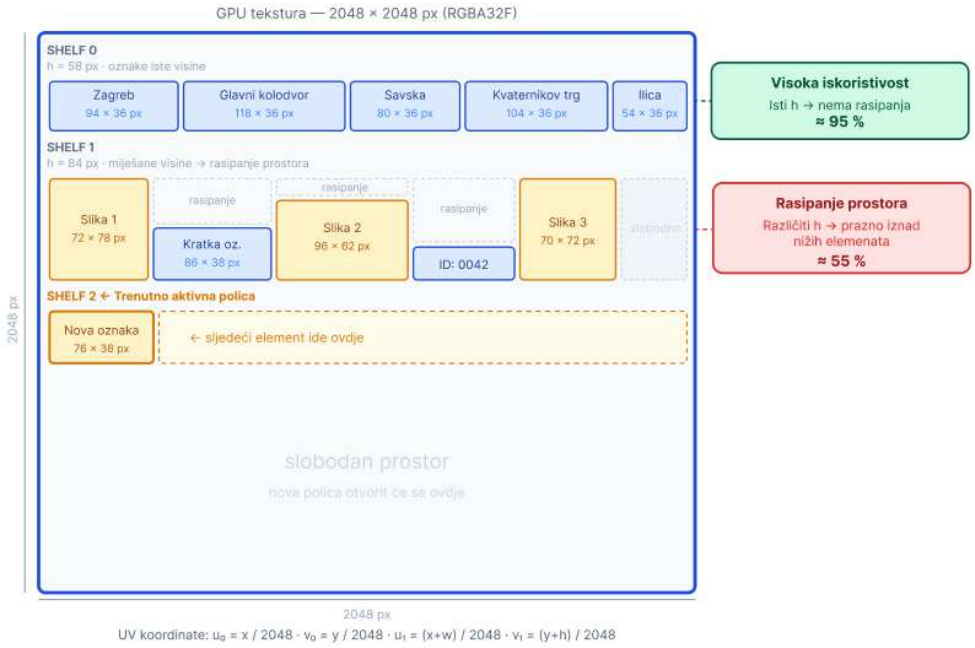
### 4.3.1. Motivacija za atlasni pristup

Prikazivanje teksta i slika u *WebGL*-u temeljni je problem koji, za razliku od *SVG* i *Canvas API*-ja, nema trivijalno rješenje. Za razliku od *Canvas 2D API*-ja koji nudi `fillText()` i `drawImage()` kao nativne operacije, *WebGL* ne zna ništa o tekstu ni slikama. Sve što izrađivač prikaza vidi su samo trokuti i teksture. Naivni pristup koji bi za svaki čvor kreirao zasebnu *WebGL* teksturu s rasteriziranom oznakom rezultirao bi eksplozijom broja tekstura jer moderna grafička kartica podržava simultano vezivanje 16 do 32 teksturnih jedinica, ali grafovi s tisućama čvorova zahtijevali bi tisuće tekstura što bi prikazivalo svaki čvor zasebnim pozivom i poništilo sve prednosti instanciranog prikazivanja. Rješenje je teksturni atlas (engl. `texture atlas`), jedna velika tekstura koja sadrži sve oznake i slike pakirane bez preklapanja. Svaki čvor referencira svoju regiju unutar atlasa putem UV koordinata, a svi čvorovi dijele jednu aktivnu teksturu. Na ovaj način tisuće čvorova s različitim oznakama prikazuju se jednim pozivom, a GPU uzorkuje ispravnu regiju atlasa za svaki fragment. Teško je precijeniti eleganciju ove metode i prednosti koje donosi, ali ti benefiti nisu besplatni i zahtijevaju poseban algoritam za ispravno funkcioniranje.

### 4.3.2. Algoritam pakiranja po policama

Oba atlasa - `LabelCache` i `ImageAtlas` - koriste isti algoritam za raspoređivanje pravokutnih regija unutar $2048 \times 2048$ teksture. Algoritam je poznat kao algoritam pakiranja po policama (engl. *shelf-packing*) i odabran je zbog iznimno male računske složenosti ($O(1)$ po umetanju u prosječnom slučaju) i prihvatljive iskorištenosti prostora (tipično 70-85% za oznake sličnih visina). Svaka polica ima fiksnu visinu jednaku visini prvog elementa koji je u nju smješten. Elementi koji su viši od police ne mogu u nju stati pa moraju otvoriti novu policu. Ovo znači da police sortiranog sadržaja (npr. sve oznake iste veličine fonta) imaju gotovo 100% iskorištenost, dok mješovit sadržaj raznih visina može imati veće rasipanje. Kada je atlas pun (`_allocate` vrati *null*) implementacija tiho ignorira neuspješno smještanje i element se ne prikazuje. U produkcijskim uvjetima ovo

se izuzetno rijetko dešava jer $2048 \times 2048$ RGBA tekstura može pohraniti nekoliko tisuća oznaka prosječne veličine, dok u prosjeku veći grafovi nemaju puno različitih oznaka. Atlas se u slučaju popunjavanja može pražniti pozivom `clear()` uz ponovnu rasterizaciju svih vidljivih oznaka, što je dovoljno jeftina operacija da ovo bude prihvatljivo.




**Slika 4.3.** Primjer funkcioniranja algoritma pakiranja po policama

### 4.3.3. Implementacija `LabelCache`

Kao što smo već rekli, `LabelCache` je mehanizam koji rasterizira tekstualne oznake čvorova u 2D *Canvas* element koristeći visoku rezoluciju, a zatim učitava nastalo platno kao *WebGL* teksturu. Ključni detalj implementacije jest *over-sampling*, što znači da se rasterizacija uvijek izvodi na $RASTER\_FONT\_PX = 48$ piksela visine, bez obzira na konfiguriranu veličinu fonta čvora u prostoru svijeta. Sjenčar vrhova skalira oznaku na ispravnu veličinu pri prikazivanju. Ovime se postiže da isti rasteriziran tekst ostaje oštar pri zumiranju do razine gdje piksel atlasa postane vidljiv kao piksel na zaslonu bez potrebe za ponovnom rasterizacijom pri svakom zumiranju. Funkcija `getOrCreate` je idempotentna operacija čija je vremenska složenost $O(1)$ za sve daljnje pozive s istim parametrima zbog keširanja po kompozitnom ključu. Padding od 2 piksela oko svake oznake spriječava miješanje tekstura (engl. *texture bleeding*), artefakt koji nastaje kada GPU linearno interpolira boje između susjednih texela te uzorkuje boju iz susjedne oznake na rubovima regije. Ova pojava posebno je izražena pri niskim razinama zumiranja gdje je oznaka prikazana

kao nekoliko piksela i GPU agresivno interpolira teksturu. Metoda `uploadIfDirty()` poziva se jednom po kadru prije prikazivanja, i samo ako je $\_dirty = true$ učitava cijelo platno u GPU teksturu čime postižemo *just-in-time* dohvaćanje potrebnih tekstura. Razdvajanje `texImage2D` (inicijalna alokacija) od `texSubImage2D` (ažuriranje sadržaja) važna je performansna optimizacija. Dok `texImage2D` alocira novu GPU memoriju što može uzrokovati memorijsko preopterećenje GPU, `texSubImage2D` samo kopira podatke u već alociranu memoriju i znatno je brža operacija.

### 4.3.4. Implementacija `ImageAtlas`

`ImageAtlas` dijeli isti mehanizam kao `LabelCache`, ali ima dodatnu kompleksnost asinkronog učitavanja slika. Metoda `getOrCreate(url)` mora vratiti rezultat odmah (sinkrono) jer se poziva unutar petlje za izradu prikaza, ali učitavanje slike s URL-a je nužno asinkrono. Ovo se rješava protokolom sa tri stanja i dva skupa (`cache` i `pending`):

1. **Cache hit** označava da se slika već nalazi u atlasu, pri čemu se vraća objekt `ImageAtlasEntry` s pripadajućim UV koordinatama.

2. **Pending hit (loaded)** označava da je slika učitana, ali još nije zapakirana u atlas. U tom slučaju funkcijom `_packImage()` slika se pakira, sprema u atlas i vraća se novi zapis.

3. **Pending hit (not loaded) ili novo učitavanje** rezultira povratnom vrijednošću `null`, zbog čega se čvor u trenutnom kadru prikazuje bez slike. Takvo ponašanje u praksi je neizbježno, ali ne uzrokuje značajnije probleme.

`_packImage` metoda skalira učitanu sliku na maksimalnu dimenziju (uz očuvanje omjera stranica) i crta je u odgovarajuću policu atlasa *Canvas 2D API*-jem. UV koordinate izračunavaju se s uvažavanjem paddinga da se izbjegne miješanje tekstura, kao i u `LabelCache` mehanizmu. Na taj način se kombinacijom *Canvas API* i *WebGL* može ostvariti integracija benefita koje donosi *WebGL* sa nativnošću grafike u pregledniku koju donosi *Canvas*.

## 4.4. Razine detalja

### 4.4.1. Motivacija za LOD

Pri vizualizaciji velikih grafova korisnik tipično alternira između dva rubna slučaja - pregled cjelokupnog grafa pri malom zoomu (takozvani *bird's eye view*) i detaljno istraživanje pojedinih čvorova pri velikom zoomu. Pri malom zoomu, čvorovi su na zaslonu veličine jednog do nekoliko piksela - prikazivanje oznaka i slika pri takvoj rezoluciji nema praktičnog smisla, teksurni atlas bespotrebno se uzorkuje kroz tisuće piksela, a kompleksni sjenčari troše GPU cikluse za elemente koji su vizualno nerazlučivi od jednostavnih točaka. Razine detalja (engl. *Level of Detail, LOD*) mehanizam je koji dinamički smanjuje vizualnu i računalnu kompleksnost prikaza prema trenutnom zoomu, bez perceptivnog gubitka informacije.

### 4.4.2. LOD pragovi i ponašanje

Zbog jednostavnosti i dovoljno efikasnog rada, implementacija koristi samo dvije LOD razine definirane fiksnim pragom vidljive veličine čvora. Takva je odluka donesena zbog činjenice da veličina čvora nije fiksna veličina te da je promjenjiva od strane korisnika. Primjerice, ako je korisnik dovoljno udaljio graf da čvor, koji može biti proizvoljne veličine, na zaslonu bude manji od praga (koji u našem slučaju iznosi 4px), ne koristi se `ImageAtlas` kako bi uštedjeli na relativno skupim operacijama učitavanja i upisivanja slika te ih obrade u sjenčaru. Osim toga, postoji i drugi prag (6px) za veličinu labela kako je `LabelCache` sličan, bilo kontekstualno, bilo po pitanju računalnih resurasa, mehanizam. Ove se provjere odvijaju na CPU zbog činjenice da ova provjera nije toliko skupa te imamo sve podatke o d3 zoom razini te o veličini čvorova i labela dostupne za provođenje ovih provjera. Kako su čvorovi i bridovi relativno jednostavni entiteti, naprednije tehnike poput geometrijskog LOD ili grupiranja (engl. *clustering*) na razini sjenčara nisu implementirane u sklopu ovog rada.

### 4.4.3. Implementacija LOD promjene

Provjere za LOD se odvijaju pri svakom ponovnom prikazivanju pri popunjavanju međuspremnika instanci tako da se korištenjem podataka o trenutnoj razini zumiranja odredi rezultantna veličina entiteta za iscrtavanje. Ako je u pitanju slika, funkcija `getOrCreate`

se poziva samo ako je zadovoljen uvjet za minimalnom veličinom čvora. U protivnom slučaju se UV koordinate slike se samo postave na nulu te se slika ne kreira, a ako već postoji, samo se ignorira dok ne bude izbrisana iz *cache*-a. Potpuno isti mehanizam se koristi i za labele uz drugačiji prag. Kako se sve ove provjere dešavaju kod svakog rerenderiranja, promjene se odvijaju dinamički, ovisno o korisničkim akcijama.

## 4.5. GPU simulacija sila

### 4.5.1. Pregled implementacije

GPU simulacija sila slična onoj iz d3-force biblioteke implementirana je u `GpuForceLayoutEngine` klasi koja koordinira tri aktivnosti po iteraciji: izgradnju četverostabla na CPU-u, prijenos četverostabla u GPU memoriju i pokretanje *ping-pong* koraka sjenčara fragmenata. Slijedi detaljan opis svakog od ta tri koraka s fokusom na implementacijske detalje koji nisu pokriveni teorijskim opisom u prethodnim poglavljima.

### 4.5.2. Izgradnja četverostabla

Jedna od ključnih komponenti simulacije je funkcija `buildQuadTree` koja implementira postupak izgradnje kvadrantnog stabla nad trenutnim pozicijama čvorova grafa na strani CPU. Implementacija je hibridna te ovisno o fazi funkcionira rekurzivno ili iterativno. Na ulazu prima niz trenutnih pozicija čvorova te vraća serijalizirano stablo spremno za učitavanje na GPU. Algoritam počinje linearnim prolazom kojim određuje omeđujući pravokutnik skupa točaka, a zatim konstruira kvadratni korijenski kvadrant veličine $\max(\Delta x, \Delta y) \cdot 1.01$, gdje je 1.01 faktor margine koji sprječava rubne slučajeve za čvorove na granici. Umetanje svakog tijela u stablo provodi se iterativno petljom s gornjom granicom od 50 razina dubine kako bi izbjegli preplavljivanje memorije. Prazni list prima elemente direktno, dok list koji već sadrži neki element se dijeli tako da se postojeći element premješta u odgovarajući podkvadrant pa se umetanje nastavlja. Unutarnji čvor samo prosljeđuje umetanje u kvadrant kojemu koordinate tijela odgovaraju. Kvadrant se numerira kao $q = 2b_y + b_x, \quad$ gdje je $\mathrm{NW} = 0$, $\mathrm{NE} = 1$, $\mathrm{SW} = 2$, $\mathrm{SE} = 3$. Po završetku umetanja svih elemenata poziva se rekurzivna funkcija `computeAggregates` koja obilazi stablo odozdo prema gore i za svaki unutarnji čvor izračunava ukupni naboj kao sumu naboja djece te težišne koordinate kao nabojima ograničen prosjek. Rezul-

tat izgradnje serijalizira se u tri zasebna Float32Array niza - `treeData`, `treeChildren` i `treeGeometry`, svaki dimenzija $texWidth^2 * 4$, gdje je texWidth $= \lceil\sqrt{\text{nodeCount}}\rceil$. Svaki čvor stabla zauzima jedan texel tako da treeData nosi težišne koordinate, ukupni naboj i tip čvora (negativna vrijednost A kanala označava list, pozitivna unutarnji čvor s veličinom kvadranta), `treeChildren` nosi četiri indeksa djece (-1 za prazni kvadrant), a `treeGeometry` koordinate gornjeg lijevog ugla kvadranta i njegovu veličinu. Padding do $texWidth^2$ popunjava se praznim vrijednostima.

### 4.5.3. `Ping-pong` simulacija

Za pohranu stanja čvorova korištena su dva para resursa, teksture (`stateTexA` i `stateTexB`) te slikovne međuspremnike (`fboA` i `fboB`). Zastavica `pingPong` određuje koji par služi za čitanje, a koji za pisanje u trenutnom koraku. Ako je zastavica istinita, čita se iz `stateTexA`, a rezultat se piše u `stateTexB` vezanim na `fboB`, i obrnuto nakon svakog koraka. Zašto to uopće koristimo? Kako čvorovi u svakom koraku trebaju čitati trenutno stanje i upisivati novo stanje, bilo bi ekstremno nepouzdano izvoditi to u jednom koraku jer bi postojala vjerojatnost čitanja već promijenjenog stanja te potencijalno pogrešnih rezultata. Ovakav mehanizam elegantno sprečava nastanak tog problema, a radi na način da metoda `simulateGPUStep` postavlja uniforme (broj čvorova, alpha, parametre sila, dimenzije tekstura stabla i susjednosti), veže teksture na jedinice 0-6 (stanje, fiksni čvorovi, tri teksture stabla, dvije teksture susjednosti) te upisuje te podatke u teksturu veličine $texWidth \times texWidth$. Svaki fragment, identificiran preko `gl_FragCoord`, računa `nodeId` i, ako je manji od broja čvorova, izvršava cijeli integracijski korak, a inače piše nulu. Fiksni čvorovi (koji imaju postavljen `fx/fy`) u `fixedTex` teksturi uzrokuju da sjenčar za taj texel vrati fiksnu poziciju i nultu brzinu bez primjene sila. Nakon crtanja `pingPong` se invertira, tako da sljedeći korak čita upravo upisano stanje. Pri inicijalnom učitavanju ili nakon centriranja na CPU-u obje se teksture stanja eksplicitno usklađuju (`_syncStateToGPU`), a `pingPong` se resetira na početnu vrijednost kako bi se izbjegla desinhronizacija. Promjene pojedinih čvorova tijekom povlačenja mogu se upisati i točkasto (`texSubImage2D` u oba state texela i u `fixedTex`) bez punog ponovnog uploada grafa. Stabilizacija grafa pokreće petlju u chunkovima veličine `CHUNK_SIZE`, dok povlačenje čvora koristi `requestAnimationFrame` petlju s jednim GPU korakom po kadru i vlastitim prigušivanjem alpha parametra koji upravlja logikom simulacije, neovisno o

glavnoj stabilizaciji.

### 4.5.4. Privlačne sile bridova na GPU-u

Bridovi se ne simuliraju parovima u sjenčaru, već preko unaprijed izgrađene strukture susjednosti na CPU-u koja se izgrađuje još jednom preko pomoćne metode `buildAdjacency`. Za svaki nesusmjereni brid u grafu generiraju se dva usmjerena zapisa (izvor prema cilju i obrnuto). Za svaki čvor `i` računa se stupanj te se u offsets teksturi pohranjuje početni indeks i broj susjeda u `edges` teksturi. Svaki zapis u `edges` teksturi sadrži indeks ciljnog čvora, željenu duljinu opruge (engl. *link distance*), jačinu i bias smjera (`dirBias`) koji odgovara d3-ovom rastavljanju sile na dva kraja brida. Susjednost se gradi pri aktivaciji simulacije ili pri promjeni skupa bridova, sprema se u `_cachedAdjacency` i učitava se u dvije `RGBA32F` teksture s vlastitim `texWidth` vrijednostima. U sjenčaru fragmenata zatim čvor čita svoj raspon susjeda iz offsets teksture, zatim u petlji dohvaća zapise iz `edges` teksture. Za opružnu silu koriste se predviđene pozicije obje strane iz ulaznog stanja ($xy + zw$), a ne trenutno akumulirana brzina samo jedne strane čime se izbjegava sustavna asimetrija i pretjerano grupiranje pri visokom alpha tijekom povlačenja, što bi nastalo da jedna strana već uključuje $manyBody$ doprinos u $zw$, a druga ne. U slučaju ako graf nema bridova ili susjednost nije izgrađena, uniform $uHasLinks$ ostaje nula i taj dio sjenčara se preskače.

### 4.5.5. Čitanje pozicija i sinkronizacija s izrađivačem prikaza

Simulator i izrađivač prikaza ne dijele *WebGL* kontekst niti teksture. Jedini zajednički model podataka su *JavaScript* objekti simulacijskih čvorova ($x, y, vx, vy$), koje `GpuForceLayoutEngine` ažurira metodom `readbackFromGPU` na način da veže odgovarajući slikovni međuspremnik (suprotan od `pingPong` zastavice nakon zadnjeg pisanja), poziva `readPixels` u `RGBA FLOAT` formatu i prepisuje prvih N texela u polje `_nodes`. Tipičan redoslijed u jednom koraku stabilizacije ili povlačenja jest: `readback` (sinkronizacija GPU -> CPU), `flushDirtyNodes` (upis povučenih ili fiksiranih čvorova u GPU teksture), izgradnja i upload četverostabla, jedan ili više `simulateGPUStep` poziva, ponovni readback, `applyCentering` na CPU-u (pomak svih slobodnih čvorova prema cilju centriranja), emit događaja
`SIMULATION_PROGRESS`, `SIMULATION_END` ili `NODE_DRAG`, te u OrbView poziv

`_graph.setNodePositions` i
`renderer.render`. Izrađivač prikaza čita pozicije isključivo iz grafa te nema direktnog pristupa simulacijskim teksturama. Zakazivanje sljedećeg chunka stabilizacije ide preko `MessageChannel` mehanizma (umjesto `setTimeout` koji uvijek ima minimalan delay od `4ms` što je isprobano i ustanovljeno da je neprihvatljivo za veći broj iteracija) kako bi se smanjila minimalna odgoda između koraka i UI ostao responzivan. `CHUNK_SIZE` od 1 znači da se četverostablo ponovno gradi gotovo pri svakom koraku što je skupo na CPU-u, ali drži Barnes-Hut aproksimaciju usklađenom s brzo mijenjajućim pozicijama.

### 4.5.6. Ispravljanje grešaka GPU simulacije

Tijekom razvoja GPU inačice algoritma razvijen je opsežan sloj za ispravljanje grešaka namijenjen postupnoj, korak po korak, validaciji rezultata. Plan je podijeljen u slojeve od provjere jednog cijelog koraka bez sila te izoliranog testiranja određivanja pozicija sve do testiranja link i manyBody sila uz usporedbu s referentnim CPU izračunom te testova na većem broju čvorova (npr. 20) za otkrivanje grešaka koje se pojave samo uz dublje stablo što je bilo neophodno nakon naivnog brisanja sloja za ispravljanje grešaka nakon uspješne potvrde da simulacija radi na malom broju čvorova (jedna grupa, do 10 čvorova) te kasnije provjere koja je ustanovila probleme na većim grafovima. Napravljene su i pomoćne metode koje omogućuju čitanje bilo koje simulacijske teksture u platno i izvoz PNG datoteka, koji, iako nemaju preveliku vizualizacijsku vrijednost, demonstriraju da sustav radi (iako se čak i nakon toga postavlja pitanje radi li ispravno). Iako su te funkcije izbrisane iz produkcijskog koda kako bi se smanjila veličina rezultantne biblioteke, metodologija postupnog testiranja na skupovima podataka različitih veličine je u velikoj mjeri doprinjela uspješnoj izgradnji ove biblioteke.

# 5. Eksperimenti i rezultati

## 5.1. Metodologija mjerenja i testna okolina

S ciljem kvantitativnog određivanja performansi razvijenog rješenja provedena je eksperimentalna evaluacija oba razvijena podsustava, pogona za određivanje raspoređivanja čvorova i *WebGL* izrađivača prikaza. Kako bi odredili brzinu simulacije raspoređivanja čvorova i brzinu vizualizacije, za svaki podustav definiran je zaseban protokol mjerenja koji izolira promatranu varijablu od utjecaja ostalih komponenti sustava. Sva mjerenja provedena su u identičnoj testnoj okolini opisanoj tablicom 5.1.

Tablica 5.1. Testna okolina

| **Komponenta** | **Specifikacija** |
|---|---|
| Operacijski sustav | Windows 11 |
| Web preglednik | Google Chrome 148 |
| WebGL verzija | WebGL 2.0 (OpenGL ES 3.0) |
| CPU | AMD Ryzen™ 9 9950X3D |
| GPU | AMD Radeon™ Graphics (2 Cores, 2200 MHz) |

### 5.1.1. Protokol mjerenja simulacije raspoređivanja

Za skupove podataka korišteni su sintetski grafovi generatorom koji generira grafove s izraženom klasterskom strukturom. Takav tip grafova je posebno zahtjevan za algoritme raspoređivanja jer zahtijeva dovoljno iteracija da klasteri međusobno razjasne svoju prostornu odvojenost. Veličine grafova raspoređene su od 1000 do 50000 čvorova, s dvostrukim brojem bridova u odnosu na čvorove, kako bi dobili uvid ne samo u kvalitativne karakteristike sustava, nego i tendencije ovisno o broju podataka. Mjerenje brzine simulacije raspoređivanja provodi se u tzv. *batch mode*-u na način da simulacija izvodi sve iteracije bez prekida i bez izrade prikazivanja između koraka, čime se eliminira dopri-

nos izrađivača prikaza iz izmjerenog vremena. Ovaj mehanizam je ključan za pouzdanu usporedbu, budući da je inicijalni naivni pristup u kojemu se jedan korak simulacije i jedan kadar izrade prikazivanja izmjenjuju miješao dva inače neovisna mehanizma čime je onemogućuio izoliranu i nepristranu analizu svakog. Mjerena veličina je prosječno medijan trajanja jedne iteracije (ms/iter) kroz tri neovisna mjerenja.

### 5.1.2. Protokol mjerenja brzine izrade prikaza

Kako je za adekvatnu i smislenu vizualizaciju prikladnije koristiti pravilnije stukture podataka, za mjerenja brzina prikazivanja korišten je sintetski rešetkasti skup podataka za iste količine čvorova i bridova kao i u prethodnom mjerenju. Mjerenje brzine prikazivanja provodi se zasebnim protokolom od mjerenja brzine simulacije jer je cilj kvantificirati koliko brzo izrađivač prikaza može prikazati scenu pri interaktivnom korištenju, neovisno o složenosti algoritma raspoređivanja. FPS uzorkuje se kontinuirano kroz trajanje simulacije mjerenjem vremenske razlike između uzastopnih poziva `requestAnimationFrame` petlje, a mjerene veličine su prosječni FPS (avg), medijalni FPS (p50) i percentilni FPS (p95). Ključna arhitekturalna odluka koja utječe na mjerenje jest način sinkronizacije simulacije i izrađivača prikaza. U originalnoj implementaciji, jedan korak simulacije i jedan kadar prikazivanja izmjenjuju se sekvencijalno unutar iste `requestAnimationFrame` petlje. Ovo uzrokuje međusobno blokiranje dvaju procesa jer spori simulacijski koraci izravno potiskuju kadrove prikazivanja i obrnuto, čineći izmjereni FPS ovisnim o brzini simulacije i obratno. Taj problem rješavamo na način da unaprijed generiramo istu strukturu grafa te na dobivenom skupu podataka provodimo mjerenja traženih veličina.

## 5.2. Rezultati mjerenja

U ovom poglavlju prikazani su empirijski rezultati testiranja kroz dva odvojena eksperimentalna protokola. Sva mjerenja provedena su kontrolirano kako bi se izolirale performanse algoritama raspoređivanja od samog prikazivanja scene.

### 5.2.1. Rezultati mjerenja simulacije raspoređivanja

Za evaluaciju računalne učinkovitosti algoritama raspoređivanja mjereno je prosječno trajanje jedne iteracije izraženo u milisekundama po iteraciji ($ms/iter$), pri čemu manji

broj označava brže izvršavanje i bolji rezultat. Veličina testnih sintetskih grafova skalira se od 1000 do 50000 čvorova, uz fiksni omjer gdje je broj bridova dvostruko veći od broja čvorova ($E = 2N$). Tablica 5.2. prikazuje izravnu usporedbu Orb biblioteke (u postojećoj CPU i razvijenoj GPU izvedbama) s popularnim postojećim bibliotekama za vizualizaciju mreža, poput Sigma.js[12], te s nekima od opisanih u uvodnom poglavlju. Nedostatak podataka za alat Cytoscape.js (nedostajući podaci označeni su s *n/a* (engl. *not available*)) označava da biblioteka nije bila u stanju uspješno završiti mjerenje za grafove veće od 5000 čvorova zbog prekoračenja vremenskih ograničenja.

Tablica 5.2. Usporedba prosječnog trajanja jedne iteracije raspoređivanja (ms)

| n (čvorovi) | Orb GPU | Orb CPU | Sigma.js | vis.js | Cytoscape.js |
|---|---|---|---|---|---|
| 1 000 | 1.84 | 0.9 | 0.93 | 2.93 | 15.79 |
| 5 000 | 6.88 | 4.6 | 5.61 | 19.52 | 403.45 |
| 10 000 | 13.33 | 50.7 | 12.26 | 41.02 | n/a |
| 20 000 | 27.75 | 113.0 | 28.62 | 98.98 | n/a |
| 30 000 | 45.72 | 181.7 | 45.63 | 166.75 | n/a |
| 40 000 | 65.86 | 269.9 | 66.87 | 252.74 | n/a |
| 50 000 | 81.00 | 367.5 | 82.26 | 336.39 | n/a |

Kako bi se detaljnije analiziralo ponašanje predloženog rješenja, specifično želja ustanoviti ima li razvijeno rješenje složenost približno jednaku teorijskoj Barnes-Hut složenosti $O(nlogn)$, Tablica 5.3. donosi izolirani pregled performansi GPU algoritma kroz fiksni prozor od 100 iteracija. Ograničenje od 100 iteracija uvedeno je kako bi svi testovi bili objektivni i pod jednakim uvjetima. Uz metriku $ms/iter$, ovdje se uvodi i propusnost izražena u iteracijama u sekundi ($it/s$), koja predstavlja teoretski ekvivalent brzine osvježavanja (FPS) kada bi se raspored računao u stvarnom vremenu. Dodatno, stupac *JS Heap (MB)* prati zauzeće radne memorije *JavaScript* procesa, pružajući uvid u memorijsku učinkovitost i utjecaj alokacije objekata na stabilnost sustava.

### 5.2.2. Rezultati mjerenja brzine izrade prikaza

Mjerne jedinice i mjerene veličine za evaluaciju grafičkog podsustava u Tablici 5.4. obuhvaćaju nekoliko ključnih parametara performansi interaktivnog rada:

- **avg FPS i p95 FPS** - Prosječan broj kadrova u sekundi te njegova 95-postotna granična vrijednost (percentil), koji opisuju glatkoću i stabilnost vizualizacije pod opterećenjem.

Tablica 5.3. Analiza performansi GPU algoritma za 100 iteracija u ovisnosti o veličini grafa

| Čvorovi ($N$) | Vrijeme (ms) | ms/iter | it/s | JS Heap (MB) |
|---|---|---|---|---|
| 1000 | 218.8 | 2.19 | 457.04 | 37.7 |
| 5000 | 686.4 | 6.86 | 145.69 | 44.2 |
| 10000 | 1541.7 | 15.42 | 64.86 | 198.5 |
| 20000 | 3200.2 | 32.00 | 31.25 | 75.0 |
| 30000 | 5586.2 | 55.86 | 17.90 | 153.1 |
| 40000 | 6953.7 | 69.54 | 14.38 | 163.7 |
| 50000 | 8984.6 | 89.85 | 11.13 | 283.5 |

- **render avg (ms)** - Izolirano prosječno trajanje isključivo faze crtanja po jednom kadru, bez utjecaja logike aplikacije.
- **TTFR (ms)** - Vrijeme do prvog prikaza (engl. *Time to First Render*), odnosno ukupno kašnjenje od pokretanja do inicijalnog iscrtavanja grafa na zaslonu.

## 5.3. Analiza rezultata

U ovom poglavlju ćemo provesti detaljnu analizu i sintezu rezultata dobivenih kroz mjerne protokole simulacije raspoređivanja i brzine izrade prikaza i navedenih u prethodnim tablicama. Glavni cilj analize je utvrditi kako se različiti arhitektonski pristupi, posebice rješenja temeljena na procesoru (CPU) u usporedbi s rješenjima ubrzanima grafičkim procesorom (GPU), ponašaju prilikom skaliranja količine podataka od 1000 do 50000 čvorova. Dobiveni rezultati jasno naglašavaju granice konvencionalnih jednodretvenih pristupa u pregledniku i demonstriraju nužnost paralelizacije i asinkrone sinkronizacije za rad s velikim skupovima podataka.

### 5.3.1. Analiza performansi algoritama raspoređivanja

Na temelju podataka iz tablice 5.2., vidljivo je da Sigma.js i razivjen Orb GPU pogoni ostvaruju uvjerljivo najbolje i gotovo podjednake rezultate pri maksimalnom opterećenju od 50000 čvorova (oko 81-82 ms/iter). Zanimljiv je fenomen na manjim grafovima (do 5000 čvorova) gdje su Orb CPU i Sigma.js neznatno brži od Orb GPU-a što se pripisuje fiksnom trošku (engl. *overhead*) inicijalizacije, prebacivanja podataka na grafičku karticu i pokretanja *kernel* funkcija, što kod manjih grafova nadmašuje samu uštedu paralelizacije. S druge strane, vis.js i Orb CPU pokazuju klasičan nelinearan rast vremena

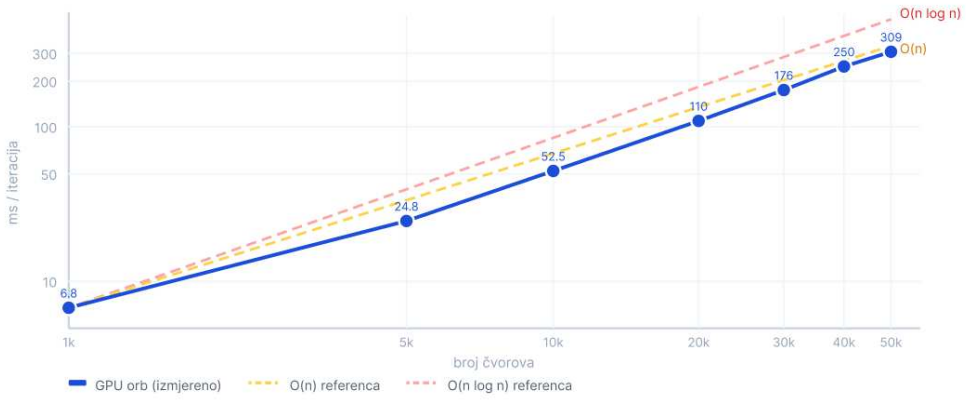
Tablica 5.4. Skalabilnost brzine izrade prikaza prema veličini grafa

| Alat | n | Heap (MB) | avg FPS | p95 FPS | render avg (ms) | TTFR (ms) |
|---|---|---|---|---|---|---|
| Orb GPU | 1000 | 78 | 60.3 | 59.5 | 0.14 | 10 |
| Orb GPU | 5000 | 36 | 60.3 | 59.5 | 0.52 | 33 |
| Orb GPU | 10000 | 47 | 60.3 | 59.5 | 1.06 | 72 |
| Orb GPU | 20000 | 150 | 60.3 | 59.5 | 2.68 | 130 |
| Orb GPU | 30000 | 119 | 60.3 | 59.5 | 4.18 | 215 |
| Orb GPU | 40000 | 127 | 60.3 | 59.5 | 5.28 | 283 |
| Orb GPU | 50000 | 314 | 60.3 | 59.5 | 6.18 | 300 |
| Orb CPU | 1000 | 55 | 60.2 | 59.5 | 0.9 | 23 |
| Orb CPU | 5000 | 40 | 57.0 | 30.0 | 4.3 | 31 |
| Orb CPU | 10000 | 38 | 41.3 | 20.0 | 8.2 | 39 |
| Orb CPU | 20000 | 84 | 21.9 | 10.0 | 21.2 | 76 |
| Orb CPU | 30000 | 132 | 13.6 | 6.7 | 40.2 | 124 |
| Orb CPU | 40000 | 148 | 10.0 | 4.0 | 79.3 | 427 |
| Orb CPU | 50000 | 198 | 7.9 | 3.3 | 110.4 | 481 |
| Sigma.js | 1000 | 31 | 57.5 | 59.9 | 1.1 | 62 |
| Sigma.js | 5000 | 62 | 57.2 | 59.9 | 4.9 | 64 |
| Sigma.js | 10000 | 45 | 60.4 | 59.5 | 6.7 | 63 |
| Sigma.js | 20000 | 68 | 48.1 | 29.9 | 14.0 | 117 |
| Sigma.js | 30000 | 99 | 33.9 | 30.0 | 27.8 | 158 |
| Sigma.js | 40000 | 158 | 26.5 | 20.0 | 35.2 | 163 |
| Sigma.js | 50000 | 386.7 | 10.4 | 10 | 97.6 | 647 |
| vis.js | 1000 | 47 | 60.3 | 59.5 | 16.6 | 64 |
| vis.js | 5000 | 144 | 57.1 | 59.5 | 17.6 | 111 |
| vis.js | 10000 | 166 | 51.1 | 29.9 | 19.6 | 254 |
| vis.js | 20000 | 577 | 20 | 20 | 50.6 | 561 |
| vis.js | 30000 | 583 | 11.0 | 8.6 | 92.7 | 747 |
| vis.js | 40000 | 1111.5 | 7.3 | 7.5 | 142.1 | 1085 |
| vis.js | 50000 | 1376.8 | 5.2 | 5.0 | 201.2 | 1383 |
| Cytoscape | 1000 | 92 | 60.1 | 59.5 | 16.6 | 75 |
| Cytoscape | 5000 | 154 | 52.1 | 29.9 | 19.0 | 332 |
| Cytoscape | 10000 | 261 | 32.9 | 12.0 | 29.6 | 713 |
| Cytoscape | 20000 | 488 | 26.1 | 6.0 | 39.8 | 1604 |
| Cytoscape | 30000 | 699 | 10.0 | 4.0 | 79.3 | 2132 |
| Cytoscape | 40000 | 1068 | 3.3 | 1.5 | 308.9 | 2740 |
| Cytoscape | 50000 | 4311 | 1.9 | 1.8 | 566.3 | 4766 |

izvršavanja, dok Cytoscape.js potpuno zakazuje u skalabilnosti - s preko 400 ms po iteraciji na samo 5000 čvorova postaje neupotrebljiv za interaktivni rad te nije bio u stanju završiti mjerenja za veće grafove.

### 5.3.2. Analiza GPU algoritma raspoređivanja

Detaljnija analiza izoliranog Orb GPU algoritma u tablici 5.3. potvrđuje visoku predvidljivost i stabilnost sustava. Vrijeme po iteraciji raste stabilno i proporcionalno kompleksnosti grafa, što dokazuje uspješno mapiranje sila na paralelne GPU dretve. Fluktuacije uočene u stupcu *JS Heap* memorije (poput skoka na 198.5 MB pri 10000 čvorova i kasnijeg pada na 75.0 MB pri 20000) ne indiciraju curenje memorije, već su izravna posljedica nedeterminističkog okidanja ugrađenog sakupljača smeća (engl. *Garbage Collector*) unutar *JavaScript* okruženja tijekom intenzivnih operacija. Što je još zanimljivije, dobiveno rješenje pokazuje složenost ne samo $O(nlogn)$, ali i bolju od linearne što možemo objasniti ograničenjem maksimalne dubine spuštanja Barnes-Hut simulacije koju smo prethodno uveli. Naravno, to ne znači da je rješenje idealno ili barem prihatljivo brzo na još većim grafovima ali pokazuje stabilnost i primjenjivost ovog rješenja. Iako se na grafovima većim od stotina tisuća čvorova gubi interaktivnost, razvijeno rješenje se može koristiti za unaprijedni statički izračun pozicija i daljnju vizualizaciju grafova.

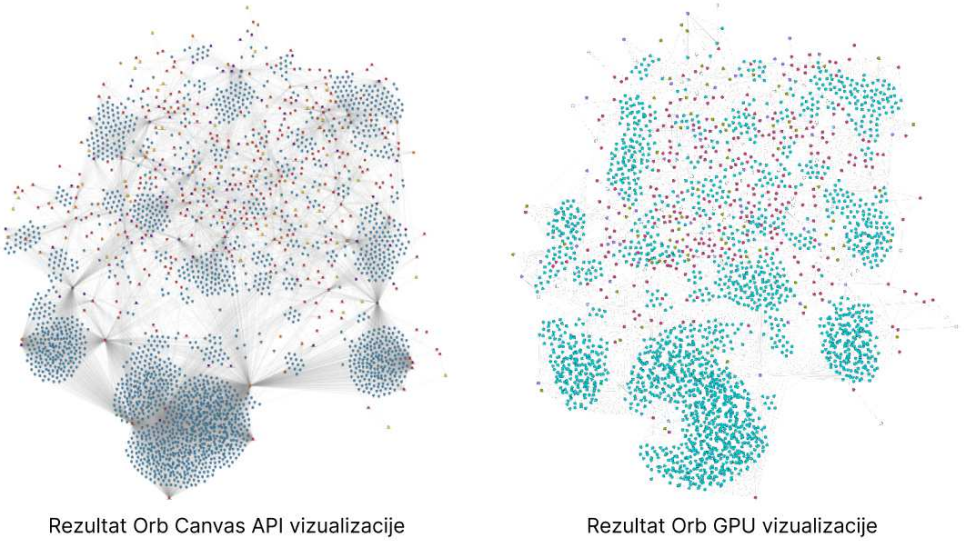



**Slika 5.1.** Graf ovisnosti trajanja jedne iteracije Orb GPU o broju čvorova

### 5.3.3. Analiza brzine izrade prikaza i skalabilnosti FPS-a

Mjerenja u tablici 5.4. najpreciznije demonstriraju prednost arhitekture alata Orb GPU kod vizualizacija grafova pravilne strukture. Dok svi ostali testirani alati bilježe velik pad performansi, Orb GPU zadržava maksimalnih i stabilnih 60.3 FPS-a (uz zanemariv

pad p95 percentila na 59.5 FPS-a) kroz cijeli raspon skaliranja, sve do 50000 čvorova. Kod Orb CPU inačice vidljiv je izravan utjecaj rasta grafa na faza crtanja (*render avg* skače na 110.4 ms), što degradira izvedbu na kritičnih 7.9 FPS-a. Sigma.js se zahvaljujući *WebGL*-u dobro drži do 40000 čvorova (26.5 FPS-a), no vis.js i Cytoscape.js (koji se oslanjaju na standardni *HTML5 Canvas* odnosno *SVG*) pokazuju izrazitu neoučinkovitost: Cytoscape.js pri 40000 čvorova troši preko 1 GB memorije, ima katastrofalan TTFR od 2.7 sekundi i pada na praktički statičnih 3.3 FPS-a.




**Slika 5.2.** Primjer vizualizacije velikog skupa podatka korištenjem Canvas API i WebGL

# 6. Zaključak

Rezultat ove poprilično opširne avanture jest proširenje postojećeg alata otvorenog kôda Memgraph Orb koje je donijelo poboljšanu i skalabilniju arhitekturu sustava uz nekoliko novih, GPU-akceleriranih modula ne samo za jednostavnu (iako i stilovima vrlo bogatu) vizualizaciju, nego i za simulaciju sila temeljenu na Barnes-Hut algoritmu. Ove promjene omogućuju korisnicima efikasniju i bržu vizualizaciju velikih skupova podataka bez gubitka postojećih funkcionalnosti, što je potvrđeno eksperimentalnim rezultatima.

GPU izrađivač prikaza, razvijen u sklopu ovog rada, postiže rezultate usporedive po pitanju brzine i memorijskih troškova s postojećim na tržištu bibliotekama i rješenjima uz veću fleksibilnost u stilizaciji vizualiziranih grafova. S druge strane, GPU simulacija sila postiže empirijsku složenost čak i bolju od teorijske $O(nlogn)$, što je objašnjivo činjenicom da masivna paralelizacija amortizira utjecaj logaritamske komponente složenosti. Iako se rezultati vizualno neznatno razlikuju od dobivenih postojećim CPU rješenjem, po pitanju raspoređivanja čvorova razvijeno rješenje ne samo postiže paritet s tržišnim, nego je u nekim slučajevima i efikasniji.

Naravno, razvijeno rješenje ima i neka ograničenja. Primjerice, mehanizam izračuna raspoređivanja čvorova se ne može u potpunosti odvijati na GPU zbog sinkrone prirode određenih koraka, što povećava cijenu sinkronizacije između GPU i CPU te je ovo rješenje teže primjenjivo na vizualizaciju grafova sa stotinama tisuća čvorova. Osim toga, vizualizacija grafova s velikim brojem zakrivljenih bridova također predstavlja problem, što je djelomično rješeno LOD mehanizmom ali se također treba uzeti u obzir pri korištenju.

Potencijal za buduća istraživanja i proširenja je ogroman i može se grupirati u tri razine. Na razini izrade prikaza, integracija s WebGPU sjenčarima eliminirala bi potrebu

za *ping-pong* mehanizmom i omogućila efikasnije izravno čitanje i pisanje pozicija. Na razini algoritama raspoređivanja, istraživanje GPU-paralelnih algoritama izgradnje kvadrantnog stabla, primjerice radix sort po Mortonovim kodovima s naknadnom paralelnom konstrukcijom, moglo bi eliminirati preostali trošak komunikacije između CPU i GPU. Na razini biblioteke, eksperimentalna podrška za 3D prikaz grafova i integracija s graf bazama podataka putem Bolt protokola u stvarnom vremenu prirodna su proširenja koja su arhitekturalno pripremljena postojećim slojem modela podataka.

Razvijeno rješenje dostupno je kao dio Memgraph Orb biblioteke otvorenog kôda, čime doprinosi širem ekosustavu alata za vizualizaciju grafova u web okruženju i pruža solidnu osnovu za daljnja istraživanja GPU-akcelerirane vizualizacije i analize mrežnih podataka u preglednickom okruženju.

# Literatura


[1] Memgraph, “Orb: Open-source graph visualization library”, https://github.com/memgraph/orb, 2024.

[2] I. Herman, G. Melançon, i M. S. Marshall, “Graph visualization and navigation in information visualization: A survey”, *IEEE Trans. Vis. Comput. Graph.*, sv. 6, str. 24–43, 2000. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:239537

[3] Khronos Group, “OpenGL ES 3.0 Specification”, https://registry.khronos.org/OpenGL/specs/es/3.0/es_spec_3.0.pdf, Khronos Group, Specification, 2012.

[4] P. Eades, “A heuristic for graph drawing”, 1984. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:63936426

[5] T. M. J. Fruchterman i E. M. Reingold, “Graph drawing by force-directed placement”, *Software: Practice and Experience*, sv. 21, 1991. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:31468174

[6] J. H. Barnes i P. Hut, “A hierarchical o(n log n) force-calculation algorithm”, *Nature*, sv. 324, str. 446–449, 1986. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:4267861

[7] M. Burtscher i K. Pingali, “An efficient cuda implementation of the tree-based barnes hut n-body algorithm”, 2011. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:14713290

[8] M. Bostock, V. Ogievetsky, i J. Heer, “$D^3$ data-driven documents”, *IEEE Transactions on Visualization and Computer Graphics*, sv. 17, str. 2301–2309, 2011. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:14970263

[9] M. Franz, C. T. Lopes, G. Huck, Y. Dong, S. O. Sümer, i G. D. Bader, “Cytoscape.js: a graph theory library for visualisation and analysis”, *Bioinformatics*, sv. 32, str. 309 – 311, 2015. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:9651322

[10] M. Harris, “Mapping computational concepts to GPUs”, u *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ur. Addison-Wesley, 2005., pog. 31, str. 493–508, dostupno na: https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-31-mapping-computational.

[11] Khronos Group, “WebGL 2.0 Specification”, https://registry.khronos.org/webgl/specs/latest/2.0/, Khronos Group, Specification, 2017.

[12] M. Jacomy, T. Venturini, S. Heymann, i M. Bastian, “Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software”, *PLoS ONE*, sv. 9, 2014. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:14763635

# Sažetak


## Vizualizacija velikih mrežnih grafova pomoću WebGL-a

Oleksandr Ichenskyi

Vizualizacija velikih mrežnih grafova u web okruženju predstavlja značajan tehnički izazov zbog ograničenja klasičnih pristupa temeljenih na *DOM* manipulaciji i CPU-baziranom crtanju. Ovaj rad istražuje mogućnosti WebGL 2 tehnologije za visoko performantnu vizualizaciju grafova s velikim brojem čvorova i bridova. Razvijen je sustav koji se sastoji od tri međusobno povezane komponente: izrađivač prikaza temeljen na instanciranom prikazivanju koji u jednom pozivu za crtanje prikazuje tisuće čvorova različitih oblika, bridova i tekstualnih oznaka; GPU-akcelelirani simulator sila koji Barnes-Hut algoritam implementira u GLSL sjenčarima uz korištenje *ping-pong* tehnike izmjene slikovnih međuspremnika; te sustav teksturnih atlasa za efikasno upravljanje oznakama i slikama. Razvijeno rješenje integrirano je u biblioteku Memgraph Orb otvorenog kôda. Eksperimentalna evaluacija na sintetskim i stvarnim skupovima podataka pokazuje značajna poboljšanja u broju sličica u sekundi i brzini konvergencije simulacije u usporedbi s rješenjima kao što su Sigma.js, Cytoscape.js i d3-force, posebice pri grafovima s više od deset tisuća čvorova.



**Ključne riječi:** vizualizacija grafova, WebGL 2, instancirano prikazivanje, algoritam Barnes-Hut, simulacija sila, GPGPU, teksturni atlas

# Abstract


## Visualization of large network graphs using WebGL

Oleksandr Ichenskyi

Visualizing large network graphs in web environments is a significant technical challenge due to the limitations of classical approaches based on DOM manipulation and CPU-side rendering. This thesis explores the capabilities of WebGL 2 for high-performance graph visualization with large numbers of nodes and edges. A system consisting of three interconnected components was developed: an instanced-rendering-based renderer that draws thousands of differently shaped nodes, edges, and text labels in a single draw call; a GPU-accelerated force simulator that implements the Barnes-Hut algorithm in GLSL shader programs using a ping-pong framebuffer technique; and a texture atlas system for efficient management of rasterized labels and node images. The developed solution is integrated into the open-source Memgraph Orb library. Experimental evaluation on synthetic and real-world datasets demonstrates significant improvements in frames per second and simulation convergence speed compared to solutions such as Sigma.js, Cytoscape.js, and d3-force, particularly for graphs exceeding ten thousand nodes.



**Keywords:** graph visualization, WebGL 2, instanced rendering, Barnes-Hut algorithm, force-directed layout, GPGPU, texture atlas