SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 850

OPTIMIZACIJA SIMULACIJE GRAVITACIJSKE SILE U SUSTAVU S VELIKIM BROJEM ČESTICA

Mario Jalšovec

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 850

OPTIMIZACIJA SIMULACIJE GRAVITACIJSKE SILE U SUSTAVU S VELIKIM BROJEM ČESTICA

Mario Jalšovec

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 3. ožujka 2025.

DIPLOMSKI ZADATAK br. 850

Pristupnik:	**Mario Jalšovec (0036529377)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Optimizacija simulacije gravitacijske sile u sustavu s velikim brojem čestica**

Opis zadatka:

Proučiti problematiku simulacije izračuna djelovanja polja gravitacijske sile u okruženju s velikim brojem objekata kao što je svemir. Izraditi simulacijski model koji uključuje djelovanje gravitacijske sile i algoritam rješavanja sudara te implementaciju na jedinicama CPU (Central processing unit) i GPU (Graphics processing unit). Posebno obratiti pažnju na optimizaciju i optimizacijske strukture podataka u okviru navedene problematike. Na nizu primjera ostvariti testiranje i usporedbu dobivenih rezultata. Analizirati i ocijeniti postignuta rješenja. Diskutirati upotrebljivost rješenja kao i moguća proširenja.

Rok za predaju rada: 4. srpnja 2025.

Sadržaj

Uvod

Simulacija N-tijela (eng. *N-body simulation*) je računalna simulacija sustava čestica u kojem su prisutne određene fizičke sile. Ovakve simulacije imaju široke primjene u znanstvenim disciplinama kao što su fizika, kemija i astronomija. Koriste se da bi se lakše i/ili jeftinije mogli proučavati određeni fizički sustavi koji se u stvarnosti sastoje od velikog broja čestica. Simulacije N-tijela obuhvaćaju sve od simulacija najmanjih čestica kao što su atomi i molekule do simulacija čitavih galaksija. [1]

Osim u znanosti ovakve simulacije se koristi i u drugim industrijama kao što su industrija video igara ili filmska industrija jer je i tamo ponekad potrebno simulirati fizičke fenomene u svrhu ostvarivanja uvjerljivih vizualnih efekata.

Ovaj rad će se baviti proučavanjem i implementacijom simulacije N-tijela koja može u stvarnom vremenu simulirati utjecaj gravitacijske sile između što većeg broja čestica.

1 Naivni algoritam simulacije gravitacije

Naivni algoritma za izračun gravitacijskog utjecaja u ovoj simulaciji sastoji se od tri glavna koraka. Prvo je potrebno izračunati gravitacijsku silu na svaku česticu, onda riješiti sudare između čestica ažurirati brzine i pozicije čestica.

1.1 Izračun utjecaja gravitacijske sile

Za izračun utjecaja gravitacije postoji dobro poznat Newtonov zakon gravitacije koje se može vidjeti na izrazu (1.1) [2] s kojim se vrlo jednostavno može simulirati gravitacijski utjecaj između dvije čestice. No pošto se u simulaciji radi s akceleracijama, a na silama potrebno je izračunati iznos akceleracije na pojedinu česticu prema izrazu (1.2).

$$F = G\,\frac{mM}{r^2} \tag{1.1}$$

$$a = G\,\frac{M}{r^2} \tag{1.2}$$

$$a_i = \sum_{i \neq j}^{N} \frac{G\,m_j}{{r_{ij}}^2}\,\widehat{r_{ij}} \tag{1.3}$$

Da bi se izračunala akceleracija na svaku čestica u simulaciji potrebno je primijeniti formulu (1.3) za svaku česticu. Pseudo kod za taj izračun bi izgledao ovako.

```
za i od 0 do N-1:
      za j od i + 1 do N-1:
            čestica_i = dohvatiČesticu(i)
            čestica_j = dohvatiČesticu(j)
            udaljenost = udaljenost(čestica_i, čestica_j)^2
            udaljenost_x = udaljenost_x(čestica_i, čestica_j)
            udaljenost_y = udaljenost_y(čestica_i, čestica_j)
            akceleracija_i = G * masa(čestica_j) / udaljenost)^2
            ax_i = akceleracija * udaljenost_x / udaljenost
            ay_i = akceleracija * udaljenost_y / udaljenost
            ax_j = - akceleracija * udaljenost_x / udaljenost
            ay_j = - akceleracija * udaljenost_y / udaljenost
```

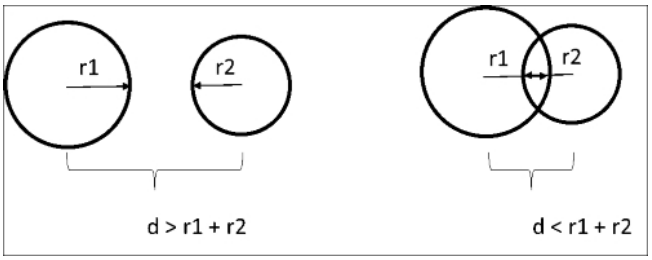
Kod 1.1- Izračun akceleracije zbog djelovanja sile gravitacije

Iz Kod 1.1 se vidi da se za svaku česticu kojih ima N prolazi kroz sve čestice kroz koje se još nije prošlo za prvu česticu tako da se prođe svaki različiti par čestica. Konačno broj primjene formule (1.2) jednak je izrazu (1.4) [3].

$$\frac{N\,(N-1)}{2} \tag{1.4}$$

1.2 Rješavanje sudara

Ova simulacija će simulirati i sudare između čestica jer će se tako postići točniji rezultati simulacije nego da se simulira samo gravitacijski utjecaj. Tako da je dodaćetno potrebno za svaku česticu provjeriti je li je u sudaru s nekom drugom česticom nakon što su izračunate akceleracije pod utjecajem gravitacijske sile. S obzirom ne to da se radi o kružnim česticama određivanje je li je došlo do sudar se radi tako da se provjeri je li zbroj polumjera svake od čestica veći od udaljenosti između njihovih središta. [4]



Slika 1.1 - Provjera sudara između dvije kružnice

Taj princip prikazuje Slika 1.1. Kada se sudar detektira potrebno je pomaknuti čestice po pravcu sudara tako da više nisu u sudaru.

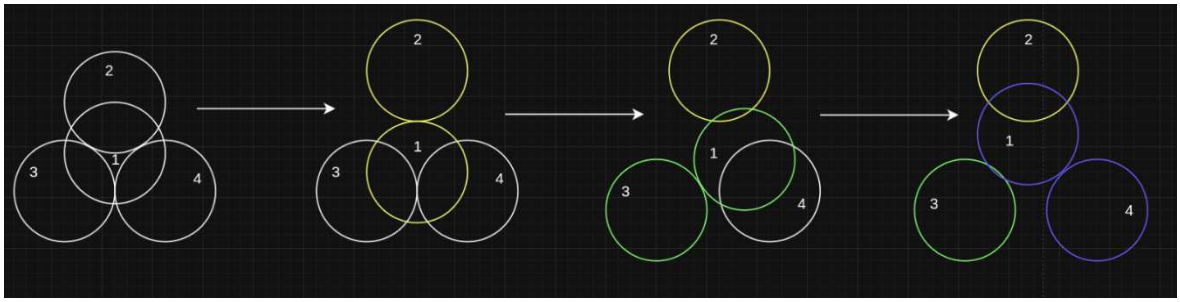
```
za i od 0 od N - 1:
  za j od i + 1 do N - 1:
    čestica_i = dohvatiČesticu(i)
    čestica_j = dohvatiČesticu(j)
    ri = radius(čestica_i)
    rj = radius(čestica_j)
    udaljenost = udaljenost(čestica_i , čestica_j)
      ako je (ri + rj) > udaljenost:
        c = 0.5 * (ri + rj − udaljenost)
        xi += udaljenost_x(i, j) * c / udaljenost
        yi += udaljenost_y(i, j) * c / udaljenost
        xj -= udaljenost_x(i, j) * c / udaljenost
```

```
yj -= udaljenost_y(i, j) * c / udaljenost
```

Kod 1.2 - Izračun pozicija čestica nakon sudara

Algoritam za rješavanje sudara treba napraviti isti broj izračuna kao i algoritam za izračun gravitacijskog utjecaja zato što se radi o istoj za petlji u oba algoritma. Tako da izraz (1.4) i ovdje opisuje broj izračuna ovisno o broju čestica.

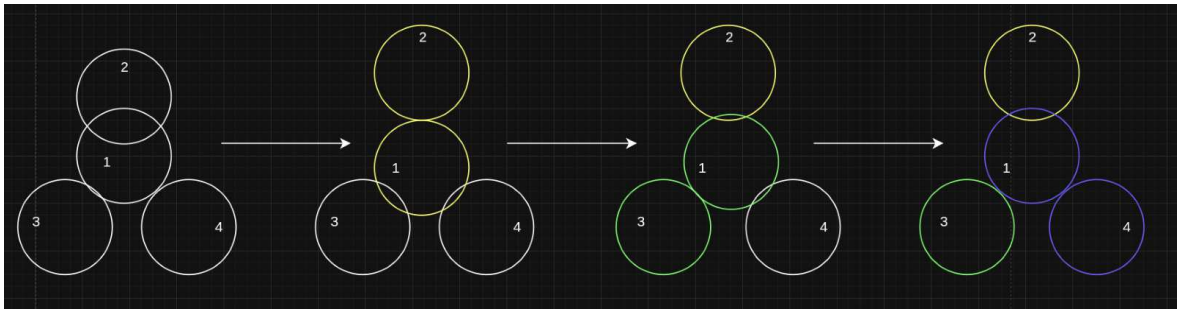
Ovdje je još potrebno istaknuti da u simulaciji s velikim brojem čestica koja nema beskonačnu vremensku preciznost jedna iteracija izračuna sudara uglavnom nije dovoljna da bi sve čestice prestale biti u sudaru. Jer ako je jedna čestica u sudaru s više drugih čestica može se desiti da rješavanje sudara s nekom česticom opet dovede do sudara s drugom česticom s kojom je već sudar prije riješen.



Slika 1.2 - Prva iteracija algoritma za rješavanje sudara

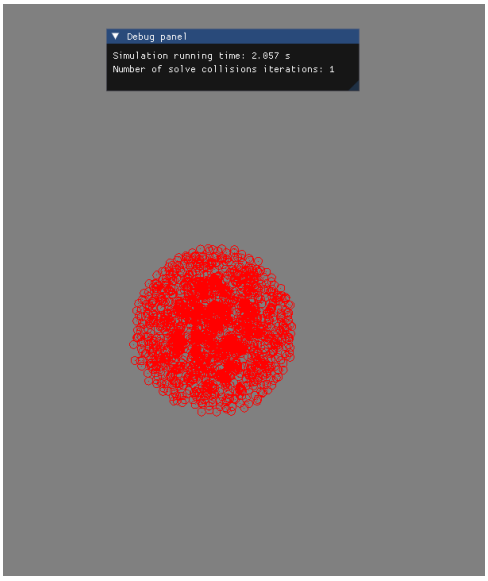
Primjer toga prikazuje Slika 1.2. Vidi se da je algoritam prvo riješio sudar između čestica 1 i 2, pa nakon toga čestica 1 i 3. No to je opet dovelo do sudara između čestica 1 i 2 koje su već provjerene. Na kraju je algoritam riješio sudar između čestica 1 i 4 i time završio izvođenje makar su neke čestice još uvijek u sudaru.

To naravno nije poželjno jer u stvarnoj simulaciji jedna čestica ne može biti na istom mjestu kao i druga. Tako da je ovdje potreban neki način da se te greške smanje. Jedan od načina kako se može smanjiti broj čestica koje će biti u sudaru je povećanjem broja iteracija algoritma.

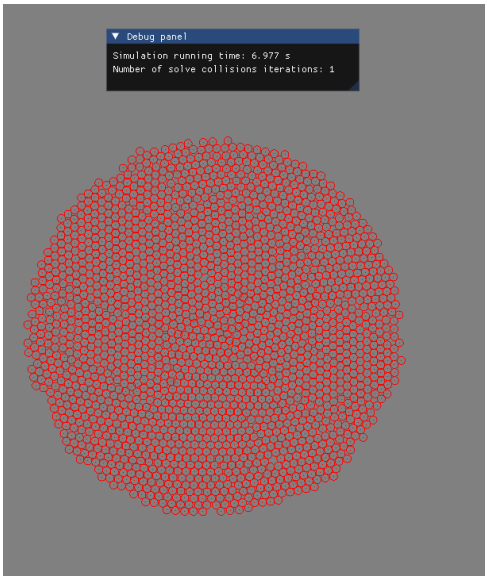


Slika 1.3 - Druga iteracija algoritma za rješavanje sudara

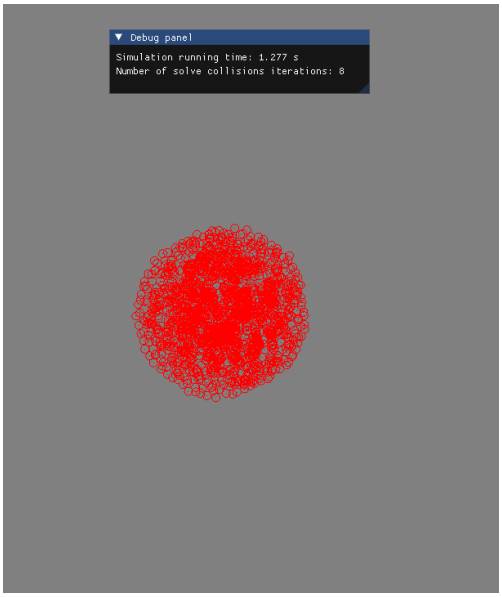
Slika 1.3 prikazuje stanje nakon druge iteracije algoritma i vidljivo je da je greška manja. Još uvijek su čestica 1 i 2 u sudaru na kraju algoritma, no preklapanje je vidljivo manje. Tako da bi se uz još par iteracija to preklapanje uklonilo.



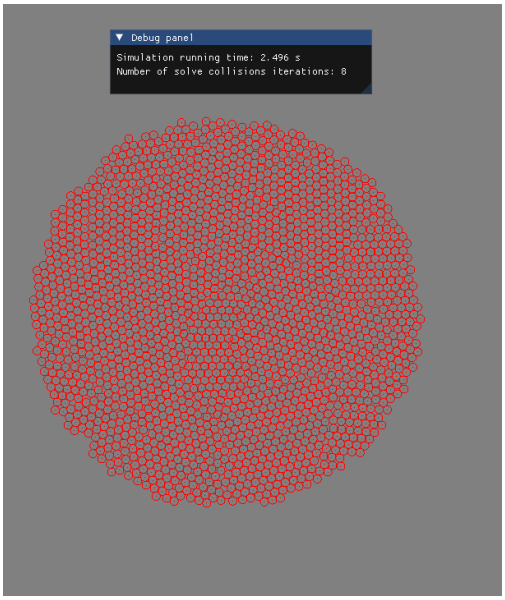
Slika 1.4 – Početno stanje simulacije s jednom iteracijom



Slika 1.5 – Završno stanje simulacije s jednom iteracijom



Slika 1.6 - Početno stanje simulacije s osam iteracija



Slika 1.7 - Početno stanje simulacije s osam iteracija

Slika 1.4 i Slika 1.5 prikazuju slučaj da kada je broj iteracija algoritma za rješavanje sudara postavljen na 1 potrebno je oko 5 sekundi da algoritam konvergira stabilnom stanju gdje se čestice ne preklapaju. No kada se broj iteracija poveća na 8 vrijeme konvergencije padne na oko jednu sekundu, Slika 1.6 i Slika 1.7.
Generalno što su čestice brže i/ili je vremenska preciznost izračuna manja to je potrebno više iteracija algoritma za izračun sudara da bi se dobio isti broj čestica koje su još uvijek u sudaru nakon provođenja algoritma.

1.3 Ažuriranje brzina i pozicija čestica

Nakon rješavanja sudara još je potrebno ažurirati pozicije čestica da bi se dobilo kretanje u simulaciji. U ovoj simulaciji to je postignuto polu implicitnom Eulerovom integracijom, izraz (1.5), jer daje bolje rezultate od implicitne integracije, a vrlo je jednostavna za implementaciju i u računanju na računalu. Formula za izračun brzine i pozicije (1.5) [5]

$$\begin{aligned} v_{n+1} &= v_n + a_{n+1}\Delta t \\ s_{n+1} &= s_n + v_{n+1}\Delta t \end{aligned} \tag{1.5}$$

Razlika između implicitne i polu implicitne Eulerove integracije je da se kod polu implicitne u koraku za izračun pozicije s koristi brzina izračunata sada u trenutačnom koraku. Kod implicitne integracije u tom koraku bi se koristila brzina izračunata u ranijem koraku i time bi se brže stvarala nakupljala greška prilikom izračuna. [5]

```
Za svaku česticu i:
      vxi += dt * axi
      vyi += dt * ayi
      axi += dt * vxi
      ayi += dy * vyi
```

Kod 1.3 - Izračun novih pozicija čestica na temelju akceleracije i brzine

Izračun novih pozicija je potrebno provesti samo jednom za svaku česticu tako da je ukupan broj izračuna jednak broju čestica.

1.4 Vremenska složenost naivnog algoritma

U koraku za izračun gravitacije potrebno je napraviti izračun za svaki par čestica, s time da poredak čestica unutar para nije bitan. Da algoritam provjerava svaku česticu sa svakom

drugom to bi jednostavno bilo N*N izračuna. No s obzirom na to da poredak unutar para broj izračuna se može podijeliti s 2 i onda konačno nije potrebno izračunati utjecaj gravitacije na i-te čestice na tu samu česticu. Tako da je konačan broj izračuna koji je potrebno napraviti za sve čestice N*(N-1) /2.

$$O(Broj\ izračuna\ gravitacije) = O\left(\frac{N\ (N-1)}{2}\right) = O\left(\frac{N^2}{2}\right) = O(N^2) \qquad (1.6)$$

Izraz (1.6) prikazuje izračun vremenske složenosti prema pravilima Velike-O notacije (*eng. Big-O Notation*). Krajnji izraz za vremensku složenost algoritma za izračun gravitacije je $O(N^2)$. [6]

Lako se može vidjeti da algoritma za izračun sudara u jednoj iteraciji treba napraviti isti broj izračuna kao i algoritma za gravitaciju. U slučaju kada je potrebno napraviti više iteracija algoritma za rješavanje sudara vrijeme broj izračuna raste proporcionalno broju iteracija tako da se vremenska složenost množi s konstantnim faktorom. [6]

$$O(Broj\ izračuna\ sudara) = O\left(k\ \left(\frac{N\ (N-1)}{2}\right)\right) = k * O(N^2) = O(N^2) \qquad (1.7)$$

Ako je broj iteracija označen slovom k, izraz (1.7) prikazuje vremensku složenost algoritma za izračun sudara u Veliko-O notaciji. Tako da je složenost naivnog algoritma neovisno o broju iteracija ista kao i za algoritam za izračun gravitacije, a to je $O(N^2)$. [6]

Kod algoritma za izračuna novog položaja čestica je potrebno za svaku česticu samo napraviti konstantan broj operacija. Tako da da je njegova složenost O(N) što je znatno brže od prijašnja dva algoritma. [6]

$$O(Broj\ izračuna\ ažuriranja\ pozicija) = O(N) \qquad (1.8)$$

Ukupna složenost cijelog naivnog algoritma koji slijedno izvodi ta tri koraka je onda, prema pravilima Velike O notacije, jednak najvećoj složenosti od tih troje. [6]

$$O(Broj\ izračuna\ naivnog\ algoritma) = O(N^2) + O(N^2) + O = O(N^2) \qquad (1.9)$$

Iz izraza (1.9) se vidi da naivni algoritam ima kvadratnu vremensku složenost, što znači da svaki puta kada se broj čestica poveća za jedan red veličine broj izračuna raste za dva reda veličine. To ujedno znači da toliko raste i vrijeme izračuna, no to već postaje problem za prosječno moderno sklopovlje za redove veličine veće od 10^4.

Broj čestica	Broj izračuna naivnim algoritmom	Optimistično vrijeme za izračun na modernom računalu	Optimističan broj sličica po sekundi na modernom računalu
1000	1,000,000	0.1 ms	10,000
10,000	100,000,000	10 ms	100
100,000	10,000,000,000	1 s	1
1,000,000	1,000,000,000,000	100 s	0.01

Tablica 1.1- Optimistična aproksimacija vremena izračuna gravitacije na modernom procesoru

Tablica 1.1 prikazuje kako vrijeme izračuna raste ovisno o broju čestica. Ovdje su korištene jako optimistične pretpostavke kao recimo da računalo može procesirati $10 * 10^9$ instrukcija po sekundi (10 GHz) i da je potrebna samo jedna instrukcija po paru čestica. No i s tim pretpostavkama vrijeme potrebno za izračun u stvarnom vremenu postaje preveliko već na 100,000 čestica. Čak i kada se uzme u obzir da moderni procesori mogu imati više dretvi to će optimistično povećati broj izračuna samo za broj dretvi koje procesor ima na raspolaganju, što će biti broj reda veličine 10^2.

Tako da je korištenjem više dretvi moguće ubrzati algoritma za neki konstantan faktor, no time se ne smanjuje vremenska složenost tako da će vrijeme izračuna još uvijek postati presporo za velik broj čestica. Da se smanji vremenska složenost i time dobije znatno sporiji rast vremena izračuna ovisno o broju čestica bit će potrebno pronaći optimalniji algoritam koji neće trebati raditi izračune za svaki par čestica, ali naravno da očuva točnost simulacije. [7]

1.5 Ubrzanje naivnog algoritma paralelizacijom

Naivni algoritma pruža najveću moguću točnost kada se radi o simulaciji gravitacije jer se ne rade nikakve aproksimacije. Tako da je ovaj algoritam idealan u simulacijama koje zahtijevaju najveću preciznost, a ne pokušavaju simulirati jako velik broj čestica. Tako da ubrzanje vremena izvođenja ovog algoritma još uvijek ima smisla u takvim primjenama. Ovaj algoritam je pogodan za paralelizaciju jer se svaka čestica može obrađivati neovisno o drugima.

Izračun gravitacijske sile na svaku čestice se teoretski se može paralelizirati tako da se stvori po jedna dretva za svaki od parova koji je potrebno obraditi i svaka dretva to može raditi neovisno o svim drugim dretvama.

```
Za i od 0 do N-1:
  Za j od i + 1 do N-1:
    stvoriDretvu:
      izračunajGravitacijskuSiluZaDvijeČestice(i, j, ax, ay,
m, r)
```

Kod 1.4 - Paralelizacija algoritma za izračun gravitacijskog utjecaja

Kada se ovako provede algoritam ubrzanje će najviše biti onoliko puta koliko dretvi procesor ima na raspolaganju. Isto tako ovdje poredak izračuna gravitacije nije bitan tako da će rezultati izračuna na jednoj dretvi biti identični onima izračunatima s više dretvi.

Paralelizacija algoritma za izračun sudara može se implementirati na isti način kao i izračun gravitacije.

```
Za i od 0 do N-1:
  Za j od i + 1 do N-1:
    stvoriDretvu:
      riješiSudareIzmeđuDvijeČestice(i, j, x ,y)
```

Kod 1.5 - Paralelizacija algoritma za rješavanje sudara

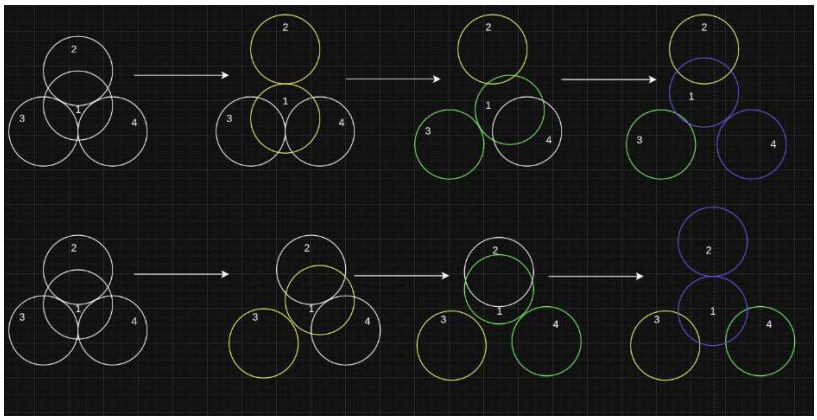
Kod algoritma za rješavanje sudara između dvije čestice postoje operacije pisanja u zajedničku memoriju, to je zato što kada dođe do sudara treba ažurirati pozicije trenutačne čestice i čestice s kojom je ona u sudaru. Tako da je potrebno osigurati da je svaka operacija pisanja u zajedničku memoriju atomarna. Što znači da će se svi koraci operacije izvršiti ili u cijelosti biti vraćeni u stanje prije izvođenja, isto tako je takvim operacijama je osigurano da samo jedna dretva može izvoditi te operacije od početka do kraja.

```
RiješiSudaraIzmeđuDvijeČestice:
   ri = radius(čestica_i)
    rj = radius(čestica_j)
    udaljenost = udaljenost(čestica_i , čestica_j)
      ako je (ri + rj) > udaljenost:
        c = 0.5 * (ri + rj − udaljenost)
       DodajAtomarno(xi, udaljenost_x(i, j) * c / udaljenost)
       DodajAtomarno(yi, udaljenost_y(i, j) * c / udaljenost)
       OduzmiAtomarno(xj, udaljenost_x(i, j) * c / udaljenost)
```

```
OduzmiAtomarno(yj, udaljenost_y(i, j) * c / udaljenost)
```

Kod 1.6 - Rješavanje sudara između dvije čestice sigurno za izvođenje na više dretvi

No iako je atomarnim operacijama riješen problem da više od jedne dretve pristupi istoj memorijskoj lokaciji istom trenutku. U paralelizaciji rješavanja sudara postoji na ovaj način postoji još jedan problem. To je da kako svaka dretva direktno ažurira pozicije čestica koje onda u isto vrijeme koriste druge dretve može doći do razlike rezultata provođenja algoritma na samo jednoj dretvi.



Slika 1.8 - Algoritam rješavanje sudara s različitim redoslijedom parova čestica

Slika 1.8 prikazuje rješavanje sudara istih početnih čestica, no s različitim redoslijedom. Prvo je prikazano stanje čestica kada se prvo rješava sudar između čestica 1 i 2, pa onda 3 i 1 i na kraju 1 i 4. Ispod toga je prikazano stanje ako se prvo rješava sudar između 1 i 3, pa onda 1 i 4 i tek na kraju 1 i 2. Tako da je naravno konačni raspored čestica različit u ta dva pristupa. No niti jedan od ta dva pristupa nije točan jer su u oba čestice još uvijek u sudaru, tako da će s više iteracija algoritam konvergirati prema točnom rješenju koje će biti jako blizu rješenja algoritma koji se nije paralelno izvršavao. Tako da se algoritam više ne odvija deterministički, što znači da ne daje iste rezultate svaki put kada, no s obzirom na to da je točnost ista kao i kod ne paraleliziranog u prosjeku. Isto tako broj potrebnih iteracija za određenu točnost u prosjeku je jednak kao i u ne paralelnom slučaju.

Paralelizacija algoritma za ažuriranje čestica teoretski gledano nije ni potrebna jer ima manju složenost od prethodna dva algoritma, ali je vrlo trivijalna i nema razloga zašto algoritma ne bi bio paraleliziran.

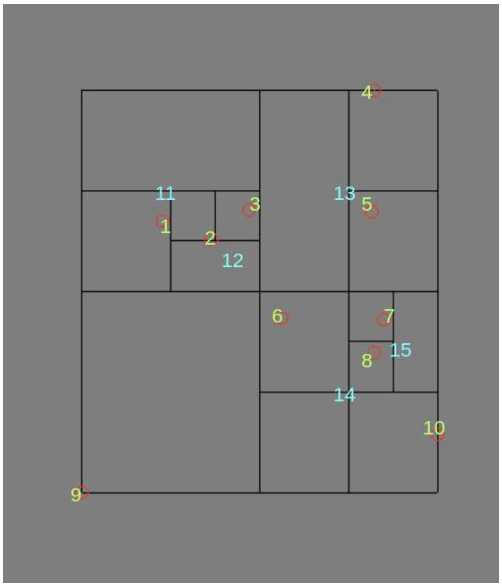
```
Za i od 0 do N-1:
  stvoriDretvu:
  ažurirajPozicijiČestice(i, ax, ay, vx, vy, x, y)
```

Kod 1.7 - Paralelizacija algoritma za ažuriranje čestica

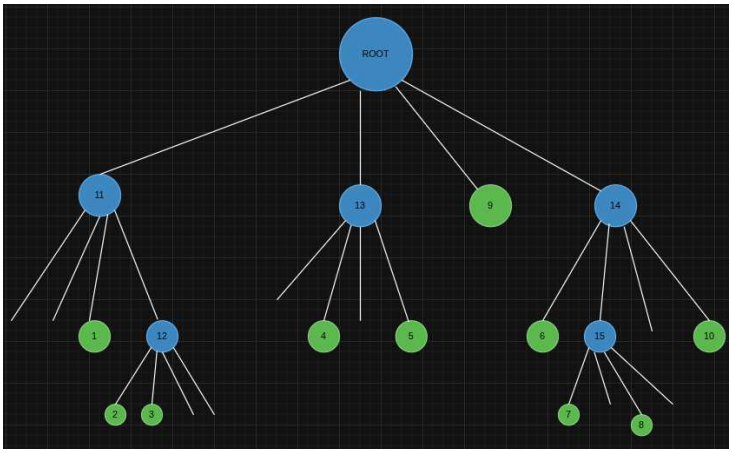
2 Barnes-Hut algoritam

Glavni cilj ovog algoritma kojega su osmislili Josh Barnes i Piet Hut 1986. godine [8] je smanjiti vremensku složenost potrebnu za izračun gravitacijskog utjecaja u sustavima s velikim brojem čestica da se omogući simuliranje do tada nezamislivog broja čestica u astrofizici. Radi se o aproksimativnom algoritmu koji čestice koje su dovoljno udaljene tretira kao jednu veliku česticu. S tim pristupom uspjeli su smanjiti vremensku složenost s $O(N^2)$ na N log(N) što je dramatična razlika za velike redove veličina broja čestica. [7]

Ideja je podijeliti prostor simulacije koristeći strukturu podatka četverostrukog stabla (eng. Quadtree) [9] što znači da svaki čvor ima četiri djece koji mogu biti drugi čvorovi, čestice ili ništa.



Slika 2.1- Podjela prostora četvrtastim stablom iz simulacije



Slika 2.2 - Pripadajući prikaz čvorova stabla za istu podjelu prostora

Slika 2.1 prikazuje kako izgleda podjela prostora za 10 čestica. Čestice su označene s brojevima od 1 do 10, ćelije brojevima od 11 do 15 isto kako bi bilo učinjeno i u samom algoritmu. Slika 2.2 pak prikazuje kako bi izgledali odnosi između ćelija u stablastom prikazu i gdje svaka čestica i ćelija spada u stablo.

Pošto Barnes-Hut algoritam računa međusobni utjecaj gravitacije za svaku ćeliju će biti potrebno spremiti ukupnu masu i središte mase. Onda će se kasnije pri izračunu sila koristi princip da ako je neki čvor dovoljno daleko od čestice da se sve čestice unutar tog čvora mogu aproksimirati centrom mase tog čvora. Udaljenost nakon koje algoritam počinje aproksimirati se može podešavati prije provođenja algoritma i o njoj će ovisiti brzina izvođenja, ali isto tako i točnost simulacije.

Barnes-Hut algoritam se sastoji od dva glavna koraka. Prvo je potrebno izgraditi četverostruko stablo, pri tome pamtiti i računati ukupnu masu i centar mase za svaki čvor. Nakon toga je potrebno za svaku česticu s pomoću izrađenog stabla izračunati gravitacijski utjecaj. Postoje još dodatni koraci koji mogu ubrzati izvođenje algoritma, no osnovni algoritam ima samo ta dva koraka. [10]

2.1 Izgradnja četverostrukog stabla

Izgradnja stabla se može podijeliti u dva glavna djela, koji se na isti način izvode za svaku česticu. U prvom djelu je potrebno pronaći trenutačnu ćeliju list, to je ćelija koja nema svoju djecu, u kojoj se prema poziciji nalazi čestica koju obrađujemo. Onda nakon što se pronađe takva ćelija je potrebno umetnuti česticu na to mjesto. Bitan detalj je da svaka čestica mora biti u svojoj zasebnoj ćeliji.

Ćelija list može se pronaći tako da se cijelo vrijeme prate pozicije svih četiri strana trenutačne ćelije. Koja je u prvom koraku korijenska ćelija. Onda se iz pozicije čestice odredi u koji od četiri kvadranta ćelije čestica treba ići. Nakon što se odredi kvadrant potrebno je provjeriti nalazi li se na tom mjestu ćelija lista. Ako ne onda se ponavljaju isti koraci, ako da onda se prestaje s pronalaženjem lista. Prilikom traženja lista potrebno je svakoj ćeliji kroz koju se prođe pribrojiti masu te čestice koja se obrađuje i ažurirati centar mase. [10]

```
PronađiĆelijuList(NovaČesticia, KorjeskaĆelija):
  TrenutnaĆelija = KorjenskaĆelija
  Lijevo, Desno, Gore, Dolje = dohvatiRuboveKorjenskeĆelije()
  Beskonačno izvodi:
    TrenutnaĆelija.ažurirajCentarMase(NovaČesticia)
```

```
    TrenutniKvadrant = izračunajKvadrant(NovaČesticia, Lijevo,
                                         Desno, Gore, Dolje)
    Lijevo, Desno, Gore, Dolje = izračunajNoveRubove(Lijevo, Desno, Gore,
                                                     Dolje, TrenutniKvadrant)
    TrenutnaĆelija = TrenutnaĆelija.dohvatiDijete(TrenutniKvadrant)
    Ako TrenutnaĆelija.jeList():
      vrati TrenutnaĆelija, TrenutniKvadrant
```

Kod 2.1- Pronalazak ćelije lista za novu česticu

Kada je ćelija list pronađena postoje dva slučaja. Lakši slučaj je da je ta ćelija prazna jer tada se čestica može direktno umetnuti u tu ćeliju i možemo završiti algoritam za tu česticu. No kada se radi o slučaju da u toj ćeliji već postoji druga čestica potrebno je osigurati da svaka od te dvije čestice dobije svoju ćeliju. To se radi tako da na mjesto čestica koja je prije bila u toj ćeliji napravi nova ćelija koja će biti roditelj staroj i novoj čestici koje od kojih će svaka dobiti svoj kvadrant. No naravno ovdje se opet može desiti slučaj da obje čestice trebaju ići u isti kvadrant, tada je potrebno ponoviti ovaj postupak sve obje trebaju ići u isti kvadrant. Ovdje je potrebno za svaku novu ćeliju koja se napraviti izračunati novi centar mase. [10]

```
UmetniNovuČesticuUStablo(NovaČestica, ĆelijaList, KvadrantList, Lijevo,
Desno, Gore, Dolje):
  ČesticaList = ĆelijaList.dohvatiDijete(KvadrantList)
  Ako nije ČesticaList:
    ĆelijaList.dodajDijete(NovaČestica, KvadrantList)
  Else:
    Beskonačno izvodi:
      NovaĆelija = Ćelija()
      Ako NovaĆelija.izvanGranica():
        Greška(NovaĆelijaIzvanGranica())
    NovaĆelija.ažurirajCentarMase(ČesticaList)
    NovaĆelija.ažurirajCentarMase(NovaČestica)
    ĆelijaList.dodajDijete(NovaĆelija, KvadrantList)
    KvadrantList = izračunajKvadrant(ĆelijaList, NovaĆelija)
    KvadrantNovaČestica = izračunajKvadrant(NovaČestica, Lijevo, Desno,
                                            Gore, Dolje)
    Lijevo, Desno, Gore, Dolje = izračunajNoveRubove(Lijevo, Desno, Gore,
                                                     Dolje, KvadrantList)
    If KvadrantList != KvadrantNovaČestica:
      ĆelijaStaraČestica = Ćelija(ČesticaList)
      NovaĆelija.dodajDijete(ĆelijaStaraČestica, KvadrantList)
      ĆelijaNovaČestica = Ćelija(NovaČestica)
```

```
      NovaĆelija.dodajDijete(ĆelijaNovaČestica, KvadrantNovaČestica)
      Vrati
    ĆelijaList = NovaĆelija
```

Kod 2.2 - Umetanje nove čestice u stablo

Jedan od problema do kojega može doći kod izgradnje stabla je ako su čestice na istim pozicijama. Tada bi se stvorilo previše novih ćelija jer bi čestice uvijek trebale ići u isti kvadrant, pa je zato potrebno dodati provjeru je li nova ćelija prešla unaprijed postavljene granice o broju ćelija da bi se izbjegla beskonačna petlja.

Kada se promatra jedna čestica prvi dio algoritma za izgradnju stabla ima vremensku složenost O(log(n)) zato što je potrebno samo jednom doći do listova u četvrtastom stablu. Drugi dio algoritma je konstantan jer ne ovisi o broju čestica ili veličina stabla. On ovisi o tome koliko je čestica koju je potrebno umetnuti blizu postojećoj čestici, no broj iteracija raste logaritamski s obzirom na njihovu blizinu jer se svaki puta granice ćelije smanje za pola. Tako da će se uvijek u relativno malo koraka doći do rješenja. Tako da je ukupna vremenska složenost umetanja jedne čestice jednaka O(log(N)).

$$O(broj\ izračuna\ za\ umetanje\ jedne\ čestice\ u\ stablo) = O(\log(N)) \qquad (2.1)$$

```
IzgradiČetverostrukoStablo():
  Za i od 0 do N-1:
    novaČestica = dohvatiČesticu(i)
    ĆelijaList, KvadrantList = PronađiĆelijuList(novaČestica)
    UmetniNovuČesticuUStablo (novaČestica, ĆelijaList, KvadrantList)
```

Kod 2.3 - Izgradnju četverostrukog stabla

$$O(broj\ izračuna\ za\ izgradnju\ cijelog\ stabla) = O(\mathrm{N}\log(N)) \qquad (2.2)$$

Naravno da bi se izgradilo cijelo stablo potrebno je provesti ovaj algoritam za sve čestice u simulaciji. Što znači da je ukupna složenost algoritma za izgradnju cijelog stabla prikazana izrazom (2.2).

2.2 Izračun gravitacijskog utjecaja

Izračun gravitacijskog utjecaja se isto radi pojedinačno za svaku česticu. Sastoji se od tri provjere koji se ponavljaju svaki puta s drugom ćelijom, a počinje se s djecom korijenske ćelije. Prvi korak je provjeriti radili li se o ćeliji listu koji ne sadrži trenutačnu česticu. Ako

se radi o takvom listu onda je samo potrebno izračunati gravitacijsku silu s tom česticom i nastavlja se dalje sa sljedećom ćelijom. No ako se ne radi o listu nego o ćeliji koja sadrži drugu djecu potrebno je izračunati omjer veličine ćelije (S) i udaljenosti čestice od centra mase te ćelije (D) i usporediti ga s unaprijed zadanim faktorom greške (θ). [10]

Ako je S / D < θ to znači da je ćelija sa svom svojom djecom dovoljno daleko od te čestice i da je dovoljno samo izračunati utjecaj centra mase te ćelije na česticu i nastaviti dalje sa sljedećom ćelijom na redu. [10]

No ako je S / D > θ to pak znači da je ćelija preblizu čestici da bi je mogli tretirati kao jedno tijelo. Tada je potrebno rekurzivno odraditi svu djecu te ćelije istim ovim algoritmom. Algoritam je gotov kada su obrađena sva djeca korijenske ćelije, a time ujedno i sve ostale ćelije koje su bile ispod praga greške. [10]

```
IzračunajGravitacijskiUtjecajRekurzivno(ĆelijaČestice,TrenutnaĆelija,
FaktorGreške):
  Za ĆelijaDijete u TrenutnaĆelija.dijeca():
    Ako ĆelijaDijete.jeList():
      IzračunajUtjecajGravitacije(ĆelijaČestice, ĆelijaDijete)
    inače:
      D = izračunajUdaljenostĆelija(ĆelijaČestice, ĆelijaDijete)
      S = izračunajVeličinuĆelije(ĆelijaDijete)
      ako S / D < FaktorGreške:
        IzračunajUtjecajGravitacije(ĆelijaDijete, ĆelijaDijete)
      inače:
        za ĆelijaUnuk u ĆelijaDijete.dijeca():
          IzračunajGravitacijskiUtjecajRekurzivno (ĆelijaČestice,
                                        ĆelijaUnuk, FaktorGreške)
```

Kod 2.4 - Izračun gravitacijske sile za jednu česticu

Vremenska složenost algoritma za izračun sila na jednu česticu je isto O(log(N)), no brzina izvođenja ovisi o parametru greške. Jer što je taj parametar veći to će algoritam obrađivati manje ćelija, ali samim time povećati grešku jer radi veće aproksimacije. Ako je parametar greške postavljen na vrijednost 0 to bi značilo da će algoritam uvijek provjeriti sve ćelije, što uključuje sve čestice i u tom slučaju bi složenost bila O(N). No s obzirom na to da se za parametar uvijek uzima pozitivna vrijednost složenost će biti logaritamska s obzirom na N jer veličina ćelije S eksponencijalno pada što je ćelija niže u stablu. To znači da će algoritam u većini slučajeva eliminirati većinu ćelija koje su duboko u stablu, a nisu preblizu toj čestici.

```
IzračunajGravitacijskiUtjecajZaSveČestice(KorijenskaČestica):
```

```
Za i od 0 do N-1:
  Čestica = dohvatiČesticu(i)
  IzračunajGravitacijskiUtjecajRekurzivno(Čestica, KorijenskaČestica,
                                          FaktorGreške)
```

Kod 2.5 - Izračun gravitacijske sile za sve čestice

$$O(broj\ izračuna\ za\ izračun\ gravitacijske\ sile) = O(\mathrm{Nlog}\,(N)) \tag{2.3}$$

S obzirom na to da je ovaj algoritam potrebno provesti za svaku česticu kojih ima N ukupna složenost algoritma je jednaka izrazu (2.3). Tako da je vremenska složenost jednaka izgradnji stabla i međusobno se neće usporavati. Iz toga slijedi da je ukupna vremenska složenost Barnes-Hut algoritma isto jednaka O(N*log(N)).

$$O(Barnes - Hut\ algoritma) = O(\mathrm{Nlog}\,(N)) + O(\mathrm{Nlog}\,(N)) = O(\mathrm{Nlog}\,(N)) \tag{2.4}$$

3 Rješavanje sudara četverostrukim stablom

Barnes-Hut algoritam u sebi nema uključen izračun sudara jer kada se simuliraju sustavi poput galaksija gdje su udaljenosti i mase simuliranih čestica užasno velike sudari su poprilično rijetki, a i kad se dese ne igraju veliku ulogu u točnosti simulacije. No u ovoj simulaciji je odlučeno simulirati i sudare za dodatnu preciznost, a i jer se za izračun sudara može iskoristiti četvrtasto stabla koje je ionako već izgrađeno kod implementacije Barnes-Hut algoritma.

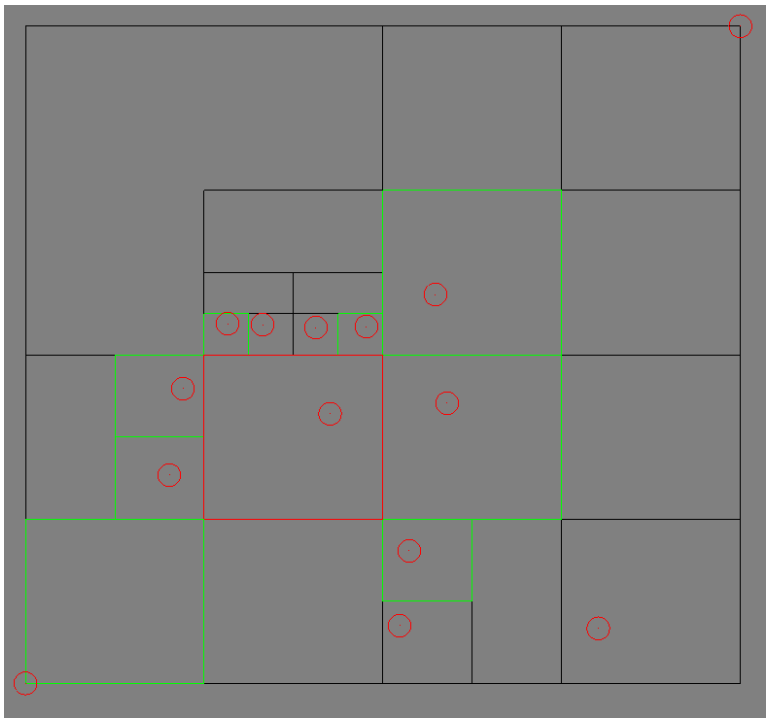
Kao što je već zaključeno iz izraza (1.9) izračun sudara naivnim algoritmom ima kvadratnu složenost što bi previše usporavalo Barnes-Hut algoritam koji ima loglinearnnu složenost prema izrazu (2.4). Tako da je potrebno smanjiti složenost algoritma za izračun sudara na barem toliko jer ako simulacija će biti onoliko brža koliko je najsporiji algoritam. No točno to je moguće postići iskorištavanjem četvrtastog stabla koje se već gradi u Barnes-Hut algoritmu. Glavna stvar kako optimizirati izračun sudara je na pametan način izabrati koje čestice je potrebno provjeriti jesu li u sudaru, a koje se mogu ignorirati jer su predaleko od promatrane čestice.

Najlakši način kako to izvesti je ako je prostor uniformno podijeljen na mrežu jer tada dovoljno provjeriti ćeliju u kojoj se nalazi čestica i čestice iz susjednih ćelija koje je trivijalno za pronaći. No isti taj princip se može primijeniti kada je prostor podijeljen koristeći četvrtasto stablo. Samo je proces pronalaska susjednih ćelija ipak malo zahtjevniji.

3.1 Osnovni algoritam pronalaska susjednih ćelija pomoću četvrtastog stabla

Ono što čini ovaj algoritam zahtjevnim od pronalaska susjednih ćelija kod uniforme mreže je da svaka ćelija dijeli roditelja s maksimalno tri druga lista. Što znači da će uvijek biti susjednih ćelija koje nisu sadržane u istom roditelju kao i promatrana čestica i da će zato trebati prolaziti kroz stablo da ih se pronađe.

Glavna ideja pronalaska susjednih ćelija je da one moraju dijeliti kut s promatranom ćelijom, a dijeliti znači da se točka koja predstavlja kut promatrane ćelije nalazi na jednoj ili obje strane od susjedne ćelije. No taj pristup radi samo za ćelije na istoj višoj ili istoj razini kao promatrana ćelija.



Slika 3.1 - Prikaz susjednih ćelija sa zajedničkim kutevima

Slika 3.1 prikazuje pretragu ćelija s dijeljenim kutevima, ovdje se traže susjedi za crvenu ćeliju, vidi se da ta ćelija dijeli kutove s osam drugih ćelija, koje su obojene zeleno. No postoje još dvije ćelije koje su susjedi, ali s njima ne dijeli rub. Do takve situacije može doći samo kada su susjedne ćelije na nižoj dubini od promatrane. Tako da je za takve ćelije potrebno dodati svu njihovu djecu u susjedne ćelije.

Za svaki od kutova potrebno je pronaći roditelja od promatrane ćelije koja u sebi sadrži taj kut, ali da kut nije na rubu roditelja. To je potrebno ponavljati sve dok se na nađe takav roditelj i pri tome pratiti broj pređeni roditelja.

```
PronađiRoditeljaGdjeKutNijeNaRubu(ćelija, kut):
  TrenutnaĆelija = ćelija
  BrojProđenihRoditelja = 0
  Dok je KutNaRubuĆelije(TrenutnaĆelija, kut):
    BrojProđenihRoditelja++
    TrenutnaĆelija = TrenutnaĆelija.roditelj()
    Vrati TrenutnaĆeljia, BrojProđenihRoditelja
```

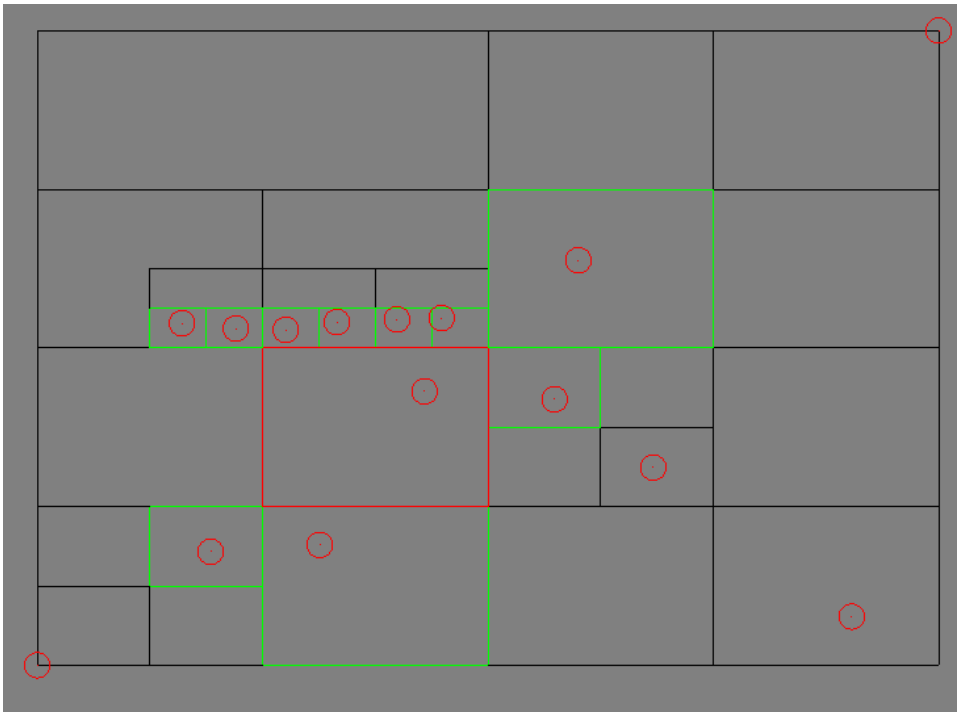
Kod 3.1 - Pronalazak roditelja kojemu zadani kut nije na rubu

```
NađiĆelijeKojimaJeKutNaRubu(Ćelija, kut, dubina, susjedneĆelije):
  TrenutnaĆelija = ćelija
  Ako je ćelija.jeList():
    susjedneĆelije.dodajBezDuplikata(ćelija)
```

```
  vrati
Za dijete u ćelija.djeca():
  Ako je Dubina > 0:
    Ako KutNaRubuĆelije(ćelija, kut):
      NađiĆelijeKojimaJeKutNaRubu(ćelija, kut, dubina-1,
    susjedneĆelije)
  Inače:
    DodajSvuDjecuRekurzivno(ćeljia)
```

Kod 3.2 - Dodavanje susjednih ćelija kojima je kut na rubu ćelije

Ovakav algoritam prikazan na Kod 3.2 nije savršen jer će u određenim slučajevima dodati čestice koje nisu susjedne, ali je zato vrlo jednostavan i brz za izvođenje pa par čestica previše ne stvara problem.



Slika 3.2 – Rezultat algoritma za pronalazak susjednih ćelija

Slika 3.2 prikazuje kako se sada nađeni svi susjedi crvene ćelije uključujući ćelije koje ne dijele kutove nego samo stranice. No vidljivo je da su u gornjem desnom kutu pronađene dvije čestice iako je samo jedna susjedna.

3.2 Dodatak algoritmu generacije četvrtastog stabla za pomoć pri izračunu susjeda

Da bi ovaj algoritam mogao ovako provesti za svaku ćeliju je potrebno, uz roditelje i djecu znati sva četiri kuta. To je najlakše prikazati tako da svaka ćelija još dodatno ima vrijednosti za gornji, donji, lijevi i desni rub iz kojih se direktno mogu dobiti kutovi. Tako da je potrebno napraviti izmjenu u algoritmu izgradnje četverostrukog stabla u djelu kada se umeće nova čestica i kada se stvaraju novu ćelije.

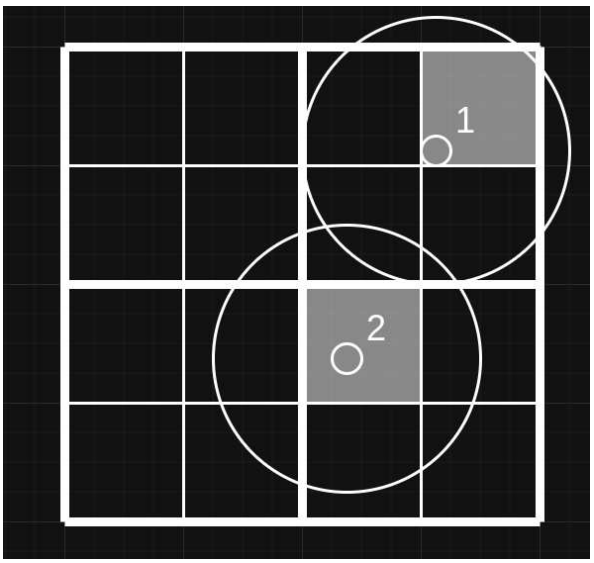
```
UmetniNovuČesticuUStablo(NovaČestica, ĆelijaList, KvadrantList,
                         Lijevo, Desno, Gore, Dolje):
  ČesticaList = ĆelijaList.dohvatiDijete(KvadrantList)
  ako nije ČesticaList:
    ĆelijaList.dodajDijete(NovaČestica, KvadrantList)
    ĆelijaList.postaviGranice(Lijevo, Desno, Gore, Dolje)
  inače:
    Beskonačno izvodi:
      NovaĆelija = Ćelija()
      Ako NovaĆelija.izvanGranica():
        Greška(NovaĆelijaIzvanGranica())
      NovaĆelija.postaviGranice(Lijevo, Desno, Gore, Dolje)
      NovaĆelija.ažurirajCentarMase(ČesticaList)
      NovaĆelija.ažurirajCentarMase(NovaČestica)
      ĆelijaList.dodajDijete(NovaĆelija, KvadrantList)
      KvadrantList = izračunajKvadrant(ĆelijaList, NovaĆelija)
      KvadrantNovaČestica = izračunajKvadrant(NovaČestica, Lijevo, Desno, Gore,
                                              Dolje)
      Lijevo, Desno, Gore, Dolje = izračunajNoveRubove(Lijevo, Desno, Gore,
                                                       Dolje, KvadrantList)
      If KvadrantList!= KvadrantNovaČestica:
        ĆelijaStaraČestica = Ćelija(ČesticaList)
        ĆelijaStaraČestica.postaviGranice(Lijevo,Desno, Gore, Dolje)
        NovaĆelija.dodajDijete(ĆelijaStaraČestica, KvadrantList)
        ĆelijaNovaČestica = Ćelija(NovaČestica)
        ĆelijaNovaČestica.postaviGranice(Lijevo,Desno,Gore, Dolje)
        NovaĆelija.dodajDijete(ĆelijaNovaČestica, KvadrantNovaČestica)
        Vrati
     ĆelijaList = NovaĆelija
```

Kod 3.3 - Umetanje nove čestice u stablo i postavljanje granica ćelija

Podebljane linije su dodane originalnom algoritmu za umetanje česticu u stablo, Kod 2.2. Sada svaka ćelija direktno zna svoje granice koje se onda mogu koristiti u algoritmu za pronalazak susjednih ćelija bez dodane vremenske cijene.

3.3 Problem prevelikih čestica u algoritmu pronalaženja susjeda

No još jedan problem do kojega može doći prilikom korištenja ovoga algoritma je da iako algoritam pronađe točne susjedne ćelije to ne treba značiti da su to sve ćelije s kojima je potrebno provjeriti sudar. Do te situacije dođe kada je polumjer čestice veći od visine ili širine ćelije.

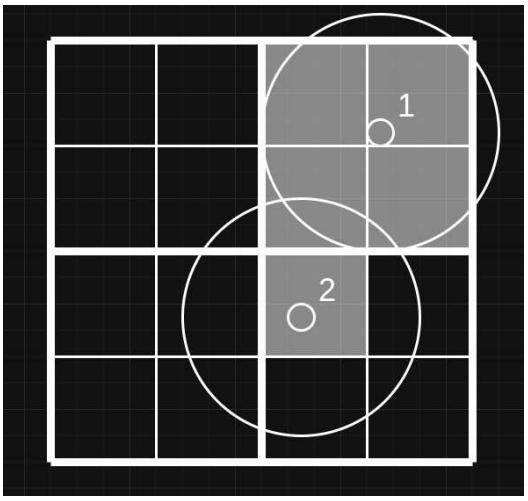


Slika 3.3 - Primjer čestica u sudaru koje nisu u susjednim ćelijama

Slika 3.3 prikazuje jednu takvu situaciju, s time da su prikazane samo dvije čestice radi preglednosti. Sive ćelije su one u kojima su sadržani centri čestica, no sa slike se vidi da te ćelije nisu susjedne i ovaj ih algoritam ne bi pronašao. Kada se radi o česticama jednakih veličina takva situacija nije očita jer se uopće ne bi trebalo izgraditi stablo koje to dozvoljava. U primjeru sa slike vidi se da u susjedne ćelije ne bi stale nove čestice iste veličine za izgradnju takvog stabla.

No u stvarnoj simulaciji, gdje rješavanja sudara nije savršeno jer čestice imaju određenu brzinu i mogu se naći puno bliže jedna drugoj nego što bi to bilo moguće u stvarnosti takvi slučajevi su česti. Isto tako ako postoji velika razlika između veličina čestica može doći do takve situacije.

Rješenje tog problema je da pronađemo roditelja od ćelije koju promatramo čija su i širina i duljina veći od polumjera čestice.



Slika 3.4 - Pronalaženje roditelja koji je veći od polumjera čestice

Ako se umjesto ćelije u kojoj je čestica jedan uzme njezin prvi roditelj tada ćelija u kojoj je čestica 2 postaje susjedna i sudari će se kasnije moći točno riješiti.

```
PronađiRoditeljaVećegOdPolumjeraČestice(čestica, ćelija):
  Ako je čestica.polumjer() > ćelija.dužina() ili
                          čestica.polumjer() > ćelija.širina():
    roditelj = PronađiRoditeljaVećegOdPolumjeraČestice(čestica,
                                              ćelija.roditelj())
    Vrati roditelj
  Vrati ćeljia
```

Kod 3.4 - Pronalazak roditelja koji je veći od promjera čestice

Tako da se algoritam za pronalazak susjednih ćelija za jednu česticu sastoji od tri glavna djela. Ako je čestica prevelika za svoju ćeliju onda je potrebno pronaći prvog roditelja koji je dovoljno velik. Nakon toga za tu ćeliju ili ćeliju roditelja za svaki od kutova pronaći ćeliju roditelja kojoj taj kut nije na rubu i na kraju za svaku od tih ćelija roditelja rekurzivno pronaći svu djecu koja sadrže taj kut i u rezultat dodati sve različite listove.

```
PronađiSusjedneČestice(ćelijaČestice):
  SusjedneĆelije = praznaLista()
  Lijevo, Desno, Gore, Dolje =  ćelijaČestice.dohvatiGranice()
  Za Kut u kutevi(Lijevo, Desno, Gore, Dolje):
    Roditelj, Dubina =
            PronađiRoditeljaGdjeKutNijeNaRubu(ćelijaČestice, kut)
    NađiĆelijeKojimaJeKutNaRubu(ćelijaČestice, kut, dubina,
                                susjedneĆelije)
  Vrati susjedneĆelije
```

Kod 3.5 - Cijeli algoritam za pronalazak susjednih ćelija za jednu česticu

3.4 Vremenska složenost algoritma za pronalazak susjeda

Korak za pronalazak roditelja kojemu promatrani kut nije na rubu je u najgorem slučaju potrebno napraviti log(N) puta jer čestica može imati samo toliko roditelja, tako da se uvijek staje na korijenskoj ćeliji.

Ukupan broj koraka koji je potrebno napraviti za korak dodavanja ćelija iz tog roditelja je isto log(N) do dubina na kojoj je ćelija za koju se traže susjedi. No nakon toga broj izračuna može varirati, pa se i približiti O(N) u najgorem slučaju. To je slučaj kad je jedna ćelija puno veća od drugih što će uglavnom značiti da je na višoj razini, dok su ostale čestice na puno manjoj relativno uz nju. Tako da će tada trebati dodati velik broj tih manjih čestica. No u slučaja kada je stablo relativno balansirano ukupna složenost će biti log(N) jer su onda u prosjeku dubine isto uravnotežene.

Većinu puta pronalazak roditelja na višoj razini jer čestica ima veći polumjer od veličine njezine ćelije nije potrebna, nego se radi relativno rijetko kada se simulacije nađe u čudnom stanju. Pa čak i kada se radi jedan ili dva izlaska iz roditelja su dovoljna, a u najgorem slučaju radi se o log(N) izračuna.

$$O(ukupan\ broj\ izračuna\ za\ pronalazak\ susjednih\ čestica) = O(\mathrm{Nlog}(N)) \quad (3.1)$$

Tako da je ukupna vremenska složenost algoritma za pronalazak susjeda prikazana izrazom (3.1)

3.5 Rješavanje sudara

Algoritmom pronalaska susjednih čestica rješavanje sudara postaje trivijalno za implementirati i značajno brže za izvesti na računalu. Jer sada više nije potrebno za svaku česticu proći svaku drugu nego se jednostavno za česticu provjere samo njezini susjedi.

```
RiješiSudare():
  Za i od 0 do N-1:
    Čestica = dohvatiČesticu(i)
    susjedneČestice = PronađiSusjedneČestice(Čestica)
    Za SusjednaČestica u susjedneČestice:
      riješiSudarIzmeđuDvijeČestice(Čestica, SusjednaČestica)
```

Kod 3.6 Algoritam za rješavanje sudara s poznatim susjednim ćelijama

4 Paralelizacija Barnes-Hut algoritma

Iako Barnes-Hut algoritam donosi veliku prednost u vremenskoj složenosti nad naivnim algoritmom i time omogućuje simulaciju redova veličine većeg broja čestica još uvijek može biti brži. Sada više ne izmjenjivanjem algoritma i smanjenjem vremenske složenosti nego paralelizacijom tako da se izvođenja algoritma ubrza za neki konstantan faktor što još uvijek donosi vrijednost u simulaciji.

Ideja je da svaka dretva na procesoru mogu što nezavisnije odrađivati posao za svoju česticu. No opet je potrebna sinkronizacija i komunikacija između dretvi u nekim većini slučajevima. Tako da je glavni zadatak povećati nezavisnost i smanjiti vrijeme potrebno na sinkronizaciju.

4.1 Paralelizacija izračuna gravitacijskog utjecaja

Algoritam za izračun gravitacijskog utjecaja, Kod 2.5, je jednostavno za paralelizirati zato jer se ovdje za svaku česticu ažuriraju samo vrijednosti akceleracije samo za tu česticu. Sve ostale operacije koje rade nad zajedničkom memorijom su operacije čitanja. Tako da ovdje nema potrebe sinkronizirati dretve, nego je dovoljno samo pustiti svaku dretvu da za svoju česticu izvodi algoritam prikazan na Kod 2.4.

```
IzračunajGravitacijskiUtjecajZaSveČestice(KorijenskaČestica):
  Za i od 0 do N-1:
    StvoriDretvu():
      Čestica = dohvatiČesticu(i)
      IzračunajGravitacijskiUtjecajRekurzivno(Čestica,
                        KorijenskaČestica, FaktorGreške)
```

Kod 4.1 - Paralelizacija algoritma za izračuna gravitacijskog utjecaja

4.2 Paralelizacije rješavanja sudara i ažuriranja položaja čestica

Paralelizacija algoritma za rješavanje sudara se svodi na paralelizaciju pronalaska susjednih čestica i paralelizaciju samog rješavanja sudara. Algoritam pronalaska susjednih čestica već radi istu stvar za svaku česticu i ne radi nikakve operacije pisanja u zajedničku memoriju. Tako da je isto jednostavan za paralizairati. Sam algoritam rješavanja sudara između dvije čestice već je osiguran da radi za više dretvi, Kod 1.6.

Tako da je samo potrebno stvoriti dretvu za svaku ćeliju i provjeriti sudare sa svim susjednim česticama.

```
RiješiSudare():
  Za i od 0 do N-1:
    StvoriDretvu():
    Čestica = dohvatiČesticu(i)
    SusjedneČestice = PronađiSusjedneČestice(Čestica)
    Za SusjednaČestica u susjedneČestice:
      riješiSudarIzmeđuDvijeČestice(Čestica, SusjednaČestica)
```

Kod 4.2 - Paralelizacija algoritma za rješavanje sudara s poznatim susjednim ćelijama

Ažuriranje pozicija čestica se nije mijenjalo provođenjem Barnes-Hut algoritma. Tako da se još uvijek koristi algoritma prikazan na Kod 1.3.

4.3 Paralelizacije izgradnje stabla

Korak izgradnje stabla je najteži korak Bares-Hut algoritma za paralelizirati zbog toga jer je cijelo umetanje čestice u stablo kritični odsječak. Kritični odsječak je dio paralelnog programa u kojem više dretvi od jednom pristupaju dijeljenim memorijskim lokacijama. U takvim slučajevima potrebno je osigurati da samo jedna dretva može pristupati tim memorijskim lokacijama u istom trenutku da bi se izbjeglo nedefinirano ponašanje. [11]

Algoritam izgradnje stabla se sastoji od dva glavna koraka, a to su pronalazak lista i umetanje čestice na njegovu poziciju. Oba djela su u stvari kritični odsječci zato jer dretve pristupaju zajedničkoj memoriji, a u ovom slučaju to je četvrtasto stablo. Prvi dio algoritma je samo čitanje iz četvrtastog stabla tako da je uredu ako to sve dretve rade u isto vrijeme. No problem postaje ako jedna dretva provodi prvi korak i čita iz stabla, a druga dretva umeće česticu na istom mjesto u stablu u istom trenutku. Isto tako do problema dolazi ako dvije dretve u isto vrijeme pokušavaju umetnuti svoje čestice na isto mjesto.

Jedan od načina kako se može riješiti ovaj problem je uvesti mogućnost zaključavanja svake od ćelija u stablu koristeći ključanicu *(eng. mutex lock*) [12]. Tako da kada neka dretva pronađe ćeliju list prvo treba provjeriti je li je zaključana i ako nije treba ju zaključati za sebe. Nakon toga ta dretva može bez smetanja drugih dretvi umetnuti svoju česticu u stablo, a sve druge dretve koje isto žele pristupiti tom listu trebaju čekati dok prva dretva ne otključa tu ćeliju.

```
PronađiĆelijuList(NovaČesticia, KorijeskaĆelija):
  TrenutnaĆelija= KorjenskaĆelija
  Lijevo, Desno, Gore, Dolje = dohvatiRuboveKorjenskeĆelije()
  Beskonačno izvodi:
    TrenutnaĆelija.ažurirajCentarMase(NovaČesticia)
    TrenutniKvadrant = izračunajKvadrant(NovaČesticia, Lijevo,
                                         Desno, Gore, Dolje)
      Lijevo, Desno, Gore, Dolje = izračunajNoveRubove(Lijevo,
                           Desno, Gore, Dolje, TrenutniKvadrant)
      TrenutnaĆelija = TrenutnaĆelija.dohvatiDijete(
                                                 TrenutniKvadrant)
      Ako TrenutnaĆelija.jeList():
        TrenutnaĆelija.zaključaj()
        Ako TrenutnaĆelija.jeList():
          vrati TrenutnaĆelija, TrenutniKvadrant
        Inače:
          TrenutnaĆelija.otključaj()
```

Kod 4.3 - Pronalazak ćelije list siguran za izvođenje na više dretvi

U ranijem pokazanom kodu za pronalazak ćelije lista kod izgradnje stabla, Kod 2.1, dodano je par linija koje omogućuju izvođenje tog algoritma na više dretvi. Sve ostaje isto do trenutka kada dretva pronađe ćeliju koja je list. No samo zato što je ta ćelija bila list u jednom trenutku ne treba značiti da će biti i u sljedećem jer je uvijek moguće da neka druge dretva već obrađuje taj list.

Zbog toga je potrebno prvo zaključati tu ćeliju list, a zaključati ćeliju ili generalnije neku memorijsku lokaciju znači prvo provjeriti je li je ta memorijska lokacije već zaključana i ako je čekati dok ne bude otključana. Tako da ako dretva zaključi da je list koji ona želi obrađivati već zaključan ona će čekati sve dok druga dretva na otključa tu ćeliju.

No ako je ćelija bila zaključana, to znači da ju je druge dretva mijenjala u međuvremenu. S toga je potrebno provjeriti jeli ta ćelija još uvijek list. Ako je ćelija stvarno još uvijek list onda dretva može biti sigurna da ju nitko drugi nije mijenjao i da ona može nastaviti dalje. No ako dretva zaključi da ćelija više nije list to znači da treba nastaviti dalje s pronalaskom lista i tada može otključati tu ćeliju da joj druge dretve mogu pristupiti.

Sada dalje dretva koja je uspjela zaključati i pronaći list može nesmetano umetnuti novu ćeliju. Potrebno je paziti na to da ta dretva otključa tu ćeliju koju je i zaključala i da osigura

da ta ćelija ostane list za vrijeme dok radi umetanje. Jer da dretva odmah doda djecu toj ćeliji listu koju je zaključala ona više ne bi bila list i druge dretve ne bi znale da je zaključana.

```
UmetniNovuČesticuUStablo(NovaČestica, ĆelijaList, KvadrantList, Lijevo,
                         Desno, Gore, Dolje):
  ČesticaList = ĆelijaList.dohvatiDijete(KvadrantList)
  ako nije ČesticaList:
    ĆelijaList.dodajDijete(NovaČestica, KvadrantList)
    ĆelijaList.otključaj()
  inače:
    ĆelijaListOriginalna = ĆelijaList
    ĆelijaDijeteListu = Ništa
    KvardrantĆelijeDijeteListu = KvadrantList
    Beskonačno izvodi:
      NovaĆelija = Ćelija()
      Ako NovaĆelija.izvanGranica():
        Greška(NovaĆelijaIzvanGranica())
      NovaĆelija.ažurirajCentarMase(ČesticaList)
      NovaĆelija.ažurirajCentarMase(NovaČestica)
      Ako je prvi korak petlje:
        ĆelijaDijeteListu = NovaĆelija
      Inače:
        ĆelijaList.dodajDijete(NovaĆelija, KvadrantList)
        KvadrantList = izračunajKvadrant(ĆelijaList, NovaĆelija)
        KvadrantNovaČestica = izračunajKvadrant(NovaČestica, Lijevo,
                                                Desno, Gore, Dolje)
       Lijevo, Desno, Gore, Dolje = izračunajNoveRubove(Lijevo,
                                Desno, Gore, Dolje, KvadrantList)
       If KvadrantList != KvadrantNovaČestica:
         ĆelijaStaraČestica = Ćelija(ČesticaList)
         NovaĆelija.dodajDijete(ĆelijaStaraČestica, KvadrantList)
         ĆelijaNovaČestica = Ćelija(NovaČestica)
         NovaĆelija.dodajDijete(ĆelijaNovaČestica,
                                KvadrantNovaČestica)
         ĆelijaListOriginalna.dodajDijete(ĆelijaDijeteListu,
                                          KvardrantĆelijeDijeteListu)
         ĆelijaListOriginalna.otključaj()
         Vrati
       ĆelijaList = NovaĆelija
```

Kod 4.4 - Umetanje čestice u stablo sigurno za izvođenje na više dretvi

Izmjene u algoritmu dodavanja čestice u stablo, Kod 2.2, su to da je potrebno otključati pronađenu ćeliju list i u slučaju kada se čestice direktno umeću, i u slučaju kada je potrebno napraviti nove ćelije. No još je dodatno potrebno dodati dio pamćenja originalne ćelije list i kvadranta u koju treba dodati nove ćelije da bi se to dodavanje moglo napraviti na kraju da se osigura da će sve druge dretve stati na toj razini i neće se probati spustiti niže od te zaključane ćelije.

```
IzgradiČetverostrukoStablo():
  Za i od 0 do N-1:
  StvoriDretvu():
    novaČestica = dohvatiČesticu(i)
    ĆelijaList, KvadrantList = PronađiĆelijuList(novaČestica)
    UmetniNovuČesticuUStablo(novaČestica, ĆelijaList, KvadrantList)
```

Kod 4.5 - Paralelizacija izgradnje četverostrukog stabla

U ovakvom pristupu vrijeme čekanja dretvi da se ćelija list otključa će biti obrnuto proporcionalno dubini stabla u trenutnom koraku izvođenja, naravno pod pretpostavkom da je stablo relativno dobro balansirano. To je zato jer na početku dok stablo nema veliku dubinu većina dretvi će htjeti umetnuti čestice u iste ćelije jer ćelija nema puno tako da će većina trebati čekati. No čim se to stablo produbi stvara se eksponencijalno više opcija ćelija u koje čestice mogu ići, pa se s time smanjuje i vrijeme čekanja pojedine dretve.

5 Implementacija Barnes-Hut algoritma

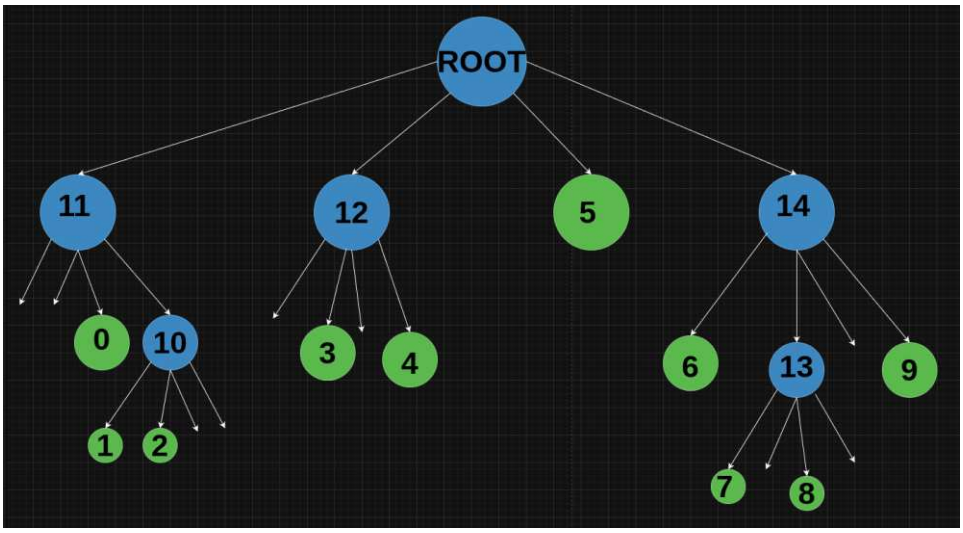
Naglasak ovog rada je na optimizaciji i performancama, tako da je logičan izbor bio programski jezik C++ koji nudi veliku fleksibilnost što se tiče optimizacija i ubrzanja izvođenja. Dodatno je izabrana CUDA tehnologija za rad s grafičkom karticom. S obzirom da je od početka bilo poznato da će biti potrebno implementirati algoritme za simulacije gravitacije i na procesoru i na grafičkoj kartici većina koda pisanog u C++ jeziku za procesor je jako slična CUDA implementaciji koja koristi sintaksu sličnu programskom jeziku C. [14]

5.1 Korištene strukture podataka

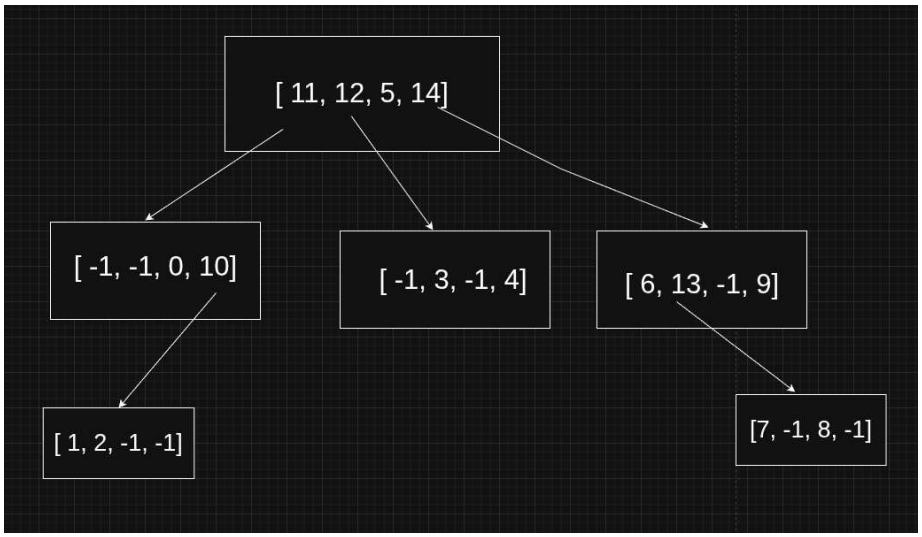
S obzirom da je cilj imati kod koji što više sliči programskom jeziku C, ovdje nisu korištene napredne strukture podataka kao klase ili unaprijed definirana stabla ili vektori. Nego su svi podatci sadržani u jednodimenzionalnim poljima.

5.2 Četvrtasto stablo

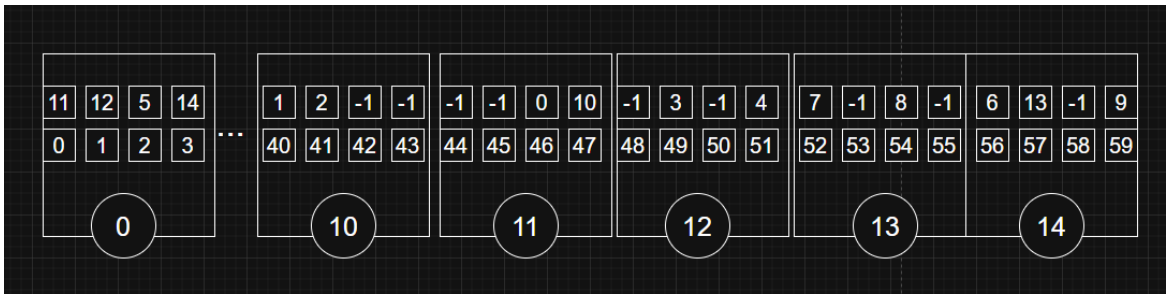
Glavnu strukturu podataka koja se koristi u Barnes-Hut algoritmu je četvrtasto stablo. Tako da je potrebno omogućiti takvu strukturu podatka običnim poljima. Stablo se inače na najlakši način implementira tako da je čvor svoje klasa koja sadrži reference na svoju djecu i još neke dodatne podatke o čvoru. No s obzirom da se ovdje samo koriste polja bit će potrebno za reference na druge čvorove imati jedno polje i onda za svaku od dodatnih podataka po jedno polje.



Slika 5.1 - Prikaz logičke raspodjele čestica u stablu



Slika 5.2 - Prikaz raspodjele čestica pomoći polja



Slika 5.3 – Prikaz raspodjele čestica u memoriji

Slika 5.1 prikazuje primjer kako može izgledati raspored čvorova u četvrtastom stablu. Zeleni čvorovi su listovi, plavu su čvorovi roditelji. Vidi se da su numerirani tako da listovi idu od 0 do 9, a roditelji od 10 do 14. No u memoriji će to izgledati drugačije jer se neće koristiti pokazivači nego jedno veliko neprekinuto polje.

Za to polje je potrebno prvo definirati maksimalan broj punih čvorova, s obzirom da svaki čvor može imati četvero djece polje će ukupno bit četiri puta veće od maksimalnog broja čvorova. Pri inicijalizaciji je potrebno sve elemente postaviti na vrijednost –1 što predstavlja da je čvor prazan.

Slika 5.3 prikazuje kako bi takvo isto stablo izgledalo u polju u memoriji procesora. U gornjem redu su označene vrijednosti polja, a ispod toga indeksi u polju. Prva četiri elementa tog polja će biti djeca korijenskog čvora. U ovom primjeru to bi bilo 11, 12, 5, 14. Ovdje se može primjeriti da svaki element koji je manji od ukupnog broja čestica predstavlja čvor list u kojem je čestica, a svaki veći ili jednak broju čestica predstavlja čvor roditelj koja ima barem jedno dijete. Navigacija po stablu je vrlo jednostavna zato jer su djeca čvora roditelja

spremljena index tog roditelja pomnožen s četiri. Znači djeca od nekog čvora kao recimo 12 se nalaze na indexu 48, 49, 50 i 51.

Time je riješen problem djece u stablu. No za algoritma detekcije sudara još je za svaki čvor potrebno znati i njegovog roditelja. To nije moguće na efikasan način dobiti iz ovoga polja, nego je potrebno prilikom izgradnje stabla prilikom umetanja svakog čvora u novo polje spremiti roditelja. Tako da svaki puta kada je potrebno pronaći roditelja od nekog čvora to se može napraviti direktno u konstantnom vremenu.

```
int *children = new int[maxNumberOfNodes];
int *parents = new int[maxNumberOfNodes];
```

Kod 5.1 - Isječak koda za stvaranje polja za spremanje roditelja i djece

5.3 Pozicije, brzine, akceleracije i mase

Za svaki čvor potrebno je pratiti njegovu ukupnu masu i njegov centar mase, a za čvorove listove dodatno i brzine i akceleracije. Tako da postoje dva polja veličine jednake maksimalnom broju čvorova pomoću kojih se prate x i y pozicije centra mase svakog čvora. Kada se radi o listovima koji predstavljaju čestice centar mase čvora je naravno centar čestice, a kada se radi o čvorovima roditeljima centar mase je aritmetička sredina svih centara mase njegove djece. Za masu postoji jedno polje iste veličine u kojoj na poziciji čvora listova pišu mase čestica, a na pozicijama čvorova roditelja pišu zbrojene mase sve djece.

Kada se radi o brzini i akceleraciji nije potrebno spremati vrijednosti za čvorove koji nisu listovi. No ako stablo nije potpuno, što znači da na svakoj dubini svi roditelji imaju uvijek svu moguću djecu, broj listova nije moguće unaprijed izračunati. Tako da će biti zadan i maksimalan broj čestica i u simulaciji će se paziti da se taj broj ne prijeđe. Svako polje će biti te veličine. Brzina i akceleracija obje imaju x i y komponentu tako da je ovdje potrebno napraviti još četiri dodatna polja.

```
float *x= new float [maxNumberOfNodes];
float *y = new float [maxNumberOfNodes];
float *mass = new float [maxNumberOfNodes];
float *vx= new float [maxNumberOfParticles];
float *vy = new float [maxNumberOfParticles];

float *ax= new float [maxNumberOfParticles];
```

```
float *ay = new float [maxNumberOfParticles];
```

Kod 5.2 – Isječak koda za stvaranje polja za spremanje pozicija centra mase, mase, brzina i akceleracija

5.4 Pozicije kutova ćelija

Pozicije kutova ćelija je potrebno moći u konstantnom vremenu dohvatiti u algoritam izračuna susjednih čestica. Svaki čvor odgovara točno jednoj ćeliji u raspodjeli prostora, a svaka ćelija ima četiri kuta. Tako da za spremiti sve kutove bilo bi potrebno osam pozicija, no kako je svaki kut presjek dvije stranice ćelije dovoljno je samo spremiti najveću i najmanju x koordinatu ćelije i isto tako za y koordinatu. Pa kada je potrebno dohvatit kut samo treba dohvatiti točnu x i točnu y poziciju koja predstavlja traženi kut. Znači da je za spremanje svih kutova ćelija dovoljno polje isto veliko kao i za spremanje djece.

```
float *edges= new float[4 * maxNumberOfNodes];
```

Kod 5.3 – Isječak koda za stvaranje polja za spremanje rubova ćelije

5.5 Susjedne ćelije

Svaka čestica u teoriji može imati beskonačno mnogo drugih čestica. Tako da nije moguće unaprijed odrediti veličinu polja u koje bi se mogle spremiti susjedne čestice. Za takvu primjenu idealno bi bilo koristiti proširujuće polje, koje se proširuje kako se u njega dodaju nove čestice. No s obzirom da je potrebno koristiti što primitivnije strukture podataka, zato da bi kasnije lakše bilo migrirati isti kod u CUDA jezik, proširujuće polje ovdje na pomaže. Tako da je odlučeno da će postojati unaprijed zadan maksimalan broj koliko svaka čestica može imati susjeda.

```
float *adjacentParticles= new
float[maxNumberOfAdjacentParticles * maxNumberOfParticles];
```

Kod 5.4 – Isječak koda za spremanje susjednih ćelija

Razlog zašto je uopće potrebno spremati susjedne ćelije, jer se onda to računanje može obaviti jednom nakon izgradnje četvrtastog stabla i onda iskoristiti u više iteracija rješavanja sudara. Jer generalno većina čestica će ostati unutar ili jako blizu svoje prvobitne ćelije nakon više iteracija rješavanja sudara.

5.6 Pomoćni koraci Barnes-Hut algoritma

Implementacija na procesoru je vrlo slična gore opisanim koracima algoritma jedino što ima još par pomoćnih koraka kao što su postavljanje polja na početnu vrijednost, računanje minimalnog okvira u koji stanu sve čestice i dovršetak izračuna centra mase.

```
resetArrays(x, y, ax, ay, mass, child, index, left, right, bottom,
          top, edgePositions, parent, adjacentCells, numParticles,
           numNodes, maxNumberOfGameObjects, threadCount);
computeBoundingBox(x, y, left, right, bottom, top, numParticles,
                   threadCount);
buildQuadTree(x, y, mass, child, index, left, right, bottom, top,
       edgePositions, parent, locks, numParticles, numNodes,
       threadCount);
computeCentreOfMass(x, y, mass, index, numParticles, threadCount);
calculateAdjacentCells(child, numParticles, parent, top, bottom,
              right, left, edgePositions, radius,
              adjacentCells,closeThreshold, threadCount);
for (int i =0 ; i < solveCollisionsIterations; i++) {
  solveCollisions(x, y, vx, vy, radius, mass, adjacentCells,
                 numParticles, restitution, threadCount);
 }
calculateForces(x,y, ax, ay, mass, child, numParticles, gravity, 0.5,
                left, right, threadCount);
integrateParticles(x, y, vx, vy, ax, ay,
                   numParticles, dt, threadCount);
```

Kod 5.5 - Isječak svih koraka Barnes-Hut algoritma iz simulacije

Funkcija `resetArrrays` samo vraća sva polja za koja je bitno da budu vraćena na početnu vrijednost prilikom nove iteracije cijeloga algoritma. Neke od stvari koje je potrebno postaviti na početne vrijednosti su polja za praćenje djece i roditelja koje treba postaviti na –1, postaviti vrijednosti s kojima se prati obuzimajući okvir na stare vrijednosti i izbrisati vrijednosti o susjednim ćelijama

Nakon toga funkcija `computeBoundingBox` računa obuzimajući okvir koji predstavlja najmanje i najveće x i y koordinate čestica. To je potrebno da bi si algoritam za izgradnju stabla mogao izračunati pozicije ćelija na temelju pozicija čestica.

Konačno funkcija `computeCentreOfMass` dodatno podijeli poziciju izračunati u svakom od čvorova u stablu s ukupnom masom tog čvora da bi se dobili pravi centri masa.

6 Implementacija paralelizacije

6.1 Implementacija na procesoru

Za implementaciju paralelizacije u ovoj simulaciji korišteno je programsko sučelje *OpenMP* [15] koje pruža vrlo jednostavnu implementaciju osnovnih potreba paralelizacije. Potrebno je paralelizirati svaki od koraka simulacije fizike.

6.1.1 Obrada svih čestica

Svaka od funkcija obrađuje sve čestice ili sve ćelije jednu po jednu u za petlji. Tako da je potrebno omogućiti da svaka dretva preuzme jedan korak za petlje.

```
# pragma omp parallel for num_threads(threadCount)
for (int bodyIndex =0; bodyIndex < n; bodyIndex++) {
      // ... kod petlje
 }
```

Kod 6.1 – Primjer paralelizacije *za petlje* koristeći *OpenMP*

U ovom primjeru prikazanim Kod 6.1 vidi se korištenje *OpenMP* sučelja da bi se postiglo baš to. Isto tako je prikazano kako se kontrolira broj dretvi koje će izvoditi zadanu *za petlju* pred definiranom funkcijom `num_threads`.

6.1.2 Atomarne operacije

Ako koraci za petlje ne pristupaju istom djelu memorije onda je tom jednom linijom paralelizacija gotova. No u slučaju da koraci petlje pristupaju istoj memoriji to će značiti da je moguće da će dvije dretve probati pristupiti tom djelu memorije i tada će doći do ne predvidljivog ponašanja. Primjer toga je slučaj kada jedan korak petlje kod provjere sudara treba ažurirati pozicije za dvije čestice u sudaru. Tada je još potrebno osigurati da se takve operacije odvijaju atomarno [13].

```
# pragma omp atomic
x[atom_1_idx] += - col_vec_x * (mass1 / (mass1 + mass2));
# pragma omp atomic
x[atom_1_idx] += - col_vec_x * (mass1 / (mass1 + mass2));

# pragma omp atomic
x[atom_2_idx] += col_vec_x * (mass2 / (mass1 + mass2));
# pragma omp atomic
y[atom_2_idx] += col_vec_y * (mass2 / (mass1 + mass2));
```

Kod 6.2 – Primjer ostvarenja atomarnih operacija koristeći *OpenMP*

U ovom slučaju ako dvije dretve u isto vrijeme dođu do ove operacije i probaju ažurirati poziciju na istom indeksu jedna od dretvi će prva izvesti tu operaciju, a za to vrijeme druga dretva će čekati dok operacija nije u potpunosti gotova i tek tada će moći izvršiti svoju operaciju. Druge po redu dretva će vidjeti izmjene koje je prva napravila tako da neće doći do ne konzistentnosti.

6.1.3 Kritični odsječak za specifičnu lokaciju

U algoritmu izgradnje stabla postoji dio za koji je potrebno osigurati da umetanje čestice na određeni list smije raditi samo jedna dretva u jednom trenutku. Tako da je cijeli dio umetanja čestice u stvari kritični odsječak.

OpenMP nudi opciju da se dio koda deklarira kao kritični odsječak. To se radi s oznakom `#pragma omp critical`. No problem kod tog pristupa je da dretve cijeli taj dio koda smatraju kao kritični odsječak, to bi značilo da bi sve dretve čekale dretvu koja trenutno umeće česticu bilo gdje u stablo. To bi naravno bilo jako ne učinkovito jer je to glavni dio algoritma za izgradnju stabla. Tako da je potreban neki drugi pristup koji omogućuje zaključavanje samo određenog dijela memorije. *OpenMP* pruža koncept ključanice (*eng. lock*) koja radi na principu *mutex* zaključavanja. To znači da ako je jedna dretva zaključala ključanicu, sve ostale koje ju isto žele zaključati trebaju čekati da ju ta prva dretva otključa.

Da bi se omogućilo zaključavanje svake od ćelija potrebno je napraviti polje veličine maksimalnog broja svih ćelija.

```
std::vector locks locks = std::vector(numNodes);
for (int i = 0; i < numNodes; ++i) {
  omp_init_lock(&locks[i]);
}
```

Kod 6.3 – Inicijalizacija *OpenMP* ključanica

Tako da kada je neka dretva pronašla list u koji želi umetnuti svoju česticu potrebno je prvo zaključati taj list i tek kada to uspije može nastaviti s umetanjem. Naravno jako je bitno da svaki put kada dretva zaključa list točno ga jednom i otključa da se izbjegne zastoj (*eng. deadlock*).

```
omp_set_lock(&locks[leaf]);
// .. obavi umetanje
omp_unset_lock(&locks[leaf]);
```

Kod 6.4 – Primjer zaključavanja i otključavanja lista koristeći *OpenMP* ključanice

6.2 Implementacija na grafičkoj kartici

Implementacija algoritma na grafičkoj kartici ostvarena je koristeći *CUDA* tehnologiju. Radi se o platformi za paralelno računarstvo koja omogućuje ubrzanje računalno zahtjevnih izračuna iskorištavajući snagu i broj procesora grafičke kartice koju je napravila kompanija *NVIDIA*. *CUDA* podrazumijeva sve od knjižnica (*eng. libraries*) i programskog sučelja do upravljačkih programa za samu grafičku karticu. Time NVIDIA osigurava maksimalno iskorištavanje resursa jer proizvode i sklopovlje, što su *NVIDIA* grafičke kartice i programsku podršku u obliku *CUDA*-e. [16]

Glavna ideja CUDA programskog sučelja je izvoditi jednostavan kod na što više grafičkih procesora u isto vrijeme. Što je pogodno za operacija u računalnoj grafici kao što su operacija nad točkama (eng. pixels) ili u području umjetne inteligencije gdje je potrebno raditi izračune nad velikim matricama. [16]

6.2.1 Dijeljena memorija

Memorija na grafičkoj kartici je fizički odijeljena od memorije koju ima procesor. Glavni razlog tome je da grafička kartica bude što bliže svojoj memoriji što smanjuje vrijeme potrebno za dohvat i spremanje podataka. Isto tako takva memorija u uglavnom brže od privremene memorije koju koristi procesor.

Ako devete grafičke kartice trebaju pristup memoriji radi čuvaju podataka, kao što trebaju u ovoj simulaciji, onda je potrebno prvo inicijalizirati sve memoriju koju će grafička kartica koristiti. To se radi naredbom `cudaMalloc` koja prima pokazivač koji postoji u memoriji od procesora i veličinom memorije koju treba inicijalizirati u bajtovima. Prilikom te naredbe u memoriji grafičke kartice se oslobađa toliko bajtova i vrijednost pokazivača postaje memorijska lokacija te memorije na grafičkoj kartici.

```
cudaMalloc((void**)&x, numNodes*sizeof(float));
cudaMalloc((void**)&y, numNodes*sizeof(float));
```

Kod 6.5 - Primjer inicijalizacije memoriju za polja koja čuvaju centar mase

No nakon što je tako inicijaliziran pokazivač njemu se više ne može pristupiti direktno, nego je za to potrebno koristiti naredbu `cudaMemcpy` koja može kopirati memoriju i s procesora na grafičku karticu i obrnuto.

```
// kopiraj memoriju s procesora na grafičku karticu
cudaMemcpy(x_cpu, x_gpu, numNodes*sizeof(float), cudaMemcpyHostToDevice);
// kopiraj memoriju s grafičke kartice na procesor
```

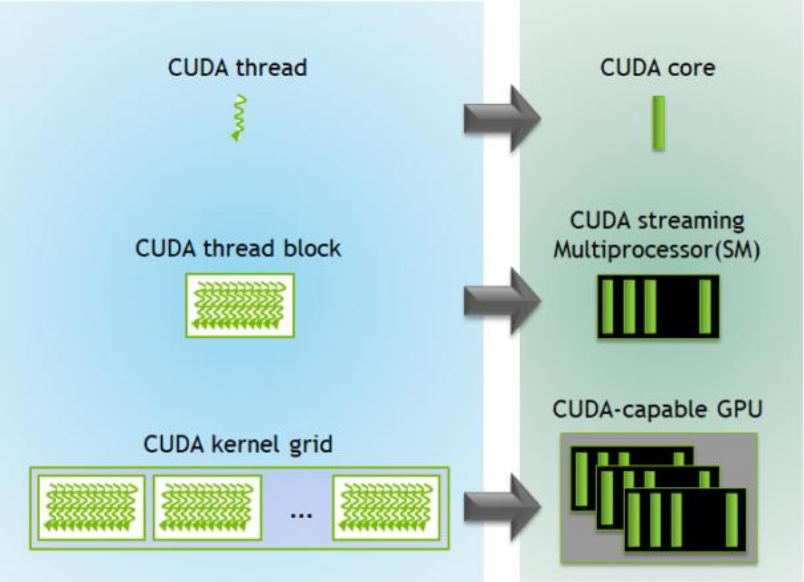
```
cudaMemcpy(x_gpu, x_cpu, numNodes*sizeof(float), cudaMemcpyDeviceToHost);
```

Kod 6.6 – Primjer kopiranja memorije između procesora i grafičke kartice

Ove naredbe je potrebno koristiti svaki puta kada je potrebno prenijeti podatke o česticama s procesora na grafičku karticu ili kada je potrebno dohvatiti rezultante pozicije čestica zbog prikaza.

6.2.2 Obrada svih čestica

S obzirom da se sva paralelizacija u ovoj simulaciji radi tako da se paraleliziraju za petlje, potrebno je to napraviti i u na grafičkoj kartici. Pisanje programskoga koda koji se može izvoditi na grafičkoj kartici se sastoji od dva dijela. Prvi dio je napisati jezgru koja će se grafička kartica moći izvesti, a drugi je pozvati tu jezgru pomoći sučelja koji nudi *CUDA* i pri tome specificirati konfiguraciju za izvođenje.



Slika 6.1 – Arhitektura CUDA tehnologije

CUDA je logički podijeljena na dretve, grupu dretvi, blokove i mrežu. Što prikazuje Slika 6.1. Svaka grupa dretvi (*eng. warp*) sadrži 32 dretve koje u isto vrijeme izvršavaju iste operacije. Blok onda sadrži više grupa dretvi, a maksimalno 32 grupe dretve što je 1024 dretve. Mreža sadrži više blokova, a služi da se olakša rad s dvodimenzionalnim i trodimenzionalnim poljima ili matricama. [16]

```
if (n == 0){
  return;
}
int blockSize = 512
```

```
int gridSize = (n+ blockSize -1) / blockSize
integrateParticlesKernel<<<gridSize, blockSize >>>(x, y, vx, vy,
                                                   ax, ay, n, dt);
```

Kod 6.7 - Primjer konfiguracije i pozivanja jezgre za ažuriranje brzina i pozicija čestica

Ovdje se vidi da je prilikom pozivanja jezgre potrebno navesti dva argumenta, a to su veličina bloka i veličina mreže. Veličina bloka predstavlja broj dretvi koje će biti sadržane u jednom bloku, a veličina mreže je broj blokova. Tako da se veličina bloka postavlja na konstantu vrijednost, a veličina mreže se računa na temelju veličine bloka i broja dretvi koje će biti potrebno. U ovom slučaju potrebno je n dretvi jer će svaka dretva biti zadužena za ažuriranje jedne čestice. Ako je veličina bloka postavljena na 1024 to znači da je potreban jedan blok sve dok je broj čestica manji ili jednak od 1024, no kada broj čestica pređe 1024 potrebna su dva bloka.

$$veličinaMreže = \frac{(N + veličinaBloka - 1)}{veličinaBloka} \tag{6.1}$$

Tako da se veličina mreže može dobiti izrazom (6.1) s kojom je garantirano da će uvijek biti dostupno barem N dretvi, no isto tako garantira da nikada neće biti iskorišteno više od 2N – 1 dretvi. Jer bitno je koristiti sto manji broj dretvi da bi se smanjilo opterećenja na grafičku karticu. Slučaj kada će najveći broj dretvi biti neiskorišten će biti kada je potrebno točno jedna dretva više od veličine bloka.

Bitno je da veličina bloka bude djeljiva s 32 zato jer jedan fizički multi-procesor (*eng. streaming multiprocesor, SM)* u sebi ima 32 dretve koje će se ionako sve izvoditi odjednom. Tako da ako je veličina bloka 33 to sigurno znači da će se aktivirati dva multi-procesora i da će u jednom 31 dretva biti ne iskorištena.

Što je veća veličina bloka to će se smanjiti broj zakazivanja tih blokova. Jer recimo da je veličina bloka jednaka 32, a potrebno je pokrenuti 1024 dretve. To će rezultirati zakazivanjem 32 bloka. No da je veličina bloka bila postavljena na 1024, bilo bi potrebno zakazati samo jedan blok. No u većini slučajeva su to vrlo male razlike koje neće utjecati na performance, tako da je preporučena veličina bloka za moderne grafičke kartice 256 ili 512. [17]

```
__global void integrateParticlesKernel(float *x, float *y, float *vx,
        float *vy, const float *ax, const float *ay, int n, float dt)
```

```
{
  int bodyIndex = threadIdx.x + blockIdx.x*blockDim.x;

   if (bodyIndex < n) {
     vx[bodyIndex] += dt * ax[bodyIndex];
     vy[bodyIndex] += dt * ay[bodyIndex];

     x[bodyIndex] += dt * vx[bodyIndex];
     y[bodyIndex] += dt * vy[bodyIndex];
   }
}
```

Kod 6.8 – Primjer *CUDA* jezgre za ažuriranje brzina i pozicija čestica

Ovdje se može vidjeti primjer kako se piše *CUDA* jezgra. Svaka od dretava koje su podijeljene u blokove će izvršavati ovaj isti kod, jedina razlika će biti u tome da će njihove lokalne varijable `threadIdx` i `blockIdx` biti različite. Prema tim varijablama se može upravljati time koja dretva će što izvoditi. Paralelizacija jednodimenzionalne za petlje je najjednostavniji primjer toga. Identifikator dretve za jednu dimenziju računa se tako da se pomnoži dretvin blok s ukupnom brojem blokova i na to doda dretvin položaj u bloku. U ovom primjeru koriste se blokovi veličine 512. Recimo da je prilikom pozivanja ove jezgre potrebno obraditi 1000 čestica. To će značiti da su dretve podijeljene u dva bloka. Tako da ih ukupno ima 1024 što je malo više nego čestica.

Identifikatori će onda ići od 0 do 1024 tako da je potrebno prvo provjeriti je li identifikator dretve koja trenutno izvodi jezgru manja od broja čestica. Nakon toga sve se može napraviti isto kao i običnoj za petlji.

6.2.3 Atomarne operacije

Operacija za koju je potrebno imati atomarnu verziju u ovoj simulaciji je dodavanje na već postojeću vrijednost. U *CUDA* jezgri operacija baš za to već postoji i prima memorijsku adresu i vrijednost za koju se želi uvećati ta memorijska adresa, a vraća staru vrijednost te memorijske lokacije.

```
atomicAdd(&x[atom_1_idx], -col_vec_x * (mass1 / (mass1 + mass2)));
atomicAdd(&y[atom_1_idx], -col_vec_y * (mass1 / (mass1 + mass2)));

atomicAdd(&x[atom_2_idx], col_vec_x * (mass2 / (mass1 + mass2)));
atomicAdd(&y[atom_2_idx], col_vec_y * (mass2 / (mass1 + mass2)));
```

Kod 6.9 – Atomarna operacija dodavanja na postojeću vrijednost u *CUDA* jezgri

Primjer toga prikazuje Kod 6.9. Tako da je s time automatski osigurano da će podatci na tim adresama ostati konzistentni između svih dretvi.

6.2.4 Kritični odsječak za specifičnu lokaciju

Opet je potrebno osigurati da umetanje čestice na neku specifičnu lokaciju u stablo radi samo jedna dretva. No ovdje je to moguće ostvariti bez korištenja dodatnog polja za zaključavanje zbog jedne atomarne operacije zvane `atomicCAS` (*eng. Atomic Compare And Swap*) [18] koja omogućuje da dretva može direktno usporediti vrijednost neke memorijske lokacije i ako je uvjet zadovoljen neprekinuto postaviti na novu vrijednost.

```
int atomicCAS(int* address, int compare, int val);
```

Kod 6.10 – Potpis `atomicCAS` operacije nad cijelim brojevima

Kod 6.10 prikazuje primjer kako izgleda potpis funkcije `atomicCAS` koja se može koristiti u *CUDA* jezgri. Prima memorijska lokacija, vrijednost za uspoređivanje i vrijednost na koju treba postaviti tu memorijsku adresu ako se uspostavi da na toj memorijskoj lokaciji piše dana vrijednost.

Kod izgradnje stabla potrebno je osigurati da samo jedna dretva može pristupiti određenom listu u polju s djecom, to je moguće ostvariti upotrebom ove operacije tako da dretva koja pronađe list na koji želi umetnuti novu česticu prvo provjeri piše li na lokaciji tog lista još uvijek njegova vrijednost. U slučaju da je vrijednost još uvijek jednaka listu postavlja broj –2 na tu lokaciju i kreće s umetanjem. Tako da će broj –2 značiti da je ćelija zaključana. Tako da kada neka druga dretva utvrdi da je list koji je pronašla postavljen na –2 treba čekati da se promijeni na neku drugu vrijednost i onda može nastaviti s pronalaskom lista.

```
// .. dretva je pronašla list 'leaf' na indeksu 'leafIndex'
int OldLeafValue = atomicCAS((int*)&child[leafIndex], leaf, -2)
if (OldLeafValue == leaf) {
  // .. kreni s umetanjem
}
while(true) {
  currentNode = child[leafIndex]
  if (CurrentNode!= -2){ // na mjestu lista je umetnut novi čvor
    break;   // nastavi s pronalaskom
  }
}
```

Kod 6.11 – Primjer koda za zaključavanje lista korištenjem funkcije `atomicCAS`

7 Rezultati

7.1 Sklopovlje

Sklopovlje korišteno kod testiranja performanci je procesor AMD Ryzen 7 7700X BOX koji ima 8 fizičkih jezgri (16 logičkih), grafička kartica Gigabyte Aorus GeForce RTX 4070 MASTER 12GB [20] i 32 GB DDR5 5200MHz privremene memorije.

Iako procesor pomoću Hyper-Threading [19] tehnologije ima 16 logičkih dretvi, samo 8 od tih 16 su fizičke jezgre. Znači da se svaka fizička jezgra operacijskom sustavu predstavlja kao procesor s dvije jezgre. To je odličan pristup za ubrzavanje procesa na operacijskom sustavu, pogotovo onih koji rade puno ulazno-izlaznih (eng. Input/output, IO) operacija. No pretpostavka je da u ovakvoj simulaciji gdje je puno više operacija računanja od onih koji pristupaju memoriji da će postojati vidljivo ubrzanje prilikom povećavanja broja dretvi do 8, a više od toga neće donositi neko dodatno ubrzanje.

Korištena grafička kartica ima 5888 jezgri što je tri reda veličine više od procesora. No svaka jezgra ime frekvenciju od 2475 Mhz što je dvostruko manje od prosječne brzine jezgre procesora. Tako da je teoretska ukupna brzina svih procesora na grafičkoj kartici više od 100 puta veća od brzine na procesora. No procesor je znatno fleksibilniji i ima pristup većem satu naredaba. Tako da što je manje sinkronizacije potrebno između dretvi to će brzina na grafičkoj kartici biti bliža teoretskoj u usporedbi s procesorom.

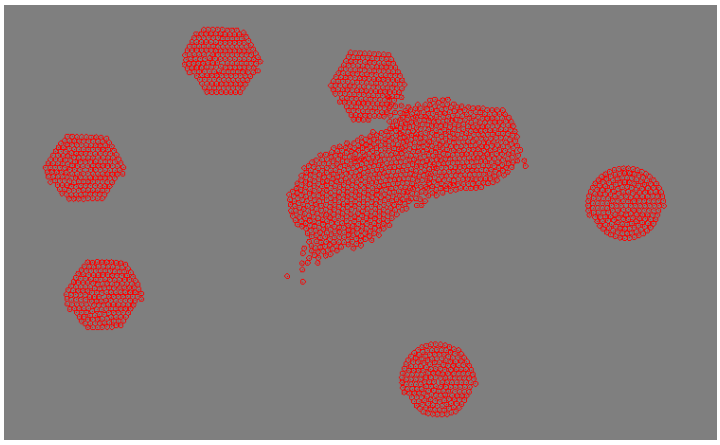
7.2 Metodologija mjerenja performanci

Bitno je da prilikom mjerenja performaci uvjeti simulaciji budu što sličniji, a onda da se mijenjaju samo određene varijable ovisno o potrebi. U ovom radu će se raditi dva različita testa i za svaki od njih će biti potrebno ponoviti s različitim parametrima.

Parametri koji su stalni u jednom izvođenju testa, ali se mijenjaju iz testa u testu su korišteni algoritma, broj dretvi ili radi li se o grafičkoj kartici i broj iteracija rješavanja sudara.

Rezultati svakog testa s određenim parametrima su vremena izvođenja svakog od koraka i oznaka vremena kada je izračun napravljen u simulaciji.

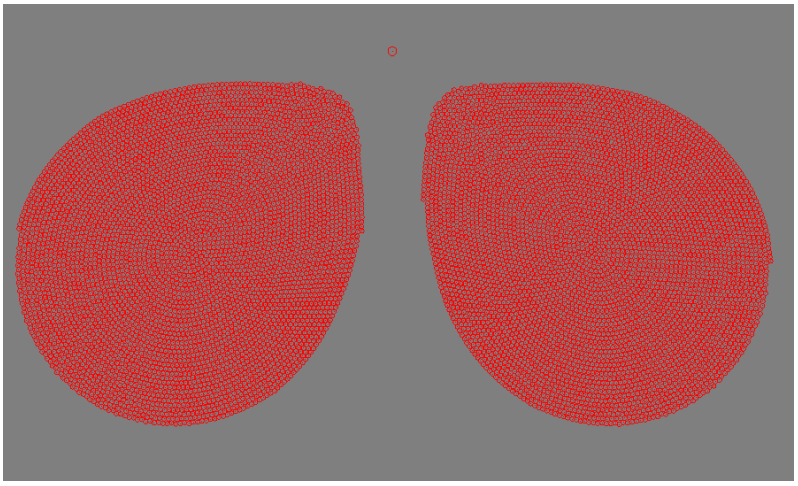
Prvi test je će biti dodavanje čestica kroz vrijeme izvođenja simulacije. Sve čestice će biti iste veličine i mase, a cilj testa je izmjeriti ovisnost vremena izvođenja o broju čestica.



Slika 7.1 – Primjer dodavanja čestica kroz vrijeme u prvom testu

Slika 7.1 prikazuje kako izgleda dodavanje novih čestica prilikom provođenja prvog testa. Na nasumičnim pozicijama generiraju se nakupine čestica, uglavnom 100 ili 500 odjednom ovisno o tome koliko čestica je potrebno ukupno dodati. S dodavanjem čestica se staje kada vrijeme simulacije poraste iznad 120 milisekundi.

Drugi test će služiti za mjerenje prosječnog vremena izvođenja za stalan broj čestica kako bi se detaljnije mogla analizirati vremena izvođenja svakog od koraka pojedinog algoritma. Ovo se radi zbog smanjenja dimenzionalnosti testa tako da bi se povećala preglednost, jer tako je na jednom grafu moguće puno preglednije analizirati i uspoređivati vrijeme izvođenja između više testova s različitim konfiguracijama parametara. No s obzirom da se uzima prosječno vrijeme izvođenja gubi se dimenzija promjene kroz vrijeme.



Slika 7.2 - Primjer inicijaliziracije čestica za drugi test

S obzirom da za ovaj test nije bitno kako se mijenja vrijeme simulacije ovisno o broju čestica sve čestice su generiranje u isto vrijeme što pokazuje Slika 7.2. Stvorene su dvije velike grupe čestica s manjom masom, 500 po grupi kod testiranja naivnog algoritma i 5,000 po

grupi kod testiranja Barnes-Hut algoritma. To se radi zato da bi se lakše vidjele razlike među konfiguraciji u vremenima izvođenja između 0 i 150ms. Jer ideja je provjeriti performanse algoritama u uvjetima za izvođenja u stvarnom vremenu.

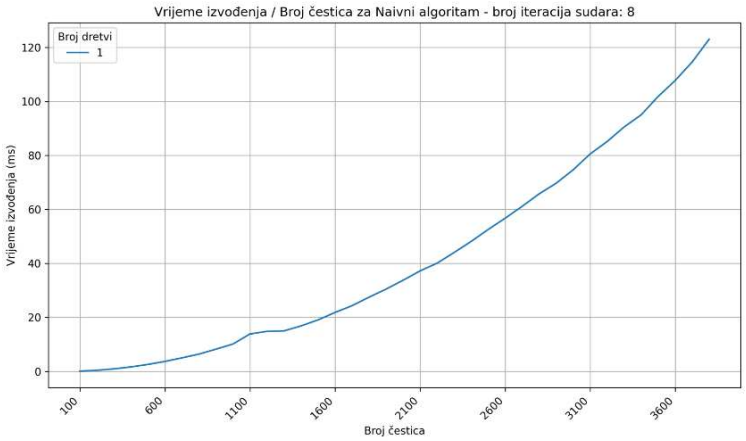
Dodatno se generira jedna čestica s većom masom tako da bi se simulirao sustav u kojem postoji dosta kaotičnog i brzog gibanja. Razlog tome je da provođenje ovog testa bude što bliže nekom realnijem nesimetričnom sustavu, jer da su sve čestice iste veličine dogodilo bi se samo grupiranje svih čestica u jednu nakupinu što je prilično optimistična situacija za Barnes-Hut algoritam jer takva raspodjela rezultira prilično balansiranim četvrtastim stablom.

Slika 7.3 - Rezultat simulacije u drugom testu nakon nekog vremena

Slika 7.3 prikazuje primjer položaja čestica nakon nekog vremena izvođenja simulacije. Može se vidjeti da se stvaraju razni zanimljivi obrasci i da je sustav poprilično kaotičan.

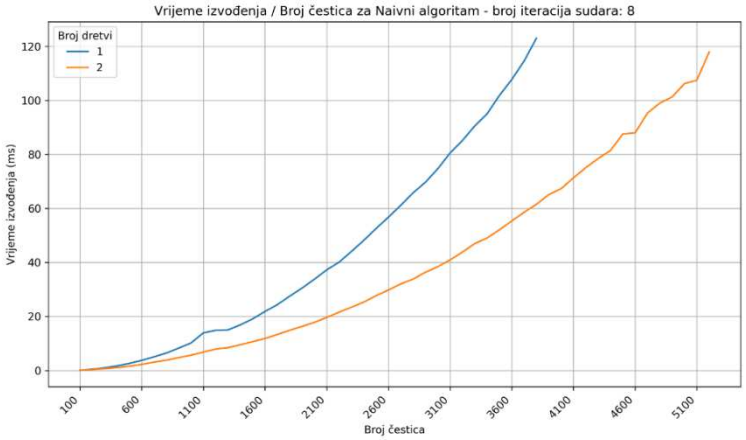
7.3 Naivni algoritam

S obzirom da se ovdje o vremenskoj složenosti $O(N^2)$ očekivano je da vrijeme potrebno za izračun fizike raste kvadratno broju čestica.



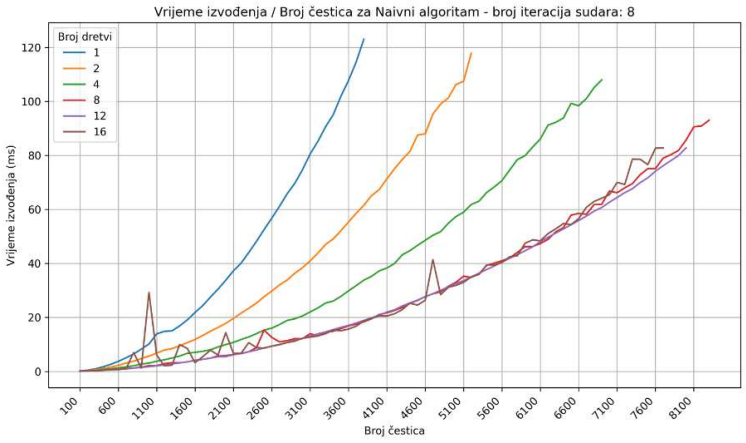
Slika 7.4 - Ovisnost brzine izvođenja naivnog algoritma broju čestica (1 dretva)

Rezultat izvođenja naivnog algoritma na jednoj dretvi prikazuje Slika 7.4. Vidljivo je da vrijeme raste približno u obliku parabole, što je i očekivano iz vremenske složenosti. Isto tako se vidi da već nakon 1000 čestica vrijeme izračuna fizike raste preko 16ms, a nakon 3500 iznad 100ms što više nije pogodno za simulaciju u stvarnom vremenu.



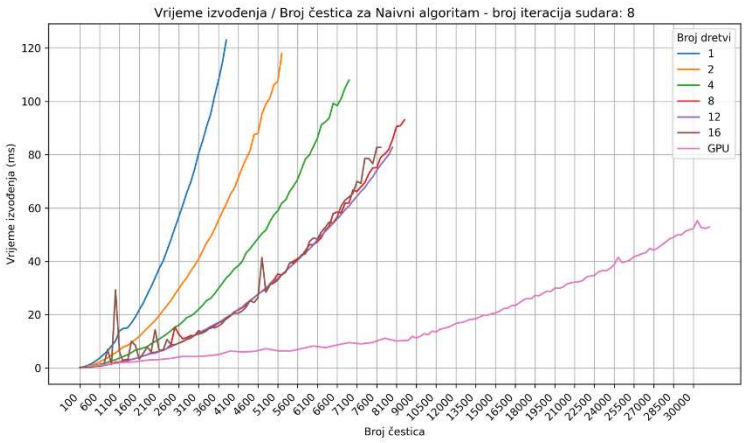
Slika 7.5 - Ovisnost brzine izvođenja naivnog algoritma o broju čestica (1 i 2 dretve)

Kada se broj dretvi poveća s jedan na dva teoretsko vrijeme izvođenja dvostruko bi se trebalo smanjiti, što bi na grafu izgledalo tako da se sve vrijednosti na y osi smanje na pola. No razliku vremena izvođenja naivnog algoritma na jednoj i dvije dretve prikazuje Slika 7.5 prikazuje da vrijeme izvođenja nije dva puta manje, ali da je oko 1.5 puta što se može objasniti da je dio vremena potrebno potrošiti na sinkronizaciju dretvi.



Slika 7.6 - Ovisnost brzine izvođenja naivnog algoritma o broju čestica (1, 2, 4, 8, 12 i 16 dretvi)

Slika 7.6 prikazuje vrijeme izvođenja za nekoliko konfiguracija broja dretvi. Vrijeme izvođenja pada proporcionalno broju dretvi sve dok broj dretvi ne postane osam. Kada je broj dretvi veći od osam vrijeme izvođenje ostaje isto kao i za osam dretvi. Razlog tome je što su ovi testovi provođeni na procesoru s osam fizičkih jezgri, a većina posla kojega dretve trebaju obaviti je direktno računanje kojega je moguće ubrzati samo fizičkim procesorima. Da se radi o algoritmima koji trebaju raditi mnogo sporih operacija koje se ne izvode na procesoru kao čitanje s diska ili mreže onda bi se moglo dobiti dodatno ubrzanje s više dretve, no to ovdje očito nije slučaj.

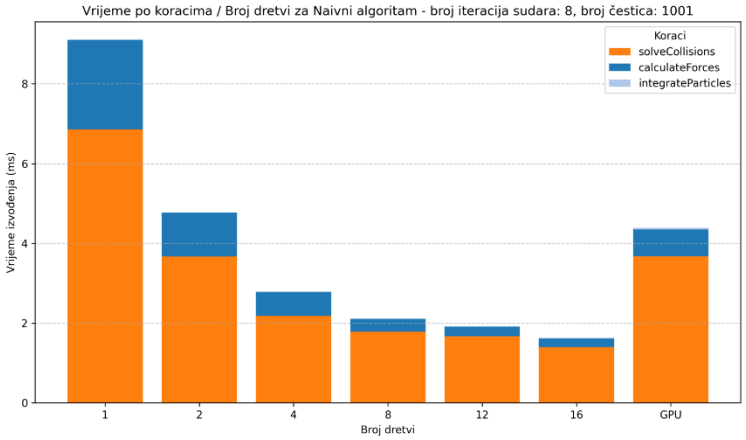


Slika 7.7 - Ovisnost brzine izvođenja naivnog algoritma o broju čestica (1, 2, 4, 8, 12,16 dretvi i GPU)

Iako grafička kartica teoretski ima nekoliko redova veličine više fizičkih jezgri neko procesor, Slika 7.7 prikazuje da je naivni algoritam koji se izvodi na grafičkoj kartici samo 10-15 puta brži od istog tog algoritma koji se izvodi na jednoj dretvi na procesoru. Razlog

tome je da što ima više dretvi to se više vremena troši na sinkronizaciju, a i to da su jezgre na grafičkoj kartici generalno sporije i manje kompleksne od onih na procesoru.

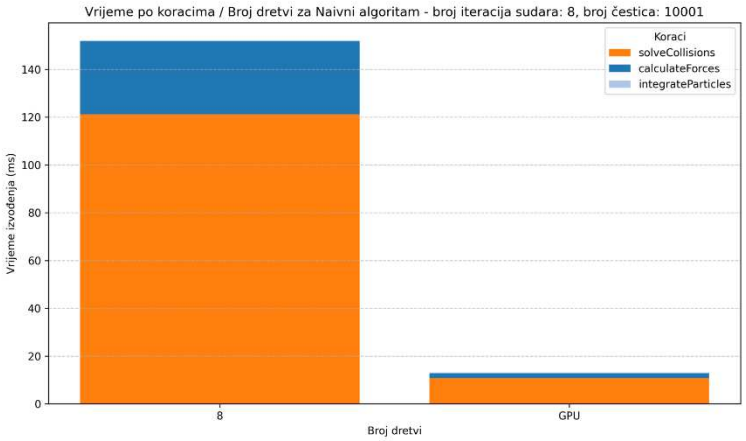
Naivni algoritam sastoji od tri koraka od kojega koraci rješavanja sudara i izračuna gravitacijskog utjecaja imaju kvadratnu složenost, a korak ažuriranja čestica ima linearnu složenost. Tako da je očekivano da će većinu ukupnog vremena izvođenja uzeti koraci s kvadratnom složenosti.



Slika 7.8 - Vremena potrebna za izvođenje svakog od koraka naivnog algoritma za različit broj dretvi

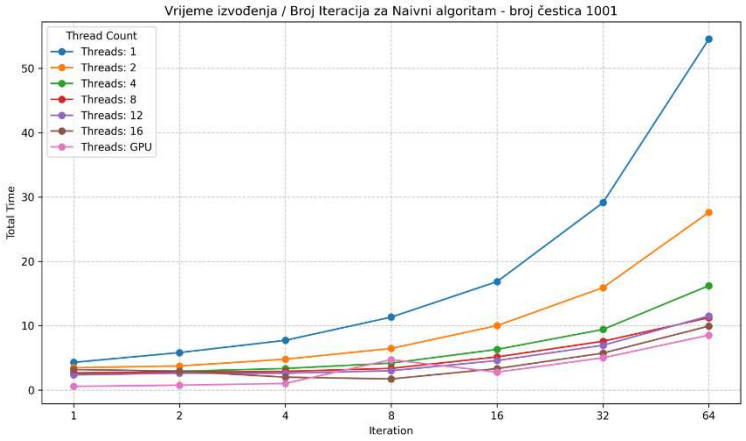
Slika 7.8 prikazuje raspodjelu vremena izvođenja svakog od tri koraka za oko 1000 čestica na različitom broju dretvi. Vrijeme potrebno za izvođenje sudara uzima najviše vremena, a to je zato jer se u ovom primjeru radi osam iteracija rješavanja sudara. Tako da je vrijeme potrebno za to otprilike osam puta veće od vremena potrebnog za izračun gravitacijskog utjecaja. Isto tako se vidi da vrijeme potrebno za ažuriranje čestica je toliko malo naspram druga dva koraka da se niti ne može vidjeti na grafu za oko 1000 čestica.

U ovom primjeru kada se radi o vrlo malom broju čestica vrijeme potrebno grafičkoj kartici da izvede sve korake je sporije od izvođenja na skoro svim konfiguracija dretvi na procesoru. To je zato što se dosta vremena gubi na prenošenje podataka s procesora na grafičku karticu i obrnuto. To vrijeme ne ovisi o broju čestica, ali s obzirom da se ovdje izračuni odrade jako brzo s obzirom na mali broj čestica, vrijeme potrebno za prijenos podatka koje je konstanto postaje velik dio ukupnog vremena potrebnog za izračun.



Slika 7.9 - Vremena potrebna za izvođenje svakog od koraka naivnog algoritma za osam dretvi i GPU za veći broj čestica

Kada se poveća broj čestica ukupno vrijeme potrebno za prijenos podataka ostaje isto, ali je sada manji dio ukupnog izračuna. Naravno vrijeme izvođenja na procesoru znatno raste i sada se puno bolje vidi razlika između procesora i grafičke kartice. No isto tako se vidi da omjer vremena za svaki korak ostaje isto za oboje.

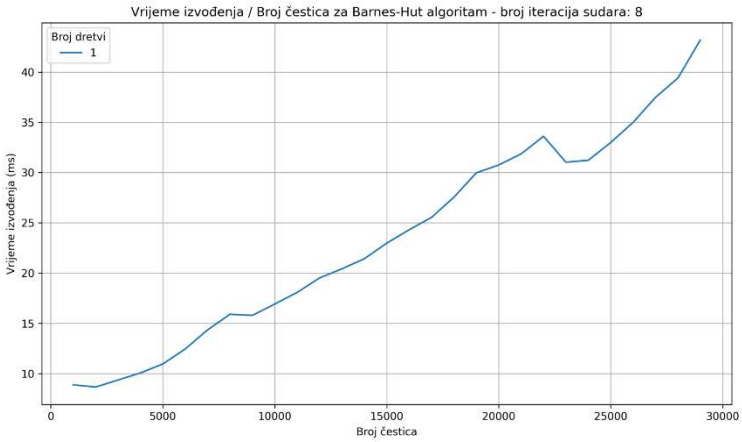


Slika 7.10 - Vremena potrebna za izvođenje naivnog algoritma za različit broj iteracija rješavanja sudara

Ukupno vrijeme izvođenja naivnog algoritma je skoro direktno proporcionalno broju iteracija rješavanja sudara. To je zato jer jedna iteracija izračuna sudara traje slično kao i izračun gravitacijskog utjecaja, a onda što je više tih iteracija to će rješavanje sudara zauzimati veći postotak ukupnog vremena izvođenja algoritma. Tu ovisnost prikazuje Slika 7.10.

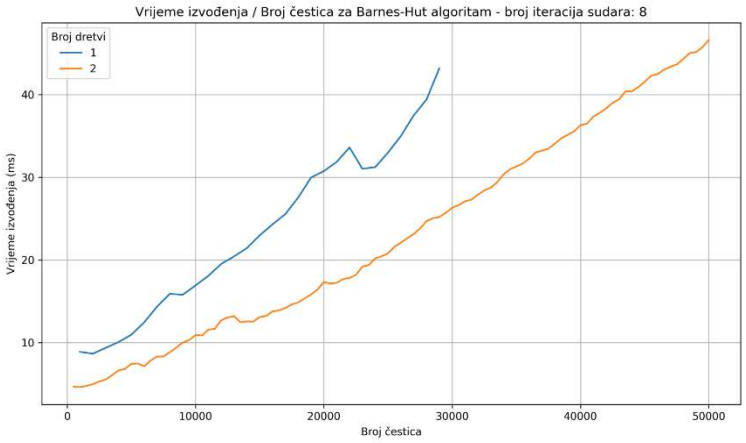
7.4 Barnes-Hut algoritam

S obzirom da Barnes-Hut algoritma ima N*log(N) složenost očekivano je da će rast vremena izvođenja isto pratiti tu funkciju.



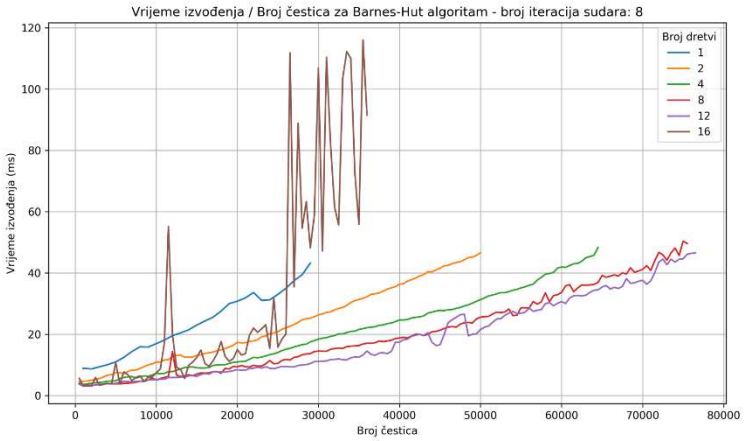
Slika 7.11 - Ovisnost brzine izvođenja Barnes-Hut algoritma o broju čestica (1 dretva)

Slika 7.11 prikazuje rast vremena izvođenja Barnes-Hut algoritma ovisno o broju čestica kada se algoritam izvodi na jednoj dretvi. Prikazana krivulja je jako blizu linearne krivulje koja znatno sporije raste od kvadratne krivulje. Tako da je na samo jednoj dretvi već moguće simulirati i 20,000-30,000 čestica u stvarnom vremenu.



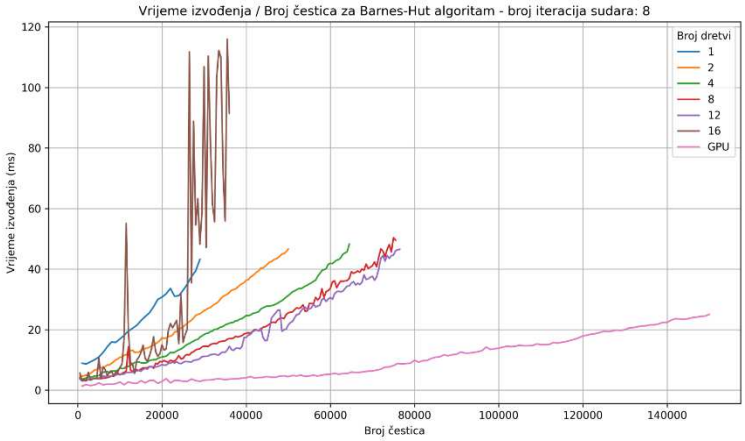
Slika 7.12 - Ovisnost brzine izvođenja Barnes-Hut algoritma o broju čestica (1 i 2 dretve)

Slika 7.12 prikazano je ubrzanje dobiveno kada se algoritam izvodi na dvije umjesto jednoj dretvi. Ubrzanje nije dvostruko veće, no poprilično je blizu tome. Razlog tome je isti kao i kod naivnog algoritma, a to je da se na sinkronizaciju gubi određeno vrijeme.



Slika 7.13 - Ovisnost brzine izvođenja Barnes-Hut algoritma o broju čestica (1, 2, 4, 8, 12 i 16 dretvi)

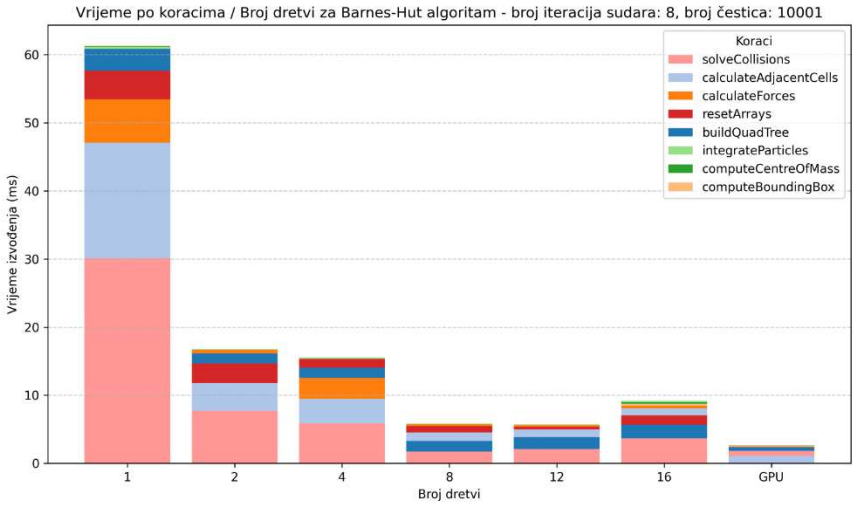
Kada se uspoređuje dobiveno ubrzanje kada se koristi više dretvi situacija je ista kao i kod naivnog algoritma. Povećanje broja dretvi rezultira manjim vremenom izvođenja sve do osam dretvi, a nakon toga se više ne dobiva ubrzanje. No dolazi do zanimljive situacije kada se koristi svih 16 logičkih dretvi. Vrijeme izvođenja je generalno cijelo vrijeme veće od toga na više dretvi, no kada se simulira više od 20,000 čestica vrijeme znatno poraste. Razlog tome nije potpuno jasan, moguće je da se operacijski sustav uplete u raspoređivanje dretava i to dovede do situacije gdje je dretva koja drži ključ za neki dio memorije često neaktivna i time, a druge je dretve čekaju. Isto tako je moguće da postoji neki problem u implementaciji algoritma koji se javlja samo kada se koriste sve logičke dretve na procesoru.



Slika 7.14 - Ovisnost brzine izvođenja Barnes-Hut algoritma o broju čestica (1, 2, 4, 8, 12 i 16 dretvi i GPU)

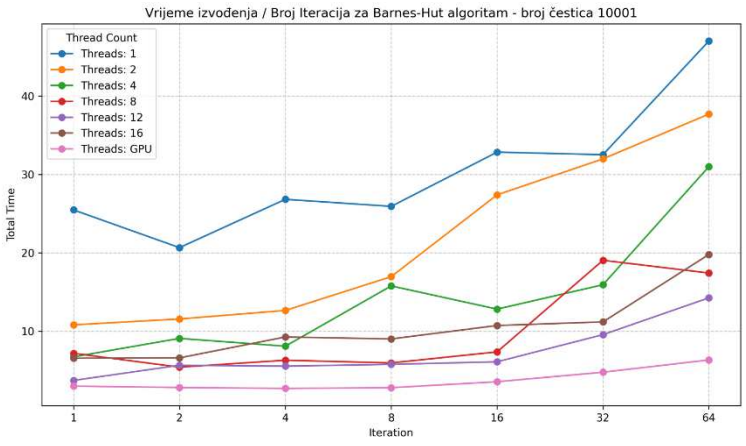
Kada se Barnes-Hut algoritam izvodi na grafičkoj kartici dobije se ubrzanje nad jednom dretvom procesora od 10-15 puta. Opet vrlo slično kao i kod naivnog algoritma. Grafička

kartica i Barnes-Hut algoritam u ovom slučaju omogućuju simulaciju više od 140,000 čestica u stvarnom vremenu.



Slika 7.15 - Vremena potrebna za izvođenje svakog od koraka Barnes-Hut algoritma za različit broj dretvi

Barnes-Hut algoritam se sastoji od nekoliko koraka čija su vremena izračuna prikazana na Slika 7.15. Vidljivo je da omjeri vremena nisu konzistentni kod izvođenja na različitom broju dretvi. To je iz više razloga, jedan od njih je da je sinkronizacija izračuna stabla i rješavanja sudara teža za veći broj dretvi, isto tako vrijeme izračun susjednih čestica varira ovisno o položajima čestica. No generalno najviše vremena se troši na izračun sudara pod koji spadaju koraci izračuna susjednih čestica i samog izračuna sudara.

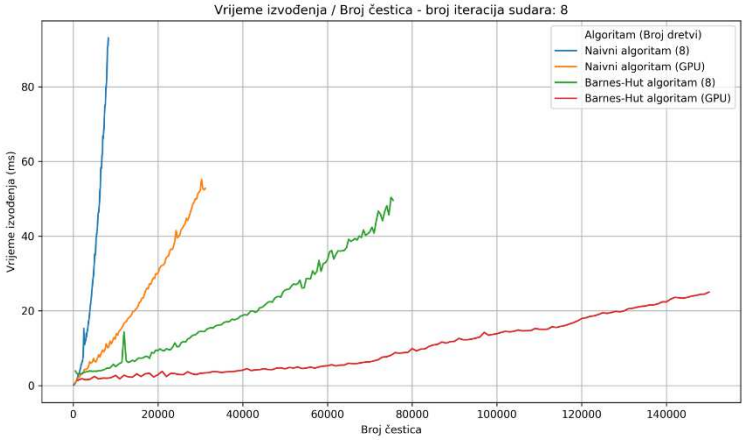


Slika 7.16 - Vremena potrebna za izvođenje Barnes-Hut algoritma za različit broj iteracija rješavanja sudara

Još jedna dobra stvar kod Barnes-Hut algoritma je mogućnost da se susjedne čestice izračunaju jednom, da se sudari izračunaju u više iteracija. Iako nije garantirano da će svi

sudari biti riješeni u naknadnim iteracijama velik broj čestica će ostati u svojim ćelijama ili blizu njih i još uvijek će imati točne podatke o svojim susjedima. Tako se štedi puno vremena za izračun sudara jer samo izračun susjednih čestica ovisi o broju čestica, naknadne iteracije izračuna sudara više ne ovise o broju čestica. Slika 7.16 prikazuje kako se povećava vrijeme izvođenja s povećanjem iteracija rješavanja sudara. Vidljivo je da vrijeme izvođenja sporo raste u ovisnosti s brojem tih iteracija, za razliku od naivnog algoritma.

7.5 Usporedba algoritama

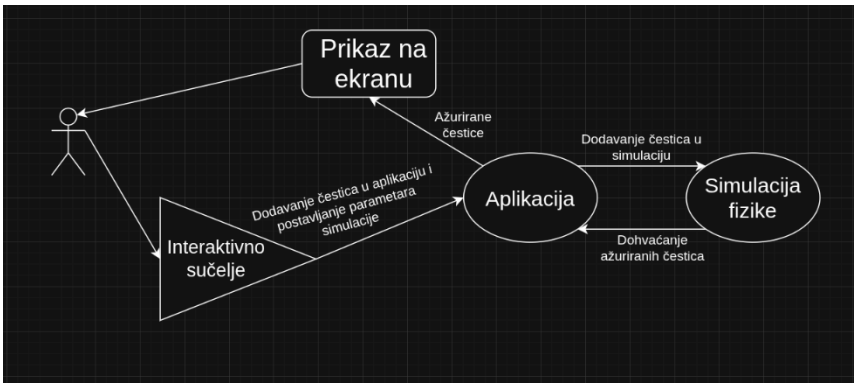


Slika 7.17 - Vrijeme izvođenja ovisno o broju iteracija rješavanja sudara za oba algoritma na 1, 4, 8 dretvi i grafičkoj kartici

Slika 7.17 prikazuje odnos vremena izvođenja naivnog i Barnes-Hut algoritma i koliko je Barnes-Hut algoritma brži od naivnog za velik broj čestica. Kada se uspoređuje broj čestica koje je moguće simulirati u stvarnom vremenu na procesoru s osam dretvi s Barnes-Hut algoritmom je moguće simulirati oko 50,000 više čestica. Što se tiče grafičke kartice ovdje Barnes-Hut algoritam može simulirati oko 100,000 više čestica za rad u stvarnom vremenu.

8 Programsko rješenje

Izračun fizike je samo jedan od tri glavna koraka cjelokupnom programskom rješenju. Osim toga potrebno je još omogućiti prikaz čestica koje se simuliraju i omogućiti korisniku simulacije da upravlja simulacijom ili dobiva neke dodatne informacije.



Slika 8.1 - Prikaz arhitekture programskog rješenja

8.1 Prikaz na zaslonu

Prikaz na ekranu je omogućen korištenjem biblioteke *FreeGLUT* [21] koja kroz jednostavno sučelje način omogućuje prikaz prozora simulacije, postavljanja koordinatnog sustava i osnovnog prikaza objekata u tom sustavu. U pozadini koristi *OpenGL* grafičko sučelje [22] za komunikaciju s operacijskim sustavom i grafičkom karticom.

```
glutInit(&argc, argv);
glutInitWindowSize(width, height);
glutInitWindowPosition(100, 100);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutCreateWindow("2D Gravity Simulation");
glutDisplayFunc(MainLoopStep);
glClearColor(0.0, 0.0, 0.0, 1.0);
gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
```

Kod 8.1 - Inicijalizacija prikaza koristeći *OpenGLUT*

Kod 8.1 prikazuje sve što je potrebno napraviti da bi se pomoću *OpenGLUT*-a prikazao prozor i inicijalizirao koordinatni sustav da bude spreman za prikaz simulacije.

```
void drawCircle(double xPos, double yPos, double radius,
                int num_segments, glm::vec3 color) {
  glBegin(GL_LINE_LOOP);
  for (int i = 0; i < num_segments; i++) {
```

```
      float theta = 2.0f * 3.1415926f * float(i) / float(num_segments);
      float x = radius * cosf(theta);
      float y = radius * sinf(theta);
      glColor3f(color.x, color.y, color.z);
      glVertex2f(x + xPos, y + yPos);
    }
    glEnd();
    glPointSize(1.0f);
    glBegin(GL_POINTS);
    glVertex2f(xPos, yPos);
    glEnd();
}
```

Kod 8.2 – Prikaz dvodimenzionalnog kružića koristeći *OpenGLUT*

Kod 8.2 prikazuje funkcija zaduženu za crtanje jedne čestice u obliku kružića. Svaki kružić je napravljen od fiksnog broja točaka koje su povezane s ravnom linijom i dodatne točke u sredini.

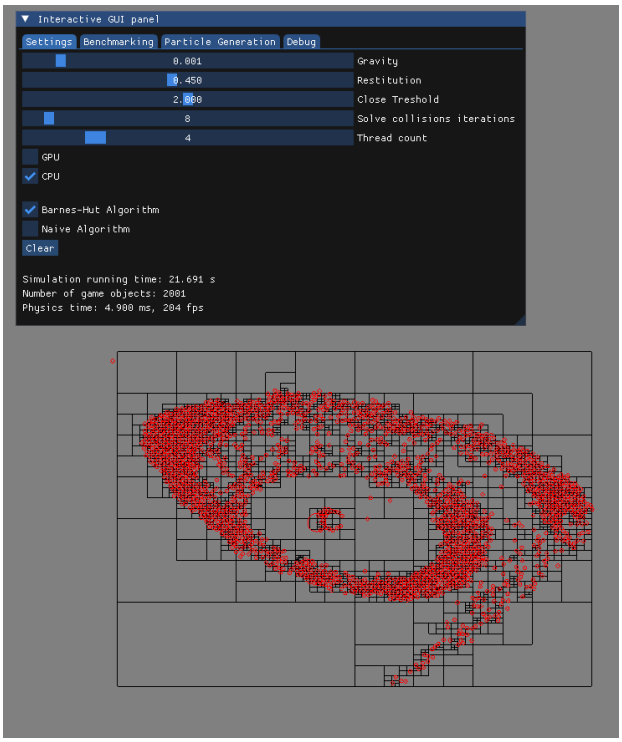
8.2 Interaktivno sučelje

Interaktivno sučelje je implementirano koristeći biblioteku *Dear ImGui* [23] koja pruža vrlo jednostavno sučelje za implementaciju interaktivnog grafičkog sučelja. Kompatibilna je za većinu grafičkih sučelja, uključujući *OpenGL* u kojega koristi ova aplikacija.

```
ImGui_ImplOpenGL2_NewFrame();
ImGui_ImplGLUT_NewFrame();
ImGui::NewFrame();
ImGui::Begin("Interactive GUI panel");
if (ImGui::BeginTabBar("MainTabBar")) {
  if (ImGui::BeginTabItem("Settings")) {
    gravity = physicsEngine.getGravity();
    if(ImGui::SliderFloat("Gravity", &gravity, 0, 0.01)){
      physicsEngine.setGravity(gravity);
    }
    // . . . Ostatak koda za ovu karticu
   }
}
ImGui::EndTabItem();
// . . . Ostatak koda za druge kartice
```

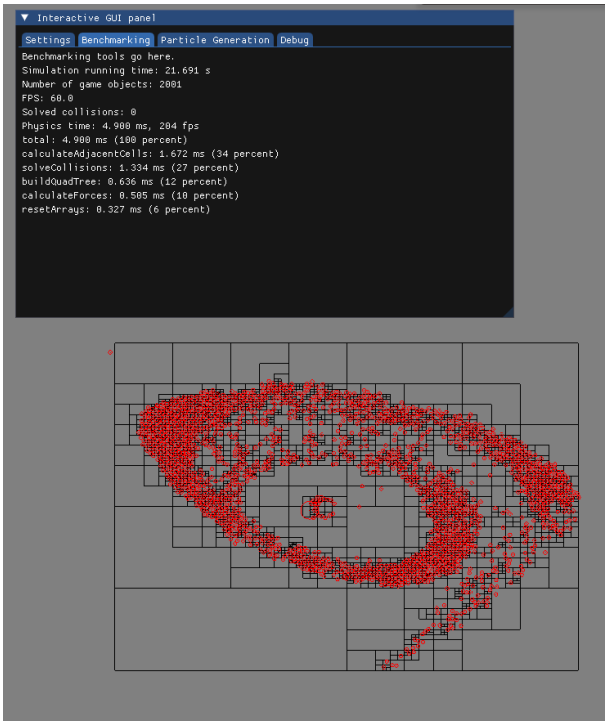
```
ImGui::EndTabBar();
ImGui::Render();
```

Kod 8.3 - Primjer dodavanja kliznog pokazivača za postavljen gravitacije



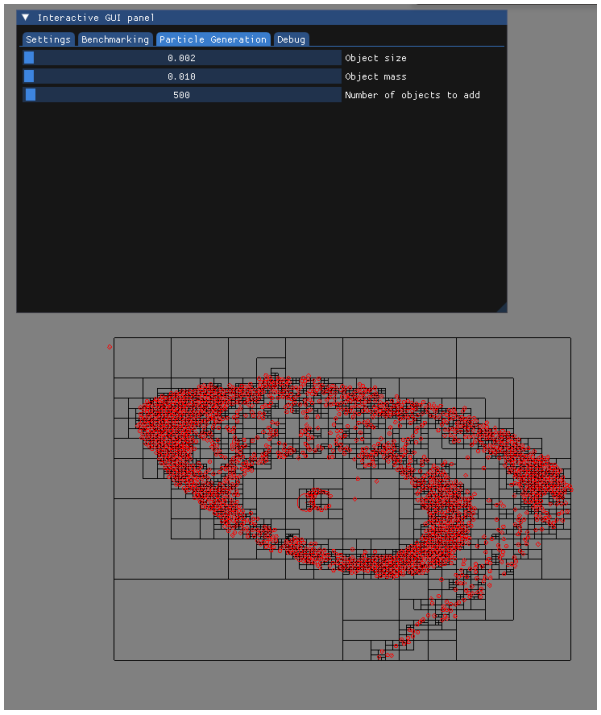
Slika 8.2 - Postavke parametara simulacije

Slika 8.2 prikazuje koje je sve parametre moguće mijenjati u simulaciji. Ovdje se bira algoritam koji će koristit za izračun i uređaj koji će se koristiti za izračun. Isto tako je moguće postaviti broj dretvi ako se koristiti procesor i broj iteracija algoritma za rješavanje sudara.



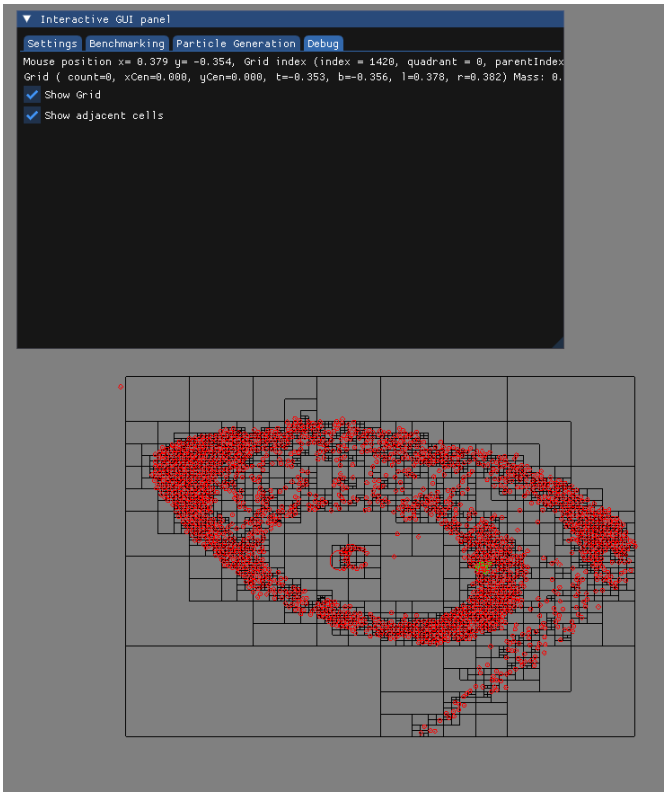
Slika 8.3 - Analiza performanci simulacije

Slika 8.3 prikazuje različite statistike koje se tiču performaci simulacije. Kao recimo ukupno vrijeme potrebno za izračun fizike i analiza trajanja svakog od koraka simulacije u milisekundama i postotku ukupnog vremena.



Slika 8.4 - Postavke za generiranje čestica

Parametre koje prikazuje Slika 8.4 koriste se kod generacije čestica. To se radi pritiskom lijeve tipke miše ili pritiskom slova R na tipkovnici. Lijeva tipka miša stvara jednu česticu na poziciji pokazivača unutar simulacije, a slovo R stvara grupu čestica u obliku ispunjenog kruga.



Slika 8.5 - Pomoćne opcije

Opcije koje prikazuje Slika 8.5 služe za pronalaženje i detekciju potencijalnih problema jer daju detaljan prikaz raznih elemenata simulacije. Moguće je prikazati kako izgleda mreža izgrađena Barnes-Hut algoritmom, pronađene susjedne čestice, pozicija miša, trenutna ćelija...

U simulaciji je dodatno moguće približiti i udaljiti kamera ovisno o lokaciji pokazivača. To se radi tipkama *Ctrl* za udaljavanje i *Shit* za približavanje. Moguće je pomicanje kamere tako da se pritisne i drži desna tipka miša i pritom povlači u željenom smjeru. Još je moguće i pauzirati simulacije pritiskom tipke *Space*.

8.3 Kompatibilnost sa sustavima bez Nvidia grafičke kartice

Simulacija je napravljena tako da ju je moguće pokrenuti i na sustavima koji nemaju instaliranu *CUDU* i/ili nemaju *Nvidia* grafičku karticu. To je učinjeno zato jer je pokretanje na grafičkoj kartici samo jedna od mogućnosti ove simulacije i nema razloga da bez *Nvidia* grafičke kartice simulacija na radi. Tako da sustavi bez *CUDA*-e mogu pokretati sve ostale dijelove simulacije, a simulacija na grafičkoj kartici samo neće raditi.

Za upravljanje prevođenja projekta koristi se program *CMake* [24] koji omogućuje prevođenja na više operacijskih sustava koristeći istu konfiguracijsku datoteku i isti izvorni kod. To je postignuto u tri koraka. Prvi korak je detektirati je li *CUDA* instalirana na sustavu.

```
find_package(CUDA QUIET)
if(CUDA_FOUND)
    message(STATUS "CUDA found, building with GPU support")
    set(USE_CUDA TRUE)
else()
    message(WARNING "CUDA not found, building without GPU
support")
    set(USE_CUDA FALSE)
endif()
```

Kod 8.4 - Isječak *CMake* datoteke za provjeri je li *CUDA* instalirana

Nakon tog koraka u *CMake*-u je postavljena varijabla `USE_CUDA` i sve ostale konfiguracije potrebne za prevođenje *CUDA* programa su dodane ako je `USE_CUDA` varijabla postavljena na jedan.

Drugi korak je omogućiti da kod simulacije ima pristup toj varijabli da bi mogao isključiti dijelove koda koji koriste *CUDA* sučelje. To se radi tako da postoji jedna konfiguracijska datoteka koju *CMake* može izmijeniti ovisno o vrijednosti `USE_CUDA` varijable u format čitljiv prevoditelju *C++* jezika.

```
#pragma once
#cmakedefine USE_CUDA
```

Kod 8.5 - Konfiguracijska datoteka koju *CMake* može mijenjati

To se u *CMake*-u radi naredbom `configure_`file koja postavlja datoteku na zadanu putanju i pri tome postavlja sve varijable označene s #cmakedefine koja su istinite u vrijeme izvođenja te naredbe.

Konačni korak je unutar *C++* datoteka postaviti sav kod koji koristi *CUDA* sučelje unutar ako/inače blokova koji će se prevoditi ovisno o vrijednosti `USE_CUDA` varijable.

```
#ifdef USE_CUDA
//... Kod za pozivanje GPU algoritma
#endif
```

Kod 8.6 - Primjer provjere varijable `USE_CUDA` unutar *C++* koda

Time je postignuto da se kod koji koristi *CUDA* sučelje neće postojati tijekom prevođenja, ako varijabla `USE_CUDA` nije postavljena.

8.4 Moguća unaprjeđenja

Simulaciju o ovom radu je moguće unaprijediti na razne načine. S obzirom da je izvođenje simualcije na grafičkoj kartici jedan od glavnih značajki ove simulacije bilo bi odlično kada bi se simulacija mogla izvoditi na bilo kojoj grafičkoj kartici. To bi se moglo omogućiti korištenjem alata *OpenCL* [25].

Još jedno poboljšanje simulacije moglo bi biti prelazak iz 2D simulacije u 3D simulacije. Omogućiti tu značajku ne bi trebalo biti preteško za implementirati jer je sve kalkulacije isto rade i u tri dimenzije.

Isto tako bi bilo moguće dodati kompleksnije oblike čestice, što bi otežalo algoritam detekcije sudara, ali bi omogućilo znatno veći broj različitih simulacija.

Još bi bilo dobro poraditi na algoritmu rješavana sudara da se tu omogući veća fizička točnost što se tiče elastičnosti sudara i drugih karakteristika sudara.

Zaključak

Simulacija gravitacije između čestica ima mnogo primjena kao što su u znanosti i industrijama video igara ili filmova. Sam izračun gravitacije i sudara između čestica je jednostavan, no takav naivan algoritam ima kvadratno vrijeme izvođenja koja prebrzo raste ovisno o broju čestica. Tako da takav algoritam nije pogodan kod simulacija velikih sustava čestica.

Zato je osmišljen Barnes-Hut algoritam koji kvadratnu vremensku složenost izračuna gravitacijskog utjecaja smanjuje na loglinearnu što omogućuje simulaciju znatno većeg broja čestica u usporedbi s naivnim algoritmom. Isto tako koristeći strukture podataka ovog algoritma može se smanjiti i vremenska složenost izračuna sudara.

Dodatno oba algoritma se mogu efikasno paralelizirati izvođenjem na više dretvi procesora ili čak izvođenjem na grafičkoj kartici. U većini slučajeva nije potrebno previše vremena utrošiti na sinkronizaciju dretvi tako da se postiže vrlo dobar omjer između korištenog broja dretvi i postignutim ubrzanjem.

Performance simulacije pod utjecanjem raznih parametara su detaljno analizirane i uspoređene i iz rezultate je vidljivo da je postignuto značajno ubrzanje prilikom korištenja Barnes-Hut algoritma i dodatno ubrzanje kada su algoritmi paralelizirani.

Za prikaz i korištenje simulacije u ovom radu izrađeno je i interaktivno grafičko sučelje koje prikazuje čestice, omogućuje dodavanje novih čestica i postavljanje raznih parametara simulacije.

Literatura

[1] Frans Pretorius, N-body Simulations Introduction, Princeton University (2025, Siječanj). Poveznica: https://fpretori.scholar.princeton.edu/n-body-simulations-introduction ; Pristupljeno: 25.05.2025.

[2] Proposition 75, Theorem 35: p. 956 – I.Bernard Cohen and Anne Whitman, translators: Isaac Newton, The Principia: Mathematical Principles of Natural Philosophy

[3] Uspensky, James, Introduction to Mathematical Probability (1937)

[4] Jeff Thompson, COLLISION DETECTION (2017, Lipanj). Poveznica: https://www.jeffreythompson.org/collision-detection/circle-circle.php ; Pristupljeno: 20.06.2025.

[5] Physics Tutorial 2: Numerical Integration Methods, New Castle University (2017). Poveznica: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/physics2numericalintegrationmethods/2017%20Tutorial%202%20-%20Numerical%20Integration%20Methods.pdf ; Pristupljeno: 25.05.2025.

[6] Donald E. Knuth, Big Omicron and Big Omega and Big Theta, SIGACT News, 8(2):18-24, April-June 1976.

[7] Nayer Sharar, Efficient Numerical Methods for N-Body Simulations with Modern Computational Techniques, NHSJS (2024, Prosinac). Poveznica: https://nhsjs.com/2024/efficient-numerical-methods-for-n-body-simulations-with-modern-computational-techniques ; Pristupljeno: 07.06.2025.

[8] J. Barnes und P. Hut, A hierarchical O(N log N) force-calculation algorithm, Nature,324(4), (1986, Prosinac). Povezanica:. https://ui.adsabs.harvard.edu/abs/1986Natur.324..446B/abstract ; Pristupljeno: 08.06.2025

[9] Jim Kang, An interactive explanation of quadtrees (2014, Ožujak). Poveznica: https://jimkang.com/quadtreevis/ ; Pristupljeno: 10.06.2025.

[10] Tom Ventimiglia & Kevin Wayne, The Barnes-Hut Algorithm, (2011, veljača). Poveznica: https://arborjs.org/docs/barnes-hut ; Pristupljeno: 06.05.2025

[11] Raynal, Michel (2012). Concurrent Programming: Algorithms, Principles, and Foundations. Springer Science & Business Media. p. 9.

[12] Mutex lock for Linux Thread Synchronization, GeekForGeeks (2024, Svibanj). Poveznica: https://www.geeksforgeeks.org/linux-unix/mutex-lock-for-linux-thread-synchronization/ ; Pristupljeno: 20.06.2025.

[13] Nkugwa Mark William, WHAT ARE ATOMIC OPERATIONS IN SOFTWARE ENGINEERING, Medium (2023, Listopad). Poveznica: https://nkugwamarkwilliam.medium.com/what-are-atomic-operations-in-software-engineering-e55c8177b3e2 ; Pristupljeno: 20.06.2025.

[14] Fred Oh, What Is CUDA?, Nvidia (2012, Rujan). Poveznica: https://blogs.nvidia.com/blog/what-is-cuda-2/ ; Pristupljeno: 26.06.2025.

[15] OpenMP (2000, Svibanj). Poveznica: https://www.openmp.org/ ; Pristupljeno: 25.06.2025

[16] Pradeep Gupta, CUDA Refresher: The CUDA Programming Model, Nvidia (2020, Lipanj). Poveznica: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ ; Pristupuljeno: 26.06.2025.

[17] OneFlow, How to Choose the Grid Size and Block Size for a CUDA Kernel?, Medium (2022, Siječanj). Poveznica: https://oneflow2020.medium.com/how-to-choose-the-grid-size-and-block-size-for-a-cuda-kernel-d1ff1f0a7f92 ; Pristupljeno: 27.06.2025.

[18] CUDA C++ Programming Guide, Nvidia (2025, Svibanj). Poveznica: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomiccas ; Pristupljeno: 20.06.2025.

[19] Anda Ioana Enescu Buyruk, Keysight Connect #13 – All about CPU Hyperthreading, Keysight (2023, Svibanj). Poveznica: https://www.keysight.com/blogs/en/keys/culture/2023/05/03/keysight-connect-all-about-cpu-hyperthreading ; Pristupljeno: 25.06.2025.

[20] AORUS GeForce RTX™ 4070 MASTER 12G, Aorus (2023, Travanj). Poveznica: https://www.aorus.com/graphics-cards/gv-n4070aorus-m-12gd/Specification ; Pristupljeno: 26.06.2025.

[21] FreeGLUT (2000, Prosinac). Poveznica: https://freeglut.sourceforge.net/ ; Pristupljeno: 27.06.2025.

[22] OpenGL (1997, Srpanj). Poveznica: https://www.opengl.org/ ; Pristupljeno: 27.06.2025.

[23] Omar Cornut, Dear ImGui (2014, Kolovoz). Poveznica: 27.06.2025. https://github.com/ocornut/imgui ; Pristupljeno: 27.06.2025.

[24] CMake: The Standard Build System, CMake (2000). Poveznica: https://cmake.org/features/ ; Pristupljeno: 29.06.2025.

[25] OpenCL: OpenCL for Parallel Programming of Heterogeneous Systems (2010, Ožujak). Poveznica: https://www.khronos.org/opencl/ ; Pristupljeno: 30.06.2025.

Sažetak

Usporedba brzine izvođenja dva načina izračuna gravitacijskih sila među tijelima u grafičkoj simulaciji.

U ovom radu uspoređuje se naivni algoritma izračuna gravitacijskog utjecaja koji ima kvadratnu vremensku složenost s Barnes-Hut algoritmom koji ima loglinearnu vremensku složenost. Isto tako se objašnjava način paralelizacije oba algoritma i na procesoru, ali i na grafičkoj kartici. Krajnje analiziraju se rezultati izvođenja svakog od algoritma pri različitim konfiguracijama kao što su izvođenja procesoru ili na grafičkoj kartici i dodatno broj za različit broj dretvi na procesoru. Opisano je i cjelokupno programsko rješenje koje uključuje i prikaz i interaktivno sučelje.

Barnes-Hut algoritam, paralelizacija, kvadratna složenost, gravitacijski utjecaj, rješavanje sudra, grafička kartica, CUDA, četvrtasto stablo, performance, problem N-tijela

Summary

A comparison of the execution speed of two methods for calculating gravitational forces between bodies in a graphical simulation.

In this paper, the naive algorithm for calculating gravitational influence, which has quadratic time complexity, is compared with the Barnes-Hut algorithm, which has logarithmic-linear time complexity. Also, the method of parallelization of both algorithms is explained, both on the processor and on the graphics card. Finally, the results of executing each of the algorithms under different configurations are analyzed, such as execution on the processor or on the graphics card, and additionally for different numbers of threads on the processor. The entire software solution is also described, which includes visualization and an interactive interface.

Barnes-Hut algorithm, paralelization, quadrati complexity, gravity influence, collisio ndetection, graphics card, CUDA, quadtree, performance, N-body problem