SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Implementacija algoritma raspodjele prostora uniformom mrežom na grafičkoj kartici koristeći CUDA tehnologiju i usporedba performanci implementacije grafičke kartice i procesora

Mario Jalšovec

Voditelj: *Željka Mihajlović*

Zagreb, Siječanj, 2025.

Sadržaj

Sažetak

Rad istražuje ubrzanje izračuna sudara u sustavima čestica korištenjem paralelizacije na procesoru i grafičkoj kartici. Počinje opisom problema rješavanja sudara velikog broja čestica i važnosti ubrzanja istog za simulacije u stvarnom vremenu.

Dalje prednosti i nedostatci paralelizacije oba pristupa detaljno su objašnjeni. Također, pruža se i interaktivna aplikacija koja demonstrira ubrzanja kroz vizualizaciju.

Konačno rad se bavi analizom rezultata, određivanju optimalnih parametara za pojedine situacije i praktičnosti primjene u stvarnim simulacijama.

1. Problem kvadratne složenosti kod izračuna sudara

Glavni problem naivnog algoritma za izračun sudara je njegova složenost. Algoritam ima kvadratnu složenost zato jer je u naivnog algoritmu potrebno usporediti svaku česticu sa svakom drugom česticu.

```
DetekcijaSudara(čestice):
   N = broj čestica
   za i = 1 do N-1:
      za j = i+1 do N:
         riješi_sudar(čestice[i], čestice[j])
```

Slika 1.1 Pseudo kod za rješavanje sudara kod N čestica

Broj potrebnih izračuna dobiva se formulom 1.2

$$BROJ\ IZRAČUNA\ = \frac{N\ \cdot\ (N-1)}{2}$$

Formula 1.2 Broj izračuna kod osnovnog algoritma detekcije sudara

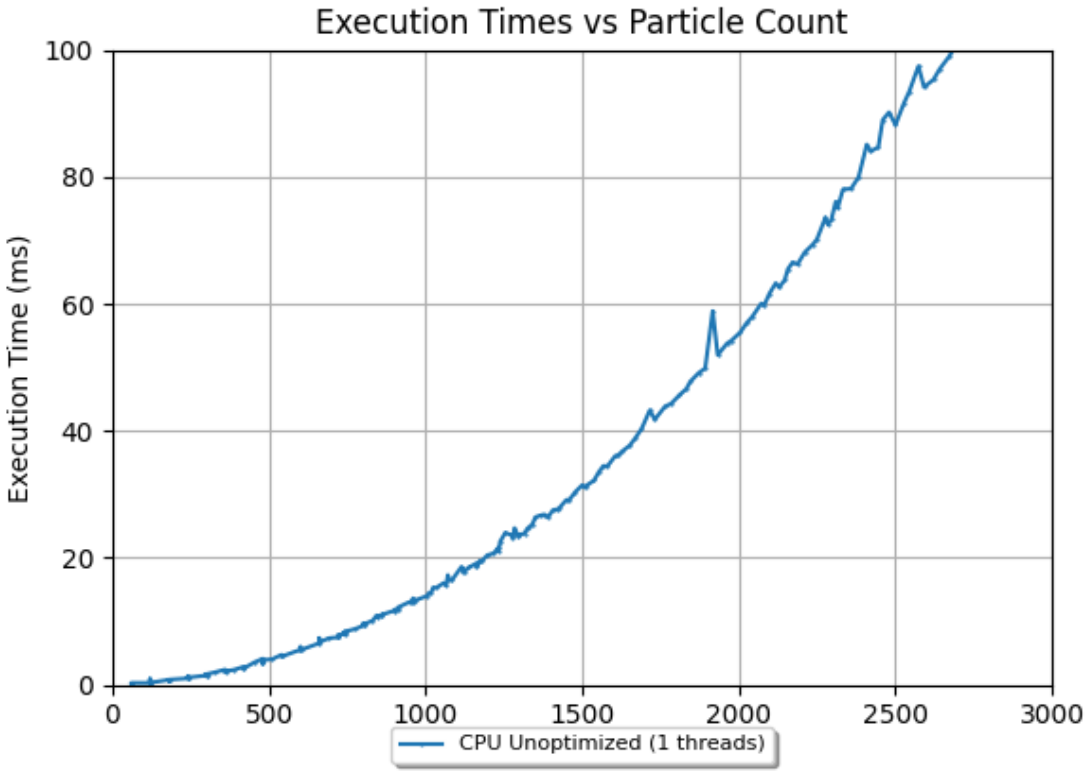
U O notaciji broj izračuna se zapisuje jednostavnije kao $O(N^2)$, što znači da algoritam raste s kvadratom broja čestica. Sustavi čestica koji se koriste za precizne simulacije često sadrže i više od stotine tisuća čestica, a kada se radi o tako velikim brojevima kvadratna složenost je prespora i za moderne procesore koji mogu raditi u redu veličine 10^9 operacija u sekundi.

Broj čestica	Broj izračuna osnovnim algoritmom	Okvirna frekvencija izračuna za rješavanje svih sudara u sekundi
1000	1,000,000	1 MHz
10,000	100,000,000	100 MHz
100,000	10,000,000,000	10 GHz
1,000,000	1,000,000,000,000	1 THz

Tablica 1.3 Okvirna ovisnost broja čestica o brzini procesora kod naivnog algoritma

U tablici 1.3 vidljivo je kako se broj izračuna sudara skalira s brojem čestica i kolika bi takt procesora bio potreban. Ta frekvencija je optimistična zato jer je za izračun jednog sudara potrebno više od jedne aritmetičke operacije i u stvarnosti želimo više od jednog izračuna u sekundi. No čak i ako se koristi optimistična procjena frekvencije vidljivo je da za velik broj

čestica problem nije moguće riješiti naivnim algoritmom na jednom samo procesoru u razumom vremenu.



Slika 1.4 Ovisnost vremena izvođenja naivnog algoritma o broju čestica

Slika 1.4 prikazuje koliko brzo vrijeme izvođenja raste s porastom broja čestica kada se naivan algoritam izvodi na stvarnom sklopovlju. Može se vidjeti da već pri par tisuća čestica simulacija više nije dovoljno brza za rad u stvarnom vremenu.

2. Paralelizacija algoritma

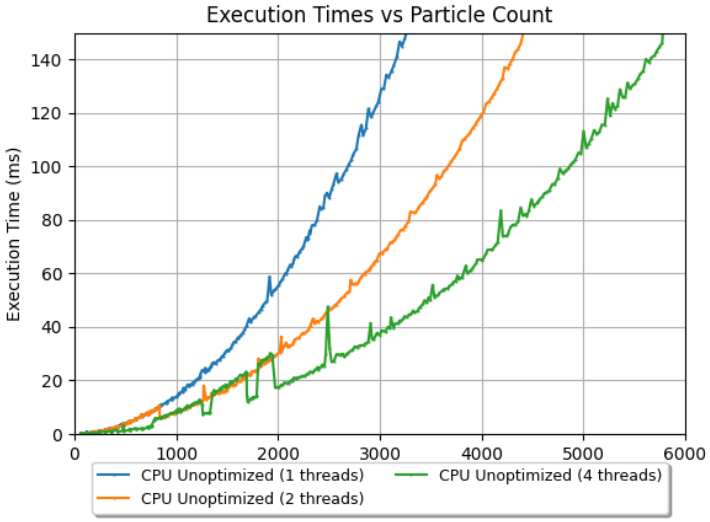
2.1 Paralelizacija naivnog algoritma na procesoru

Algoritam rješavanja sudara naivnim algoritmom moguće je jednostavno podijeliti na nezavisne poslove i time postići paralelizaciju i dobiti ubrazanje koje se okvirno skalira s brojem procesora.

```
DetekcijaSudaraParalelizirana(čestice):
    N = broj čestica
    M = broj dretvi
    paralelno za M:
      id = id drevete
      pocetni_indeks, zavrsni_indeks = izracunaj_indekse_dretvi(id, N, M)
      za i = pocetni_indeks do zavrsni_indeks:
        za j = i+1 do N:
          riješi_sudar(čestice[i], čestice[j])
```

Slika 2.1.1 Pseudo kod paraleliziranog naivnog algoritma

Na slici 2.1.1 može se vidjeti jedan od načina kako je moguće paralelizirati naivni algoritam na procesoru gdje se vanjska *za* petlja dijeli na M dretvi gdje svaka od njih preuzima istu količinu posla na predvidljiv način. Očekivano ubrzanje algoritma nakon ove paralelizacije proporcijalno je broju dretvi.



Slika 2.1.2 Usporedba vremena izvedbe algoritma ovisno o broju čestica

Na slici 2.1.2 prikazano je kako se eksperimantalno vrijeme izvođenja algoritma skalira s brojem čestica i ti podatci se jako dobro slažu s očekivanim linearnim skaliranjem ovisno o broju procesora. No to još uvijek nije dovoljno brzo jer kvadratna funkcija još uvijek raste prebrzo, a moderni korisnički procesori uglavnom imaju manje od 10-ak jezgara. To znači da je maksimalno ubrzanje na takvom procesoru oko deset puta. Očito će za neki puno veći broj čestica biti potrebno koristit neku drugu metodu osim skaliranja procesora.

2.2 Paralelizacija naivnog algoritma na grafičkoj kartici

Grafička kartica (GPU) je ključna komponenta u današnjim računalima koja je specijalizirana za obradu i prikazivanje grafičkih podataka, ali njezine prednosti nisu ograničene samo na grafiku. GPU-ovi su izvrsni za paralelno izvršavanje zadataka zbog velikog broja jezgri koje omogućuju istovremeno izvršavanje velikog broja jednostavnih instrukcija.

GPU je idealan za izvršavanje paralelnih algoritmima koje je moguće svesti na jednostavne instrukcije koje koriste osnovne tipove podataka i vektore. Naivni algoritam rješavanja sudara spada baš u takve algoritme zato jer koristi samo pozicije objekata koje je moguće zapisati kao vektor i jednostavne matematiče operacije i instrukcije.

```
Riješi sudar(id_čestica1, id_čestica2):
  pozicija1 = pozicije[id_čestica1]
  pozicija2 = pozicije[id_čestica2]
  radius1 = radiusi[id_čestica1]
  radius2 = radiusi[id_čestica2]

  udaljenost = abs(pozicija1 – pozicija2)
  ako je udaljenost < radius1 + radius2
    delta = radius1 + radius2 – udaljenost
    vektor_sudara = (pozicija2 – pozicija1) / udaljenost * delta
    pozicije[id_čestica1] -= vektor_sudara
```

Slika 2.2.1 Pseudo kod za rješavanje sudara između dvije čestica

Na slici 2.2.1 je prikaz algoritma za rješavanje sudara između dvije čestice. Može se vidjeti da je algoritam vrlo jednostavan i da osim instrukcija pisanja i čitanja iz memorije koristi samo jednostavne aritmetičke operacije koje su sve dostupne u kodu kojega grafička kartica može izvršavati. Tako da je kod algoritma moguće skoro i direktno prepisati u kod koji grafička kartica može izvršiti.

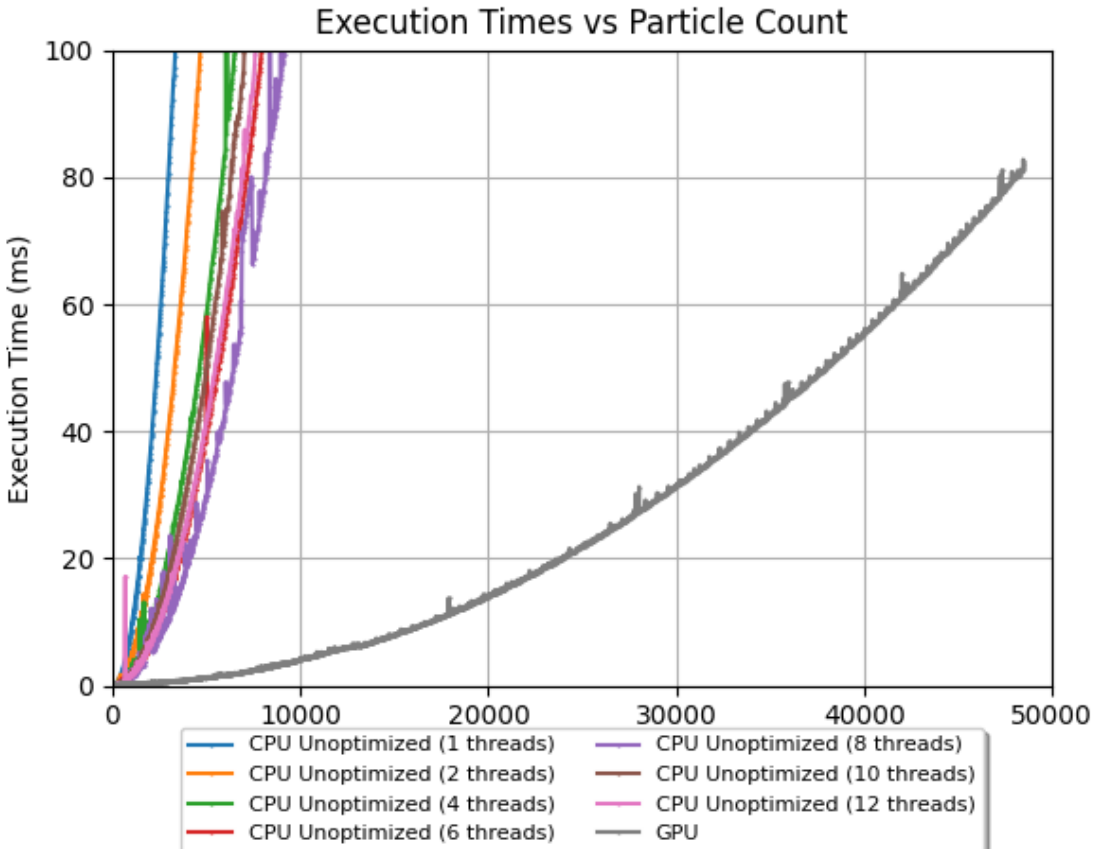
U ovom radu za pokretanje koda korištena je Nvidia grafička kartica i CUDA programski jezik koji ima poprilično sličnu sintaksu kao C++ programski jezik. Prije pokretanja samog koda algoritma na grafičkoj kartici potrebno je napisati *kernel* po pseudo kodu sa slike 2.2.2

```
MainCUDA(pozicije, radiusi, broj elemenata):
  inicijaliziraj memoriju(d_pozicije, veličina(pozicije))
  inicijaliziraj memoriju(d_radiusi, veličina(radiusi))
  inicijaliziraj memoriju(d_rez, veličina(pozicije))

  kopiraj memoriju s procesora(d_pozicije, pozicije)
  kopiraj memoriju s procesora(d_radisui, radiusi)

  dimenzije 3d bloka = 16, 16
  dimenzija 3d mreže = broj elemenata / 16 + 1, broj elemenata / 16 + 1
```

```
  Riješi Sudar(dimenzije bloka, dimenzije mreže, broj elemenata, d_pozicije, d_radiusi,
d_rez)

  sinkroniziraj sve dretve()
  kopiraj memoriju na procesor(pozicije, d_ rez)

  oslobodi memoriju(d_pozicije)
  oslobodi memoriju(d_radiusi)
  oslobodi memoriju(d_rez)


Riješi Sudar(dimenzije bloka, dimenzije mreže, broj elemenata, d_pozicije, d_radiusi, d_rez):
  i, j = izračunaj_indeks(dimenzije bloka, dimenzije mreže)
  ako je i < broj elemenata & j < broj elemenata & i != j
    Kod algoritma za rješavanje sudara()
```

Slika 2.2.2 Pseudo kod CUDA programa za izračunavanje sudara

Kod pisanja ulaznog *main* CUDA programa između ostalog potrebno je eksplicitno brinuti o alociranju memorije, kopiranju memorije s procesora i natrag i oslobađanju memorije. Kada je sve potrebno za funkciju kopirano iz glavne memorije u memoriju grafičke kartice potrebno je postaviti parametre paralelizacije. CUDA kod paralelizira koristeći blokove i mrežu. Blokovi su jedinice koje sve izvršavaju isti kod u isto vrijeme, a mreža je način na koji su ti blokovi organizirani. Ovdje se koriste 3d blokovi jer je potrebno ostvariti funkcionalnost dvostruke *za* petlje koja prolazi kroz N elemenata. Svaka dretva unutar bloka izvršava isti kod i svaka zna svoj indeks unutar bloka, svoj blok i svoje mjesto u mreži i na temelju toga može izračunati za koji element je zadužena i treba obraditi.



Slika 2.2.3 Usporedba performanci naivnog algoritma na CPU i GPU

Slika 2.2.3 prikazuje kako se vrijeme izvođenja naivnog algoritama ponaša na procesoru s više dretvi i vrijeme izvođenja tog istog algoritma na grafičkoj kartici. Detaljnija analiza rezultat performanci na stvarnom hardveru biti će napravljena u kasnijem poglavlju, no iz slike je očito da se postiže ubrzanje više od jednog reda veličine što znači da je algoritam pogodan za paralelizaciju na grafičkoj kartici. U ovom radu koriste se samo osnovne funkcije CUDA programskog jezika, tako da su sigurno moguće dodatne optimizacije za koje bi bilo potrebno dublje ući u CUDA programiranje.

3. Optimizacija algoritma za detekciju sudara

3.1 Implementacija algoritma

Glavni problem kad naivnog algoritma je što uvijek provjeravamo sve parove čestica iako jedna čestica ne može biti u sudaru s više od par čestica u isto vrijeme jer je udaljena više od pola radiusa do druge čestice. Ako bi za svaku česticu mogli znati točno s kojim drugim česticama ju je potrebno usporediti i ako bi taj broj bio konstantan tada algoritam teoretski postaje O(n). Zato jer za svaku česticu trebamo raditi samo konstantan broj usporedba koji ne ovisi o broju čestica.

Ne postoji algoritam koji bi nam mogao točno reći koje čestice je potrebno provjeriti, a da ne ovisi o broju čestica, ali postoje algoritmi koji eliminiraju većinu čestica i znatno ubrzavaju izvođenje algoritma. Glavni algoritam koji se koristi za to je separacija prostora. Najlakša separacija prostora, koja je ujedno implementirana u ovom radu je kvadratna mrežna separacija koja dijeli prostor na MxM kvadratnih pod prostora.

Kod podijele prostora na mrežu potrebno je izmijeniti algoritam da više ne provjerava svaku česticu sa svakom drugom nego samo čestice koje su u istim ili susjednim ćelijama, ako postavimo ograničenje da je svaka čestica uvijek manja od veličine ćelije. Prije provedbe algoritma potrebno je svaku česticu dodati u ćeliju u kojoj je sadržana. Nakon toga potrebno je iterirati po svim ćelijama i za svaku ćeliju i njene susjede provoditi naivni algoritam.

```
DetekcijaSudaraOptimizirana(čestice):
  N = broj čestica
  M = veličina mreže
  PostaviČesticeUĆelije(čestice)
  za sve ćelije
     za čestica1 unutar trenutne ćelije
       za sve susjedne ćelije i trenutnu ćeliju
         za čestica2 unutar druge ćelije
           riješi_sudar(čestice1, čestice2)
```

Slika 3.1.1 Pseudo kod optimiziranog algoritma za rješavanje sudara

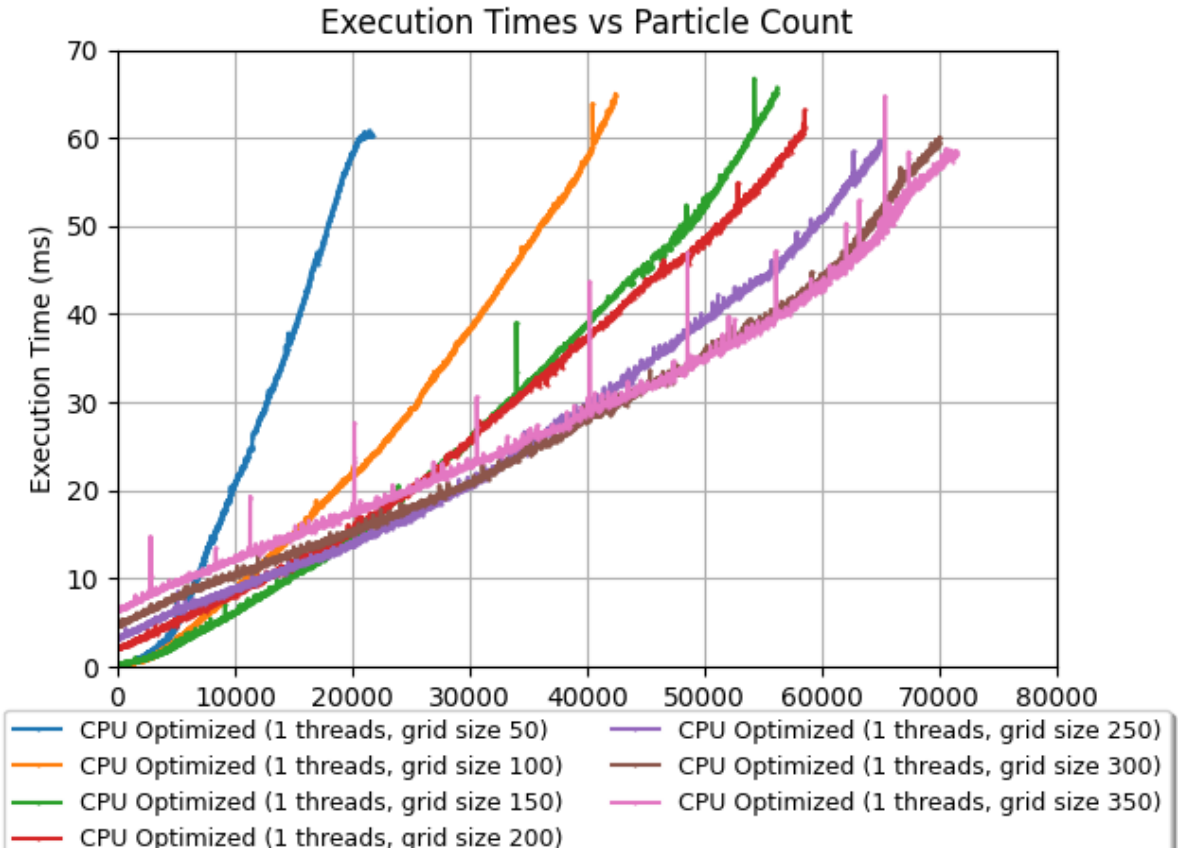
```
PostaviČesticeUĆelije(čestice):
  za čestica u čestice
    kordinate = dohvatiKordinate(čestica)
    id ćelije = izračunajIdĆelije(kordinate)
    ćelije[id ćelije] = čestica
```

Slika 3.1.2 Pseudo kod postavljanja čestica u pripadajuće ćelije

Na slici 3.1.1 prikazan je pseudo kod za jednu od mogućih implementacija strukture mreže koji je kompliciraniji od naivnog algoritma zbog korištenja četiri *za* petlje, umjesto dvije. Optimiziranim algoritmom više nije potrebno iterirati po svim česticama u dvostrukoj *za* petlji, nego po svim ćelijama i onda svim česticama unutar te ćelije čime se osigurava da algoritam neće provjeravati čestice koje su udaljene za više od jedne ćelije. To eliminira velik broj provjera jer kada se radi o mreži s većim dimenzijama većina čestica neće biti blizu trenutne ćelije koja se provjerava.

3.2 Utjecaj parametara optimizacije na performance

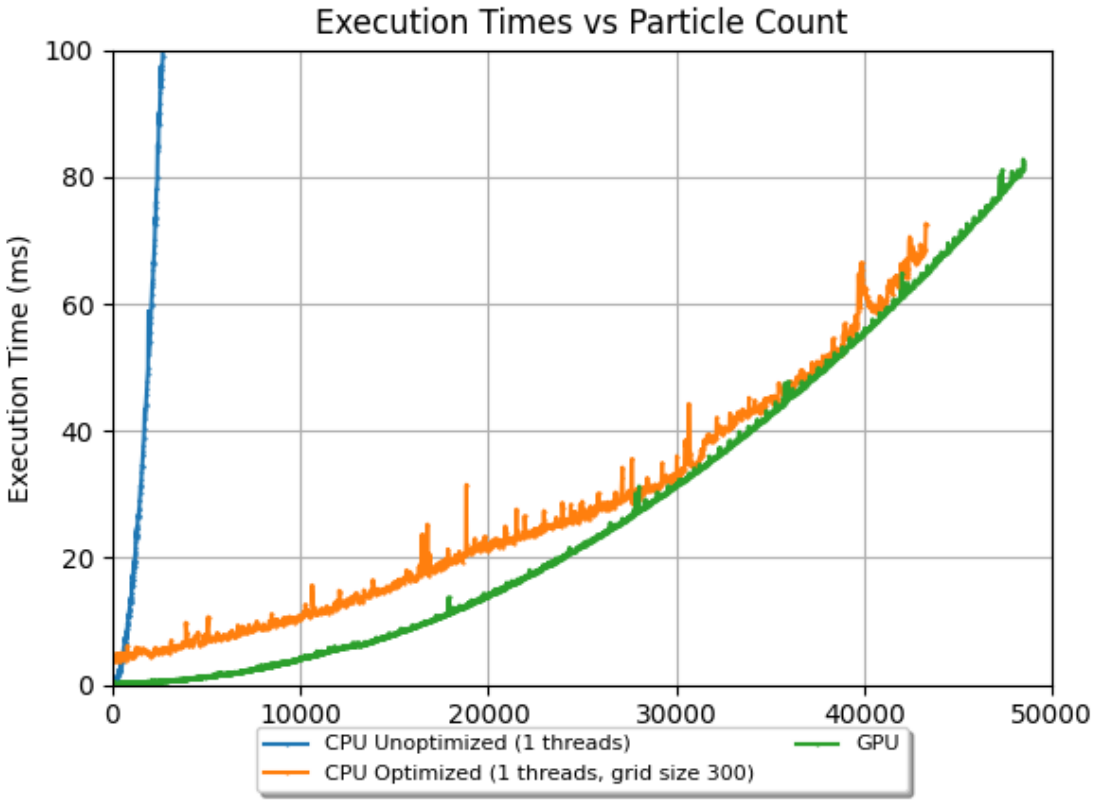
Kod ovog algoritma broj usporedbi značajno ovisi o parametrima simulacije, kao što su veličina mreže i veličina čestica. Na primjer kada je veličina čestica znatno manja od veličine ćelija za svaku ćeliju će biti potrebno raditi više usporedbi, a kada je veličina čestica veća od veličine ćelije tada će neke ćelije biti potpuno sadržane unutar neke čestice i bespotrebno vrijeme će biti izgubljeno na provjeru tih ćelija.



Slika 3.2.1 Usporedba različitih veličina mreža kod optimiranog algoritma

Slika 3.2.1 pokazuje kako promjena parametara veličine mreže, to jest broj ćelija utječe na performance koje je moguće dobiti optimizacijom. Broj ćelija je obrnuto proporcionalan s veličinom ćelija, tako da kada imamo pre malo ćelija i njihova veličina bude puno veća od veličine samih čestica dobivene performance neće biti toliko značajne. Može se vidjeti da sva testiranja rađena s mrežom manjom od 150 ćelija rezultiraju boljima, ali ne dovoljno dobrim brzinama simulacije. Eksperimentalnim putem pokazalo se da je optimalna veličina mreže oko 300x300 ćelija, dok bi veličina čestice trebala biti otprilike iste veličine kao i ćelija.

3.3 Performance optimiranog algoritma na jednoj dretvi



Slika 3.3.1 Usporedba naivnog algoritma na CPU i GPU i optimiranog algoritma na CPU

Slika 3.3.1 prikazuje usporedbu naivnog algoritma i optimiranog algoritma detekcije sudara i iz rezultata mjerenja vidljivo koliko ubrzanje se dobiva kada primijenimo optimizaciju u kojoj se koristi struktura podataka mreže i načelo da je potrebno provjeriti samo čestice u susjednim ćelijama. Optimizacijom algoritma na procesoru dostignute su performance naivnog algoritma na grafičkoj kartici sa samo jednom dretvom što znači da je algoritam nekoliko redova veličine brži.

3.4 Paralelizacija optimiranog algoritma

Algoritam je optimiran s paralelizmom na umu tako da je način na koji je optimalan za izvođenje na više dretvi. S obzirom da smo prostor podijelili na više jednakih dijelova paralelizacija je vrlo jednostavna i sve što treba napraviti je svakoj dretvi dati drugu ćeliju. Svaka dretva u tom slučaju posao može obavljati potpuno neovisno o drugim dretvama.

```
DetekcijaSudaraOptimiziranaIParalelizirana(čestice):
  N = broj čestica
```

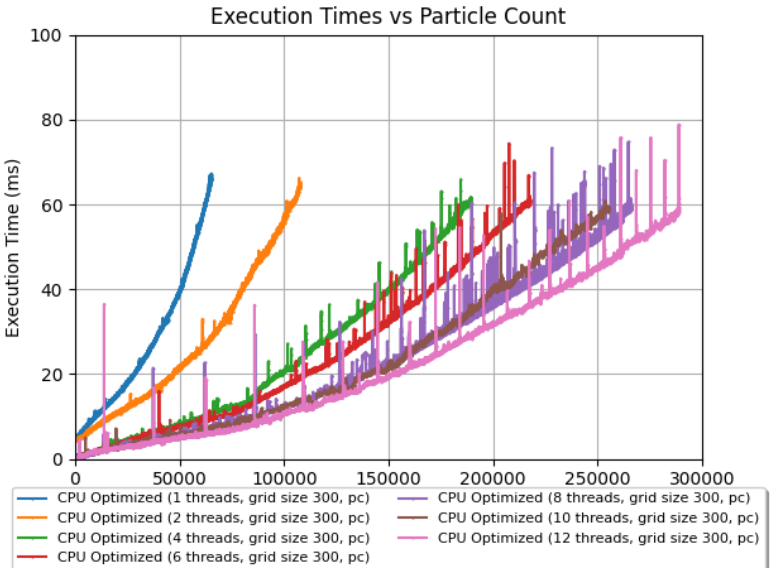
```
M = veličina mreže
PostaviČesticeUĆelijeParalelno(čestice)
pralelno za sve ćelije
    za čestica1 unutar trenutne ćelije
      za sve susjedne ćelije i trenutnu ćeliju
        za čestica2 unutar druge ćelije
          riješi_sudar(čestice1, čestice2)
```

Slika 3.4.1 Pseudo kod paraleliziranog optimiranog algoritma

```
PostaviČesticeUĆelijeParalelno(čestice):
  paralelno za čestica u čestice
    kordinate = dohvatiKordinate(čestica)
    id ćelije = izračunajIdĆelije(kordinate)
    ćelije[id ćelije] = čestica
```

Slika 3.4.2 Pseudo paralelizirane funkcije za postavljanje čestica u ćelije

Na slikama 3.4.1 i 3.4.2 prikazan je pseudo kod s kojim je postignuta paralelizacija optimiranog algoritma i jedino što je potrebno je promijeniti dvije linije koda. U funkciji na slici 3.4.2 jedino što treba napraviti je podijeliti za petlju koja ide po svim česticama tako da svaka uzme jednak dio posla i zasebno ih postavlja u ćeliju u koju pripadaju. Druga stvar koju je potrebno napraviti da se postigne paralelizacija je podijeliti vanjsku za petlju sa slike 3.4.1 po dretvama.



Slika 3.4.3 Optimizarani algoritam na procesoru

Na slici 3.4.3 može se vidjeti utjecaj paralelizacije na izvođenje optimiranog algoritma. Dobiva se odlično skaliranje s brojem dretvi kao što je bilo očekivano zbog implementacije algoritma.

4. Paralelizacija optimiranog algoritma na grafičkoj kartici

4.1 Ideja

Algoritam koji se koristi za simulaciju sudara objekata u mreži prilagođen je za izvršavanje na grafičkoj kartici (GPU) kako bi se povećala efikasnost i smanjio broj razmjena podataka između procesora (CPU) i grafičke kartice. Ova prilagodba uključuje optimizaciju postojećeg algoritma pisanog za procesor te uvođenje paralelizacije u ključnim dijelovima izvođenja.

4.1.1 Pojednostavljenje algoritma za grafičku karticu

Osnovni algoritam koji je ranije bio dizajniran za procesorsku obradu pojednostavljen je tako da koristi osnovne strukture podataka koje su optimalne za izvršavanje na GPU-u. Umjesto složenih dinamičkih struktura, koristi se jednostavnija reprezentacija podataka kako bi se omogućila brža i efikasnija obrada velikog broja objekata.

4.1.2 Paralelizacija postavljanja objekata u mrežu

Jedna od glavnih optimizacija uključuje paralelizaciju procesa postavljanja objekata u ćelije mreže. Na CPU-u je taj zadatak obavljala jedna dretva, što je bilo ograničenje u pogledu brzine i skalabilnosti. Taj dio je radio dovoljno brzo na procesoru, ali nije ga moguće direktno iskoristiti da pripremi mrežu za GPU jer bi to rezultiralo prevelikom broju razmjene podataka Između CPU i GPU, a to bi pak drastično smanjilo performance. Na GPU-u, ovaj zadatak se paralelizira tako da svaki objekt može biti obrađen istovremeno od strane zasebne dretve.

4.1.3 Rješavanje sudara objekata

Jednom kada su objekti postavljeni u ćelije mreže, faza rješavanja sudara ostaje gotovo identična verziji koja se izvršavala na CPU-u. Objekti unutar iste ćelije provjeravaju međusobne sudare, pri čemu se koriste iste formule i pristupi kao u originalnom algoritmu. Međutim, zahvaljujući većem broju dretvi na GPU, puno više sudara se može rješavati u isto vrijeme, što dodatno povećava efikasnost izvođenja.

4.2 Implementacija

4.2.1 Strukture podataka

Svako svojstvo čestice, kao njezina pozicija na X ili Y osi ili pak njena brzina ili akceleracija trebaju biti zasebno polje fiksne duljine. Zato je potrebno prije prevođenja zadati najveći broj čestica koje simulacija podržava.

```
const static int maxNumberOfGameObjects = 2000001;

float gameObjectsXPositions[maxNumberOfGameObjects];
float gameObjectsYPositions[maxNumberOfGameObjects];
float gameObjectsRadius[maxNumberOfGameObjects];
float gameObjectsXLastPosition[maxNumberOfGameObjects];
float gameObjectsYLastPosition[maxNumberOfGameObjects];
float gameObjectsXAcceleration[maxNumberOfGameObjects];
float gameObjectsYAcceleration[maxNumberOfGameObjects];
```

4.2.1 Polja za svojstva čestica u simulaciji

Glavna struktura podataka za algoritma separacije prostora je mreža. Ona je u ovoj implementaciji napravljena kao jedno polje veličine *M * M * najveći broj čestica u ćeliji*. Gdje je *M* broj ćelije u jednom redu ili stupcu, a najveći broj čestica u ćeliji je potrebno postaviti prije prevođenja.

```
const static int maxCellSize = 25;
const static int gridSize = 300;
int grid[maxCellSize * gridSize * gridSize];
```

4.2.2 Polje za mrežu

4.2.2 Postavljanje elemenata u ćelije

Postavljanje elemenata u ćelije je napravljeno tako da je za svaku česticu zadužena jedna dretva koja onda treba dodati redni broj elementa u pripadajuću ćeliju. Ovdje je jako bitno da ne postoji mogućnost da dvije dretva probaju dodati isti element na istu poziciju. To se može dogoditi jer svaka dretva treba prvo provjeriti nalazili se već neki objekt na poziciju na koju ga ona želi dodati. Pa ako dođe do situacije da više dretvi provjeri neku poziciju i zaključe da je taj indeks slobodan sve će u isto vrijeme probati dodati taj element i to će dovesti do neželjenog ponašanja algoritma.

Taj problem je riješen korištenjem CUDA naredbe *atomicCAS* koja sinkronizira dretva i osigurava da jedna dretve jedna po jedna obavljaju čitanje i pisanje na neku memorijsku lokaciju. To se može vidjeti na slici 4.2.3.

```
__global__ void positionObjectsInGrid(float *xPos, float *yPos, int *grid, int sizeOfGrid, int maxCellSize, int numElements) {

    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i >= numElements) {
        return;
    }

    const int xGrid = floor( x: (xPos[i] + 1.0) * 0.5 * sizeOfGrid);
    const int yGrid = floor( x: (yPos[i] + 1.0) * 0.5 * sizeOfGrid);
    const int cell = yGrid * sizeOfGrid + xGrid;

    int iterations = 0;
    for (int l = 0; l < maxCellSize; l++) {
        iterations++;

        if (atomicCAS( address: grid +cell * maxCellSize + l,  compare: -1,  val: i) == -1) {
            break;
        }
    }
    if (iterations == maxCellSize) {
        printf( format: "Max cell size reached in cell %d when adding object %d\n", cell, i);
    }

}
```

4.2.3 Funckija za postavljanje elemenata u ćelije

4.2.3 Rješavanje sudara

Rješavanja sudara se radi prilično slično kao i na procesoru. Jedino se koristi malo drugačije logika traženja susjednih ćelija i iteriranja po ćelijama i elementima zbog jednostavnije strukture podataka. Svaka dretva je zadužena za jednu ćeliju i za koju prvo treba naći susjedne ćelije, a nakon za svaku od susjednih ćelija riješiti sudare između objekata u dretvinoj ćeliji i susjednoj ćeliji. Sama implementacija se može vidjeti na slici 4.2.4

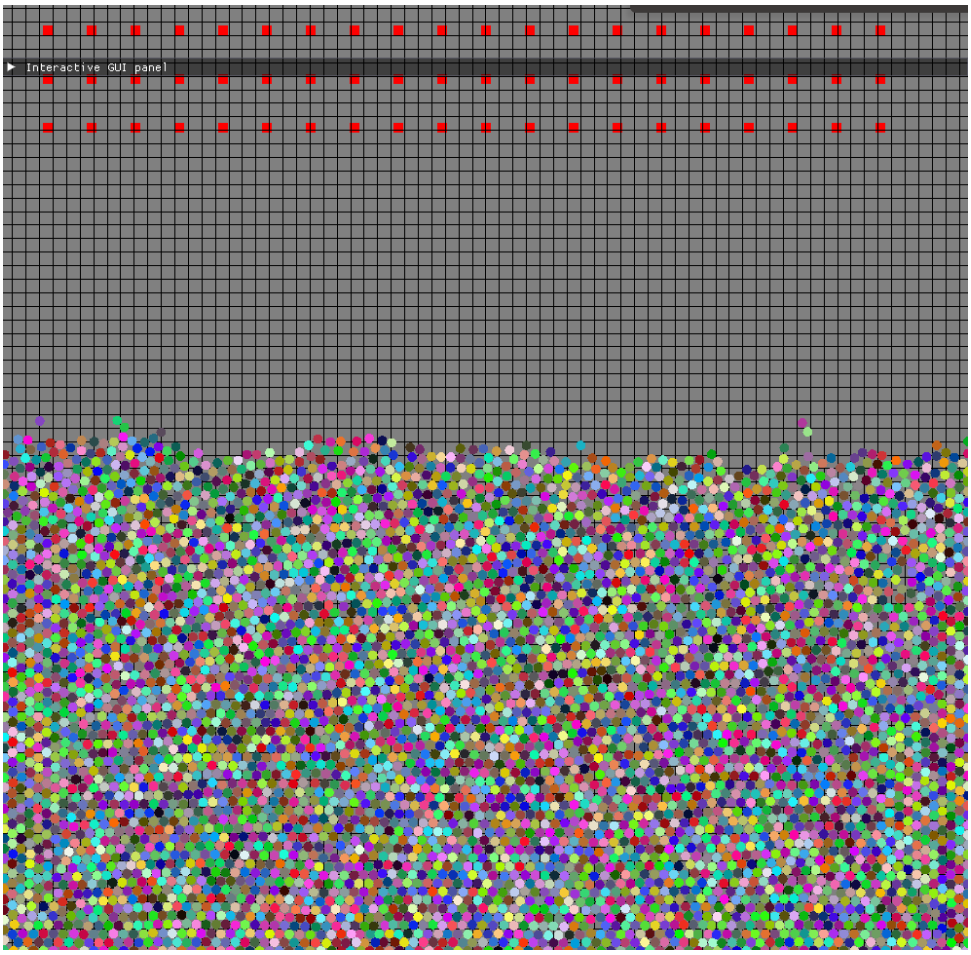
```
__global__ void solveContactGrid(float *xPos, float *yPos, float *xPosRes, float *yPosRes, float *radii, int *grid, int sizeOfGrid, int maxCellSize, int numElements) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (numElements == 0) {
        return;
    }
    if (i < sizeOfGrid * sizeOfGrid){

        // nadi x i y koordinate ćelije
        int x = i % sizeOfGrid;
        int y = i / sizeOfGrid;

        // iteriraj po svim susjednim ćelijama
        for (int iAdj = -1; iAdj <= 1; iAdj++) {
            for (int jAdj = -1; jAdj <= 1; jAdj++) {

                int x_adj = x + iAdj;
                int y_adj = y + jAdj;

                // ako je ćelija unutar granica simulacije
                if (x_adj >= 0 && x_adj < sizeOfGrid && y_adj >= 0 && y_adj < sizeOfGrid) {
                    int cell1 = i;
                    int cell2 = y_adj * sizeOfGrid + x_adj;

                    // iteriraj po svim objektima u prvoj ćeliji
                    for (int iCell = 0; iCell < maxCellSize; iCell++) {
                        if (grid[cell1 * maxCellSize + iCell] == -1) {
                            break;
                        }
                        // iteriraj po svim objektima u prvoj ćeliji
                        for (int jCell = 0; jCell < maxCellSize; jCell++) {
                            if (grid[cell2 * maxCellSize + jCell] == -1) {
                                break;
                            }
                            int obj1_idx = grid[cell1 * maxCellSize + iCell];
                            int obj2_idx = grid[cell2 * maxCellSize + jCell];

                            // riješi sudar između sva objekta
```

Slika 4.2.4 Implementacija rješavanja sudara pomoću mreže

5. Interaktivna simulacija

5.1 Simulacija

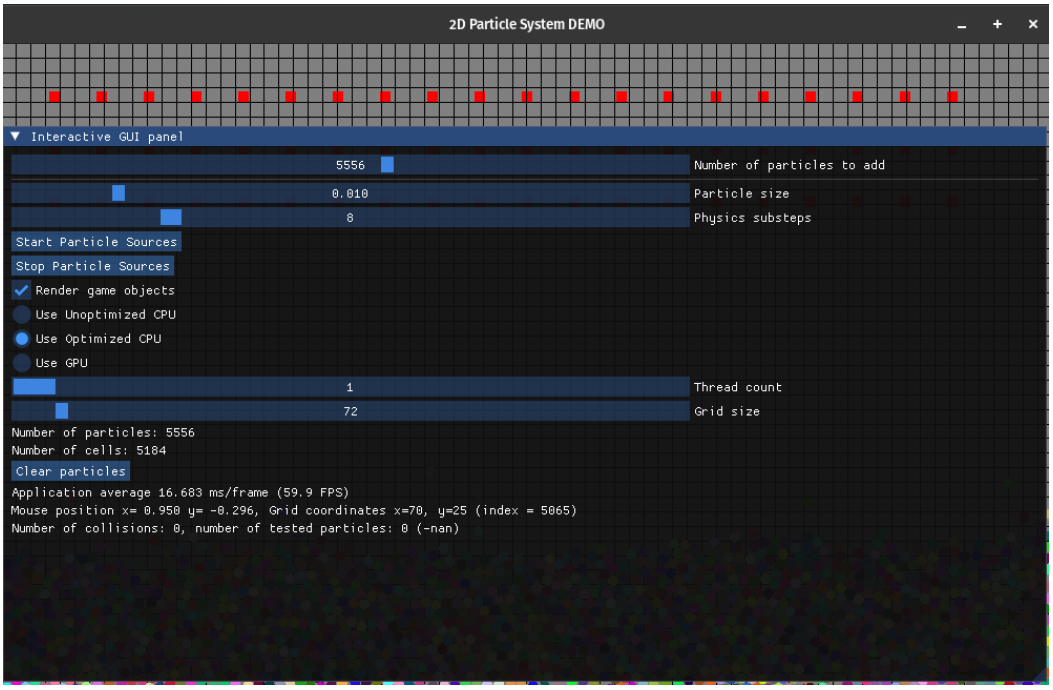
Simulacija prikazuje čestice u obliku krugova na koje djeluje gravitacija. Da bi se postigla veća razina stabilnosti simulacije postavljeno je ograničenje za maksimalnu brzinu čestica. To je učinjeno s ciljem povećanja stabilnosti simulacije koja je potrebna da bi mjerenja bila smislena. Zato je prilikom izvođenja simulacije moguće vidjeti da čestice izgube brzinu kada se međusobno sudare, no to ne utječe na rezultate brzine izvođenja algoritma. Isto je bitno napomenuti da prilikom svakog prikaza okvira (engl. frame) osam puta provodimo izračune sudara isto s ciljem povećanja stabilnosti simulacije.



Slika 5.1.1 Prikaz simulacije

5.2 Interaktivno grafičko sučelje

Za potrebe ovog rada izrađena je simulacija s interaktivnim grafičkim sučeljem da bi se omogućilo lakše testiranje i prikaz rezultata. Mogućnosti koje grafičko sučelje nudi su postavljanje broja čestica koje će biti dodane u simulaciju, veličine čestica i broj koraka izračuna fizike.



Slika 5.2.1 Prikaz interaktivne simulacije

Slika 4.2.1. prikazuje grafičko sučelje simulacije u kojem se mogu podešavati razni parametri simulacije i kontrolirati samu simulaciju. Simulacija podržava tri opcije po izvođenje algoritma za rješavanje sudara. Naivni (ne optimiranog) algoritam koji će se izvoditi na procesoru, isti taj algoritam koji se izvodu na grafičkoj kartici i optimirani algoritam koji se izvodi na procesoru.

Isto tako je moguće promijeniti algoritam prilikom izvođenja simulacije da razlika u performancama može vidjeti direktno na stvarnom primjeru. Dodatno je još moguće podesiti broj dretvi na procesoru pomoću kojih će se raditi izračuni sudara kao i veličina mreže u slučaju da se radi o optimiranom algoritmu koji koristi mrežu. Moguće je i zaustaviti prikazivanje čestica u simulaciji da bi se povećala brzina izvođenja sveukupne simulacije kada se radi o jako velikom broju čestica.

6. Analiza rezultata

6.1 Specifikacije sklopovlja

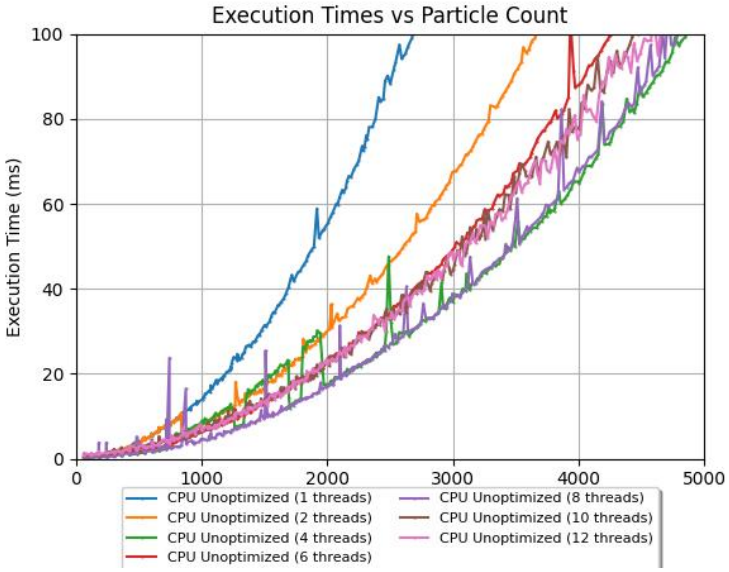
Mjerenja su rađena na dva različito konfigurirana računala. Radi se o osobnom prijenosnom računalu i stolnom računalu od kojih stolno računalo ima brže sklopovlje.

Računalo	Model procesora	Broj jezgri procesora/ broj dretvi	Standardna brzina procesora	Maksimalna brzina procesora	Grafička kartica	Broj jezgri grafičke kartice
Prijenosno računalo	Intel® Core™ i7-1165G7	4/8	2.80 GHz	4.70 GHz	-	-
Stolno računalo	AMD Ryzen 7 7700X BOX	8/16	4.5 GHz	5.40 GHz	NVIDIA GeForce RTX 4070	5888

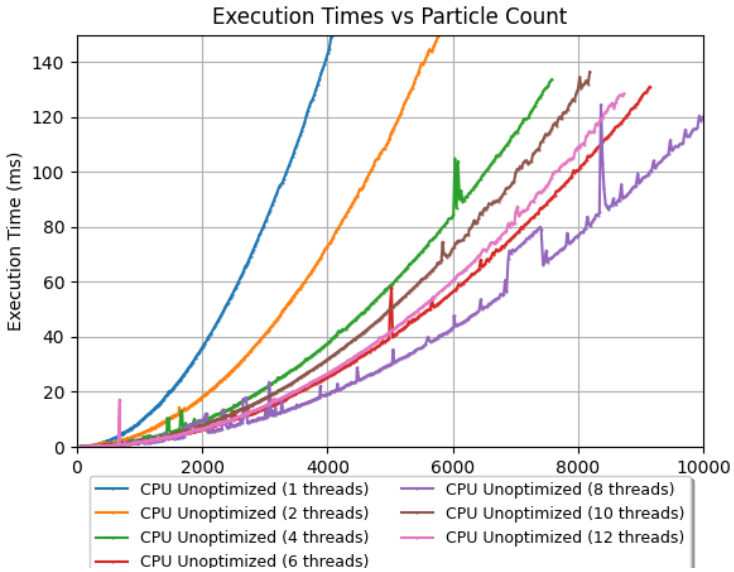
Tablica 6.1.1 Specifikacije sklopovlja

6.2 Rezultati performanci naivnog algoritma

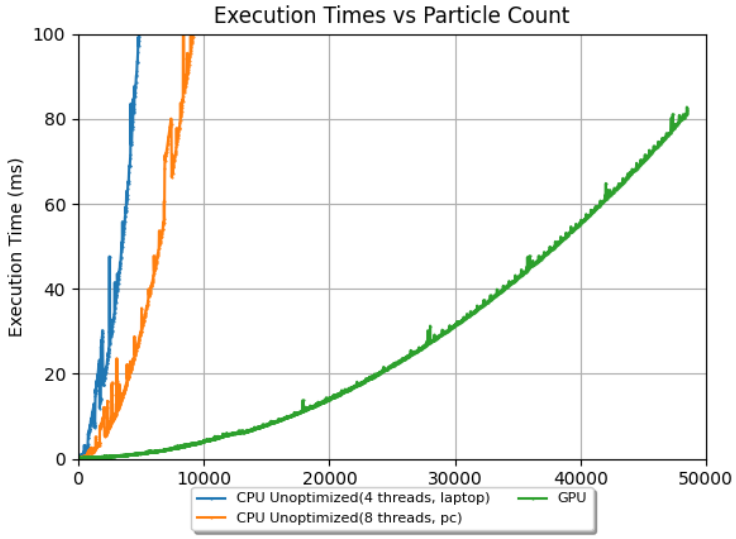
Na slikama 5.2.1 i 5.2.2 mogu se vidjeti kako vrijeme izvođenja algoritma ovisi o broju dretvi. U oba najbolje performance se dobivaju kada se za broj dretvi koristi fizički broj jezgri na procesoru. Prijenosno računalo koje je prikazano na slici 6.2.1 ima 4 fizičke jezgre i najbolje performance se dobivaju kada se koriste 4 dretve. Prijenosno računalo ima 8 fizičkih dretvi i vidljivo je da se najbolje performane i tamo dobivaju kada se uzme 8 dretvi. Za ovu primjenu procesor na stalnom računalu je otprilike dvostruko brži u odnosu na procesor u prijenosnom računalu. Broj čestica koje je moguće simulirati naivnim algoritmom u najboljem slučaju kada se koristi 8 dretvi na stolnom računalu je 4000 ako se želi postići mogućnost simulacije u stvarnom vremenu.



Slika 6.2.1 Naivni algoritam na procesoru prijenosnog računala pri različitom broju dretvi



Slika 6.2.2 Naivni algoritam na procesoru stolnog računala pri različitom broju dretvi

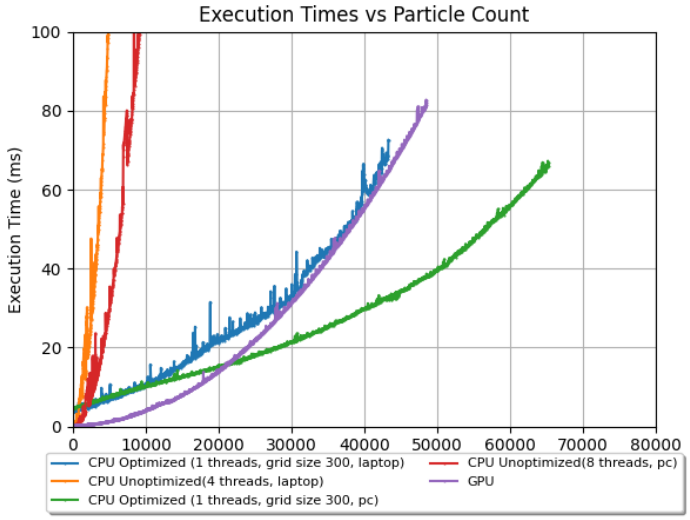


Slika 6.2.3 Usporedba izvođenja naivnog algoritma na grafičkoj kartici i procesoru

Slike 5.2.3 prikazuje brzinu izvođenja simulacije kada se naivni algoritam izvodi na grafičkoj kartici i odmah je vidljivo da se vrijeme izvođenja znatno smanjilo. Za naivni algoritam grafička kartica je skoro 30 puta brža od procesora zato jer ima znatno više fizičkih dretvi za izvođenja algoritma. Koristeći grafičku karticu naivinim algoritmom moguće je računati sudare za oko 25 tisuća čestica što je značajno ubrzanje s obzirom da se radi o algoritmu koji ima kvadratnu složenost. No još uvijek postoje razne upotrebe gdje to nije dovoljan broj čestica koje je moguće simulirati i sljedeći korak je optimizacija samog algoritma, a ne sklopovlja na kojem se izvodi.

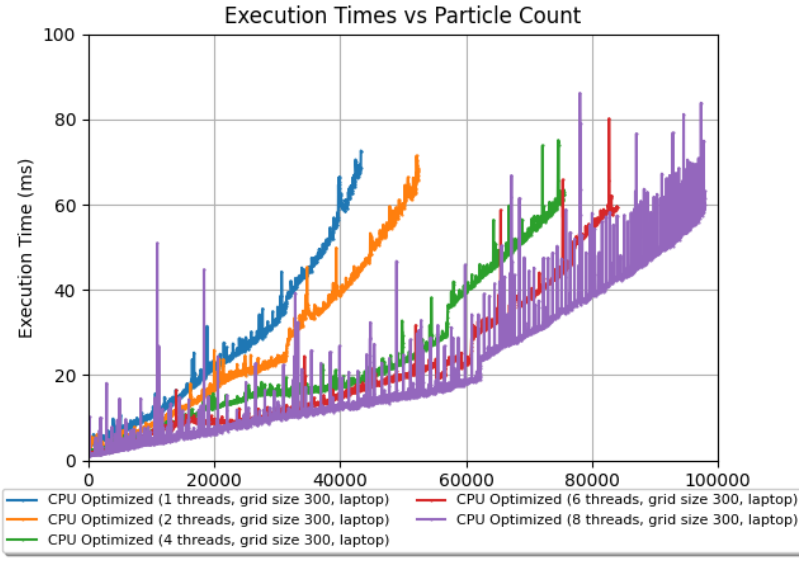
6.3 Rezultati performanci optimiranog algoritma

Nakon optimizacije algoritma provedeni su isti testovi na procesorima kao i kod naivnog algoritma. Kao što je bilo za očekivati optimirani algoritam na samo jednom dretvi je značajno brži od naivnog algoritma, čak i kada se radi o naivnog algoritmu koji se izvodi na grafičkoj kartici. Optimirani algoritam na prijenosnom računalu postiže otprilike istu brzinu kao i naivni algoritam na grafičkoj kartici, no kada se radi o bržem procesoru na stolnom računalu optimirani algoritam je znatno brži. To se može vidjeti na slici 6.3.1 koja uspoređuje ne optimirani algoritam s optimiranim.

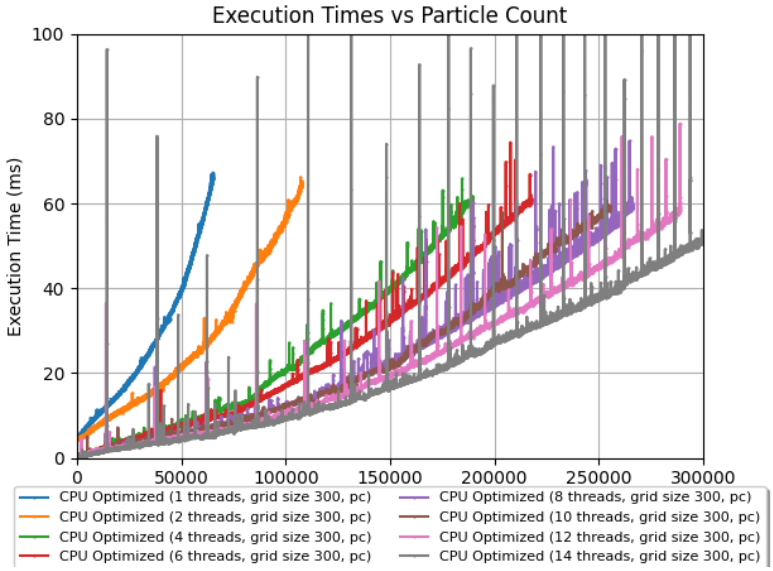


Slika 6.3.1 Usporedba naivnog algoritma s optimiranim algoritmom

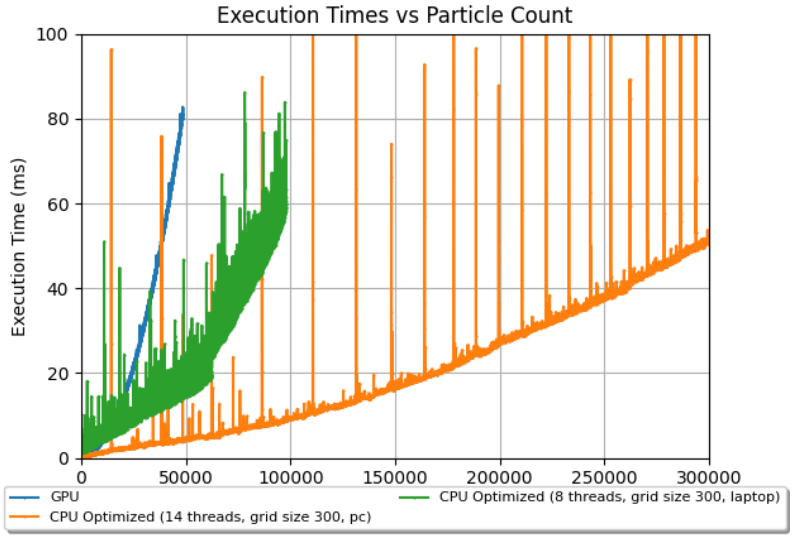
Pošto je optimizirani algoritam moguće jednostavno paralelizirati očekujemo velika ubrzanja pri korištenju više od jedne dretve. Na slici 5.3.2 može se vidjeti performance optimiranog algoritma na prijenosnom računalu kada se koriste više dretvi, a na slici 5.3.3 može se vidjeti isto to samo kada se radi o prijesnom računalu. Ovdje se vidi da je broj čestica koje je moguće simulirati u stvarnom vremenu već u stotinama tisuća što je prihvatljivo za neku ozbiljniju simulaciju. Iz slika je isto tako vidljivo da se performance jako dobro skaliraju s porastom broja dretvi. Optimirani algoritam je jedino sporiji u slučajevima kada ima jako malo čestica za koje je potrebno riješiti sudar zato jer on još uvijek treba proći kroz sve ćelije kojih može biti jako puno naspram malog broja čestica. To se može vidjeti na grafovima skroz na početku kada se radi o broju čestica manjem od otprilike 2000.



Slika 6.3.2 optimirani algoritam na procesoru prijenosnog računala pri različitom broju dretvi



Slika 6.3.3 Optimirani algoritam na procesoru prijenosnog računala pri različitom broju dretvi

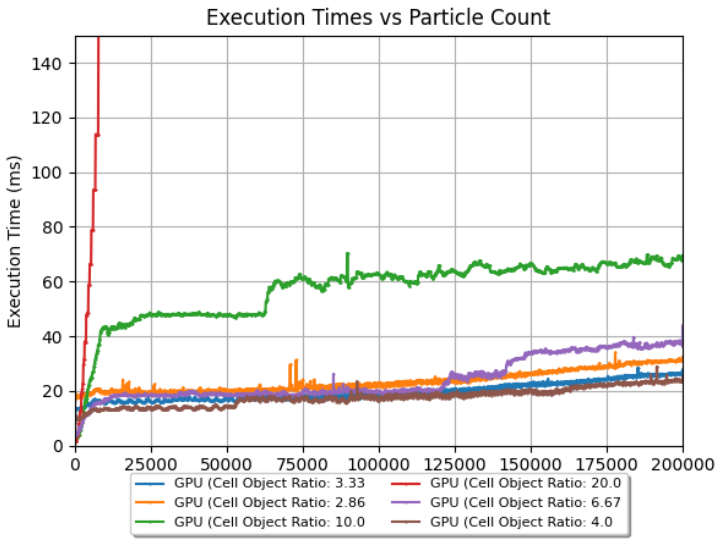


Slika 6.3.4 Usporedba izvođenja naivnog algoritma na grafičkoj kartici i optimiranog na procesoru

Na slici 5.3.4 može se vidjeti kako se performance naivnog algoritma na grafičkoj kartici uspoređuju s performancama optimiranog algoritma na procesoru kada se koristi optimalan broj dretvi za pojedini procesor. Grafička kartica je sporija i od procesora na prijenosnom računalu koji je i jeftiniji i troši manje energije za nje. Kada se radi o procesoru na stolnom računalu grafička kartica je za nekoliko redova veličina sporija, a isto se radi o jeftinijoj komponenti koja troši manje energije.

6.4 Rezultati optimiranog algoritma na grafičkoj kartici

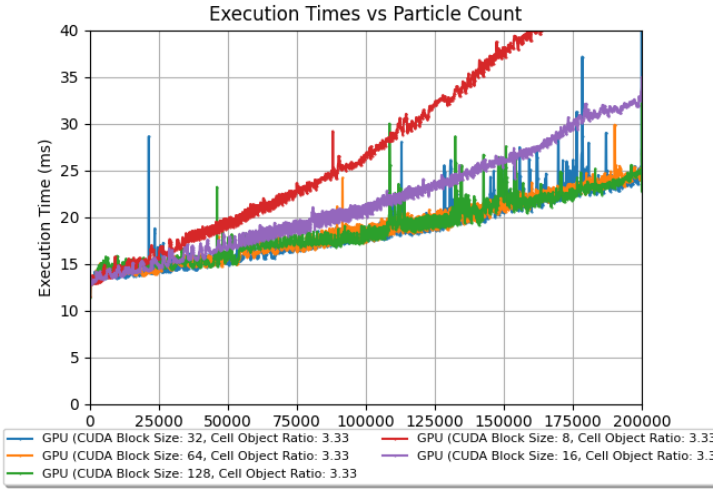
Jedan od parametara koji značajno utječe na performance je omjer veličine jedne ćelije s veličinom jedne čestice. Na slici 6.4.1 može se vidjeti kako taj omjer utječe na performance algoritma na grafičkoj kartici.



6.4.1 Ovisnost performaci o omjeru veličine ćelije i veličine čestice

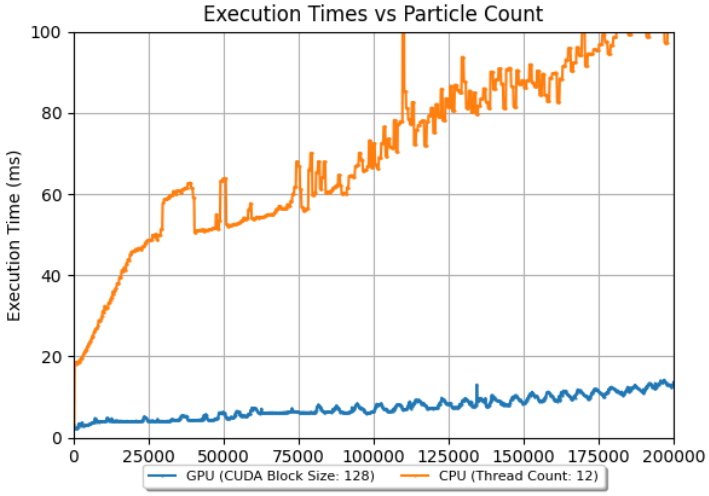
Kao što se može vidjeti na slici 6.4.1 kada je omjer veličine ćelije i čestice prevelik, znači da u jednu ćeliju može stati velik broj čestica, dolazi do znatnog pada peformanci. To je zato jer se prednosti mreže ne iskorištavaju u potpunosti pa algoritma još uvijek mora napraviti velik broj usporedbi čestica između dvije ćelije koji raste kvadratno u ovisnosti s brojem čestica. U ovoj simulaciji omjeri koji su manji od 5:1 imaju slične performance, ali najbolji omjer u ovom testu je bio omjer 4:1

Performance implementacije ovog algoritma na grafičkoj kartici ovise o još jednom važnom parametru, a to je to broj dretvi unutar jednog bloka grafičke kartice. Dretva na grafičkoj kartici podjeljene su u blokove gdje svaki blok izvršava istu operaciju samo s drugačijim podatcima.



6.4.2 Ovisnost performanci o veličine bloka

Na slici 6.4.2 vidi se da premala veličina bloka, kao naprimjer 8, dovode do manjih performci. Kada se taj parametar poveća na barem 32 performance su dovoljno slične. To je zato jer u ovom slučaju sve dretve trebaju raditi identičan posao i nema razloga dretva razdjeljivati u manje blokove.



6.4.3 Usporedba performanci procesora i grafičke kartice na istoj simulaciji

Slika 6.4.3 uspoređuje kako se mijenja vrijeme izračuna sudara kad se za to koristi procesor i kada se za to koristi grafička kartica. Jasno se vidi da je grafička kartica znatno brža i vrijeme izračuna znatno sporije raste, što je i bila pretpostavka ovoga rada. Testiranja nisu provedena za više od 200 000 čestica jer je to bio maksimum čestica koji stanu u simulaciju. No iz grafa možemo prepostaviti da bi prefornce grafičke kartice nastavile rasti sličnim tempom i da bi se lagano moglo simulirati znatno više čestica u razumnom vremenu.

7. Daljnji razvoj

Ovaj rad je za cilj ima poboljšati performance naivnog algoritma za detekciju sudara što je i postignuto do određene mjere. No moguće je još puno više ubrzati taj algoritam ili naći neku primjenu gdje bi optimirani algoritam bio potreban.

7.1 Daljnja optimizacija algoritma

U ovom radu je isprobana samo najjednostavnija moguće razdioba prostora, a to je na kvadratnu mrežu. No tu naravno postoje očite ne efikasnosti kao što je postajanje ćelija koje su prazne jer ako je ćelija prazna na nju samo trošimo resurse bez ikakve potrebe. Razdiobe prostora kao što su binarna razdioba ili neke kompleksnije bi mogle dovesti do boljih performanci u algoritmu detekcije sudara.

7.2 Primjena detekcije sudara u fizički točnoj simulaciji

Algoritam iz ovoga rada kojim je moguće simulirati stotine tisuća čestica u stvarnom vremenu je moguće iskoristiti za neku fizičku simulaciju gdje je ustvari potreban tako velik broj čestica. Trenutna simulacija u ovom radu simulira neku osnovnu interakciju čestica pri utjecaju gravitacije, no kada bi se neki drugi rad fokusirao na samu simulaciju, a ne algoritam detekcije sudara sama simulacija bi donjela puno veću vrijednost.

7.3 Unapređenje simulacije iz dvodimenzionalnog prostora u trodimenzionalni

Svi korišteni algoritmi i strukture podatka bi se relativno jednostavno mogle prenamijeniti za rad u višoj dimenziji gdje je puno veći broj čestica koje bi se trebale simulirati. Tamo bi optimizacija vremena izračuna bila još korisnija.

8. Zaključak

Naivni algoritma detekcije sudara ima kvadratnu složenost koja se loše skalira čak i na modernom sklopovlju tako da ga je potrebno ubrzati. Prva stvar koju je moguće učiniti je paralelizirati sam algoritam, no to još uvijek nije dovoljno jer se time dobiva linearno ubrzanje koje raste sporije od kvadratne složenosti algoritma. Paralelizacija algoritma na grafičkoj kartici dovodi do značajnog ubrzanja s obzirom na paralelizaciju na procesor, no još uvijek nije moguće simulirati dovoljan broj čestica za ozbiljniju simulaciju.

Potrebno je optimirati sam algoritam detekcije sudara, a to se radi podjelom prostora na segmente. U ovom radu radi se o osnovnoj podjeli na kvadratnu mrežu, gdje je svaka ćelija iste veličine i uvijek je potrebno provjeriti sve ćelije. Tom optimizacijom postižu se značajno bolje performance i moguće je izračunati sudare stotinama tisuća čestica u stvarnom vremenu na procesoru, a još i više na grafičkoj kartici.

Daljnji koraci bi bili dodatna optimizacija algoritma korištenjem naprednije podijele prostora i implementacija tih algoritama na grafičkoj kartici.

9. Literatura

[1] 2D collision detection, poveznica: https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection ; pristupljeno: 10.05.2024.

[2] CUDA C++ Programming Guide. poveznica: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html ; pristupljeno: 05.05.2024.

[3] Spatial Partition, poveznica: https://gameprogrammingpatterns.com/spatial-partition.html, pristupljeno: 05.05.2024.

[4] Dear Imgui Github Repozitorij, poveznica: https://github.com/ocornut/imgui, pristupljeno: 29.04.2024.

[5] Izvorni kod rada, poveznica: https://github.com/kulaxa/Particle-Collision-Optimization, pristupljeno: 16.05.2024.

Sažetak

Cilj ovoga rada je bio ubrzati algoritma detekcije i rješavanja sudara u sustavima čestica. Korištene metode su bile paralelizacija algoritma na procesoru, paralelizacije algoritma na grafičkoj kartici i optimizacija algoritma podjelom prostora.

Sve tri metode su implementirane i provedena su testiranja vremena izvođenja za svaku od metoda. Isto tako je implementirana jednostavna simulacija i grafičko sučelje za kontrolu simulacije. Rezultati testiranja su analiziani i međusobno uspoređeni. Za kraj su navedene i opisane mogućnosti za daljnji razvoj.

Iscrtavanje volumetrijskih podataka
Tehnička dokumentacija
Verzija <1.0>

Studentski tim: Željko Kelava

Nastavnik: Željka Mihajlović

Sadržaj

1. Opis razvijenog proizvoda

U ovom projektu izrađena je aplikacija za prikaz volumetrijskih podataka koristeći OpenGL sučelje. Algoritam koji se koristi za pretvaranje skupa slika rasterskog oblika u 3D poligonalnu mrežu zove se algoritam pokretnih kocki (engl. *marching cubes*).

Ovaj projek nastavlja se na aplikaciju izrađenu u sklopu seminara. Algoritam pokretnih kocki sada koristi interpolaciju, iscrtavanje je prebačeno na grafičku karticu te je moguće direktno učitavati .nii datoteke.

2. Tehničke značajke

2.1 Korišteni alati

OpenGL – sučelje koje definira standard za iscrtavanje 2D i 3D grafike

GLEW – biblioteka koja dodaje proširenja za OpenGL

GLFW – biblioteka koja kontrolira stvaranje prozora te obradu unosa korisnika

GLM – biblioteka koja implementira operacije nad matricama prilagođena OpenGL-u

nifticlib – biblioteka za čitanje .nii datoteka

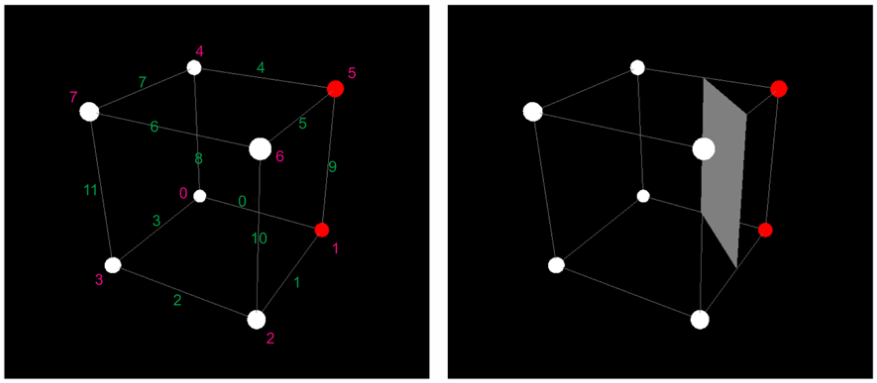
imgui – biblioteka za prikaz korisničkog sučelja

2.2 Opis aplikacije

Aplikacija je izrađena u programskom jeziku C++. Prijašnja verzija aplikacije koristila je *freeglut* biblioteku za upravljanje prozorom i unosom. Ta biblioteka ne radi s modernim *OpenGL*-om pa je umjesto nje korištena biblioteka *GLFW*. Ta promjena je napravljena kako bi se iscrtavanje moglo prebaciti na grafičku karticu.

Rezultat pokretanja programa je 3D vizualizacija podataka dobivenih snimanjem magnetskom rezonancom (eng. *MRI*). Uređaj za snimanje magnetskom rezonancom proizvodi 3D podatke tako da uzima veći broj 2D slika (poprečnih presjeka) koje zatim postavlja jednu na drugu. Svaka točka dobivena takvim snimanjem ima samo jedan podatak odnosno vrijednost. Algoritam koji pretvara tako dobivene podatke u 3D poligonalnu mrežu zove se algoritam pokretnih kocki (eng. *marching cubes*).

Osim tih podataka, algoritam pokretnih kocki kao ulaz još prima vrijednost koja predstavlja granicu. Sve točke koje imaju vrijednost veću od vrijednosti granice pripadaju 3D tijelu, ostale ne pripadaju. Algoritam zatim prolazi kroz sva 2x2x2 susjedstva i ovisno koje točke tog susjedstva pripadaju, odnosno ne pripadaju 3D tijelu određuje kako izgleda granica u tom dijelu. Svi slučajevi (njih 256) određeni su prije i zapisani u tablici. Slika 2.1 prikazuje primjer kako može izgledati to susjedstvo.

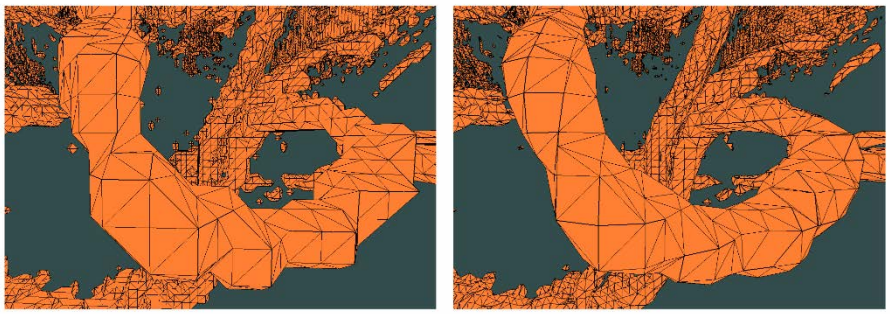


Slika 2.1 Primjer 2x2x2 susjedstva i izgled granice

Program započinje čitanjem .nii datoteke koristeći biblioteku *nifticlib*. Prolazi kroz sve vrijednosti i pronalazi maksimum te zatim skalira te vrijednosti na interval [0, 255]. Zatim uzima sva 2x2x2 susjedstva i provodi algoritam pokretnih kocki. U jednostavnom slučaju točke koje određuju granicu nalaze se na sredini bridova. Uzmimo za primjer samo 2 točke i želimo odrediti gdje započeti vrh poligona. U prošloj verziji programa granica bi bila točno na sredini između vrhova. Recimo da jedna točka ima vrijednost 40, a druga 100 i da je granica pripadnosti 50. Očito druga točka puno više „pripada“ 3D tijelu, odnosno granica ne bi trebala biti na sredini već bliže prvoj točki. Mjesto gdje želimo postaviti granicu možemo dobiti formulom:

$$t = \frac{granica - vrijednost_1}{vrijednost_2 - vrijednost_1}$$

gdje *t* predstavlja omjer gdje granica treba biti. Za *t* = 0.5 granica je na sredini. U slučaju od prije t = 0.167. Slika 2.2 prikazuje razliku između algoritma bez interpolacije i s interpolacijom.



Slika 2.2 Lijevo algoritam bez interpolacije, desno s interpolacijom

Drugo unapređenje aplikacije uključuje prebacivanje iscrtavanja poligona na grafičku karticu. Da bi se podaci iscrtavali na grafičkoj kartici potrebno je inicijalizirati buffer i kopirati podatke. Zatim je potrebno definirati organizaciju podataka u bufferu (broj komponenti vrha, tip podatka, korak, pomak ...). Osim toga potrebno je još napisati kôd za *vertex shader* i *fragment shader*. Vertex shader služi za transformaciju točaka. U ovom slučaju točke se samo množe s projekcijskom matricom i matricom pogleda. Uloga fragment shadera je bojanje točaka. I ovaj shader je vrlo jednostavan jer svakoj točki (poligonu) pridružuje istu boju.

```
void initBuffer(std::vector<Triangle>& vertexData){

    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vbo);

    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vertexData.size() * sizeof(Triangle), vertexData.data(), GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Point3D), (void*)0);
}
```

Za podatke veće rezolucije prebacivanje iscrtavanja na grafičku bilo je neophodno. Prošla verzija programa je za neki ulazni skup podataka imala oko 20 sličica po sekundi. Kod iscrtavanja na grafičkoj kartici taj broj porastao je na oko 180.

3. Upute za korištenje

Kôd napisan u C++ -u je dostupan na git-u. Potrebno ga je skinuti i kompajlirati. Na git-u se nalaze sve potrebne biblioteke te naredba za kompajliranje. Aplikaciju je potrebno pokrenuti i odabrati *File -> Open*, i učitati .nii datoteku.

Kretanje u 3D prostoru ostvareno je tipkama „WASD“. Kamera se pomiče micanjem miša. Pritiskom na tipku „I“ moguće je uključiti ili isključiti interpolaciju u algoritmu pokretnih kocki.

Tipke „Q“ i „E“ kontroliraju granicu koja određuje pripadnost točaka objektu.

Pritiskom na tipku „O“ iz geometrije nastaje .stl datoteka.

4. Literatura

[1.] Paul Bourke, Polygonising a scalar field, https://paulbourke.net/geometry/polygonise/

[2.] William E. Lorensen i Harvey E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, https://dl.acm.org/doi/10.1145/37402.37422

[3.] OpenGL, https://www.opengl.org/

[4.] GLEW, https://glew.sourceforge.net/

[5.] GLFW dokumentacija, https://www.glfw.org/documentation.html

[6.] nifticlib, https://sourceforge.net/projects/niftilib/files/nifticlib/

Crtanje po objektima u VR-u
Tehnička dokumentacija
Verzija 1.0

Studentski tim: Matija Kunc

Nastavnik: Željka Mihajlović

Sadržaj

1. Opis razvijenog proizvoda

U sklopu ovog projekta razvijena je aplikacija za crtanje po objektima u VR-u unutar grafičkog programskog pogona Unity. Ova aplikacija je nadogradnja, odnosno poboljšanje prethodne aplikacije za crtanje po objektima u VR-u izrađene u sklopu seminara.

Poboljšanja u odnosu na prethodnu verziju uključuju promjenu pomoćne biblioteke za virtualnu stvarnost, te dodatne funkcionalnosti promjene boje i debljine markera za crtanje.

2. Tehničke značajke

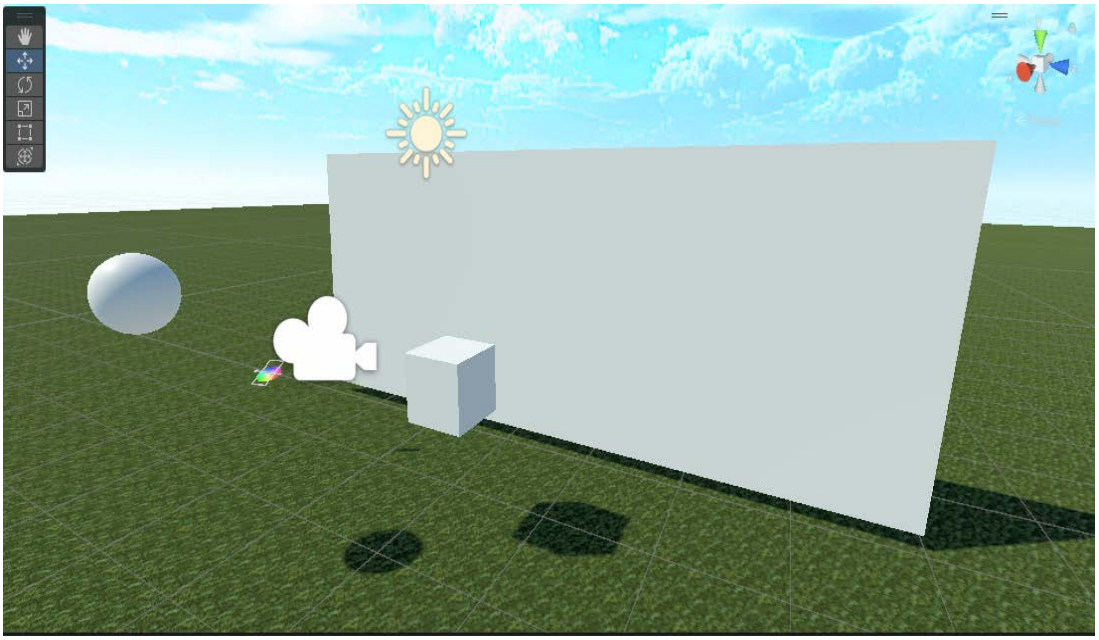
2.1 Opis

Aplikacija izrađena u Unity Engineu sastoji se od vanjske biblioteke koja podržava rad s VR sustavima (*OpenXR*), ovo je promjena u odnosu na prethodnu verziju aplikacije koje je koristila biblioteku specifičnu za rad s HTC Vive sustavom virtualne stvarnosti. Implementacija sa novom bibliotekom zahtijevala je ponovno mapiranje komandi na kontrolere. Primarna korist ove biblioteke bila je u većem spektru funkcionalnosti koje omogućuju jednostavniju implementaciju praćenja virtualnih kontrolera i njihove interakcije s ostalim virtualnim objektima.

Skripte za crtanje i pomoćne skripte za označavanje objekata i dohvaćanje raznih svojstava objekata (npr. *tag*, materijal i sl.) napisane su po uzoru na skripte iz prethodne verzije uz manje preinake. Ove skripte sadrže najveći dio logike crtanja. Sve skripte razvijene su u *C++* programskom jeziku, koji je standardni jezik koji se koristi u Unity Engineu.

Nove funkcionalnosti su promjena boje i debljine markera. Debljina markera implementirana je s pomoću klizača, elementa korisničkog sučelje Unity Enginea. Skripta koja ih povezuje jednostavno očitava vrijednosti klizača i ovisno o njima mijenja debljinu markera, odnosno radijus kružnice unutar koje se mijenja boja na teksturi objekta. Promjena boje je implementirana uz pomoć kotača boja, skripte koja je povezana s kontrolerima i očitava vrijednosti boje točke u koju kontroler pokazuje pri promjeni boje.

Za potrebe testiranja i demonstracije napravljena je jednostavna scena s jednostavnim objektima.



Slika 1. Scena za testiranje

2.2 Implementirani elementi i poboljšanja

2.2.1 Crtanje

Glavni dio aplikacije odnosi se na logiku crtanja. Na objekt markera dodijeljene su skripte za detekciju sudara. One rade na principu ispucavanja zrake i detekcije presjeka te zrake s objektima scene. Ideja je sljedeća, vrh markera je objekt kocke, na njega je vezana skripta koja cijelo vrijeme iz gornje strane kocke ispucava nevidljivu zraku, kada ta zraka prođe kroz neki objekt provjeravaju se neki uvjeti. Ti uvjeti su sljedeći: udaljenost od početka zrake (vrha kocke) do objekta je dovoljno mala (kako ne bi bilo moguće crtanje po objektima s beskonačnih ili velikih udaljenosti, već samo kada je marker jako blizu objekta), zatim se provjerava oznaka (*tag*) objekta, ako je ta oznaka *Drawable* onda to označava objekt po kojem se može crtati, zatim se pokušava dohvatiti skripta objekta po kojem crtamo koja vraća referencu na materijal objekta i njegovu teksturu. Kada to sve imamo kreće postupak crtanja. Detektira se točka u kojoj zraka sječe objekt, ta se točka onda preslikava na teksturu objekta, te se onaj piksel na teksturi koji odgovara pikselu na objektu u 3D prostoru mijenja u boju materijala markera. Pri tom se prati prethodna točka presjeka, odnosno točka koja se promijenila u prethodnom iscrtavanju (*frameu*) i interpolira se, između te stare točke i nove točke, linija koja ih spaja u nekom broju koraka (u projektu 100, ali moguće je i više, samo što to usporava proces). Tako dobivamo pune linije umjesto samo točaka ako se marker po objektu kreće brzo. Kôd je, kao i u prvoj verziji napisan na temelju kôda s [6].

```
private void Draw()
{
    if (Physics.Raycast(_tip.position, transform.up, out _touch, _tipHeight))
    {
        if (_touch.transform.CompareTag("Drawable"))
        {
            if (_drawable == null)
            {
                _drawable = _touch.transform.GetComponent<Drawable>();
            }
            UpdatePenColor();
            _touchPos = new Vector2(_touch.textureCoord.x, _touch.textureCoord.y);

            var x = (int)(_touchPos.x * _drawable.textureSize.x - (_penSize / 2));
            var y = (int)(_touchPos.y * _drawable.textureSize.y - (_penSize / 2));

            if (y < 0 || y > _drawable.textureSize.y || x < 0 || x > _drawable.textureSize.x) return;

            if (_touchedLastFrame)
            {
                _drawable.texture.SetPixels(x, y, _penSize, _penSize, _colors);

                // Interpolating between previous touch position and current position
                for (float f = 0.01f; f < 1.00f; f += 0.01f)
                {
                    var lerpX = (int)Mathf.Lerp(_lastTouchPos.x, x, f);
                    var lerpY = (int)Mathf.Lerp(_lastTouchPos.y, y, f);
                    _drawable.texture.SetPixels(lerpX, lerpY, _penSize, _penSize, _colors);
                }

                transform.rotation = _lastTouchRot;

                _drawable.texture.Apply();
            }

            _lastTouchPos = new Vector2(x, y);
            _lastTouchRot = transform.rotation;
            _touchedLastFrame = true;
            return;
        }
    }

    _drawable = null;
    _touchedLastFrame = false;
}
```

Slika 2 Isječak kôda za logiku crtanja

Slika 3. Natpis FER rukom napisan na plohi

2.2.2 Praćenje kontrolera

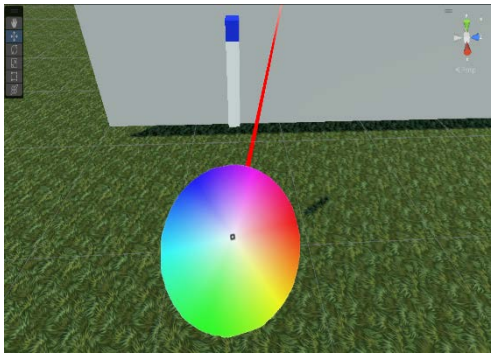
Praćenje kontrolera radi se s pomoću pomoćne biblioteke *OpenXR*, koja je među ponuđenima u Unity Engineu za rad s VR sustavima. Razlika između *OpenXR* i HTC VIVE biblioteke je ta što je *OpenXR* napravljena da podržava velik broj VR sustava različitih marki, i što ima bogatije funkcionalnosti nego HTC VIVE-ova. Glavni dobitak korištenja ove biblioteke je ta što je jednostavnije kontrolirati mogućnosti da virtualni kontroleri prolaze kroz virtualne objekte, kada stvarni se kontroleri pomiču kroz njih. Ovaj problem je postojao u prošloj verziji projekta, u kojoj je bilo teško za crtati po objektima, zato što je bilo potrebno jako precizno pomicati kontrolere tako da se u virtualnom svijetu ne bi pozicionirali iza lica objekta. Bilo je potrebno držati ih na točnoj udaljenosti od površine objekta i svaki pokret koji to ne bi ostvario imao bi neželjene efekte, kao odmicanje položaja virtualnog od položaja stvarnog kontrolera, nekontrolirano trzanje, pa čak i pucanje položaja virtualnih kontrolera u prostoru i izlijetanjem u nekom smjeru.

S pomoću *OpenXR* biblioteke bilo je moguće postaviti skriptu *Grabable* na objekt markera, kojim bi taj marker onda mogli dohvatiti u virtualnom prostoru, za razliku od prije kada je marker bio povezan uvijek s položajem kontrolera. Ta skripta omogućila je da i ostavimo marker sa strane ili ga primimo drugom rukom, što je isto vrlo korisno. Naravno trebalo je podesiti neke parametre na skripti i dodatno ponovno mapirati komande gumbe kontrolera, što je u prošloj verziji bilo nešto jednostavnije jer je HTC VIVE-ova biblioteka imala intuitivnije i slikovitije korisničko sučelje nego *OpenXR*.

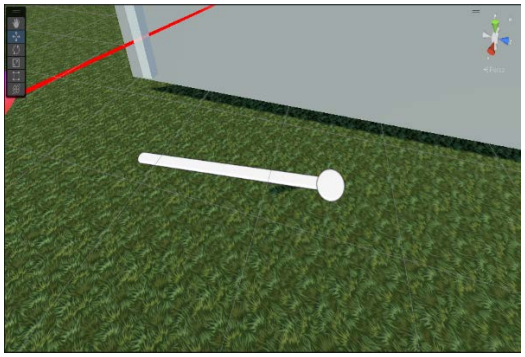
Implementacijom s ovom bibliotekom dobila se funkcionalnost da se marker pri sudaru s drugim objektom ne prolazi kroz objekt već se „zalijepi“ vrhom na površinu, nakon čega ga je moguće micati po površini i ostavljati trag.

2.2.3 Promjena boje i debljine markera

Ovo su dvije nove funkcionalnosti u odnosu na prošlu verziju projekta. Debljina markera ostvarena je s pomoću Unity-jevog UI VR klizača. UI VR klizač je objekt koji je dio korisničkog sučelja, on je dijete praznog objekta koji je povezan s položajem kontrolera, također putem roditeljske veze. Klizač se nalazi na desnoj ruci, odnosno odmah iznad desnog kontrolera, kako bi mu pristup bio što lakši. Moguće ga je sakriti pritiskom na okidač na desnom kontroleru. Skriptom je povezan s markerom i njegova vrijednost koja varira se množi pri izračunu površine koju treba pobojati pri crtanju.

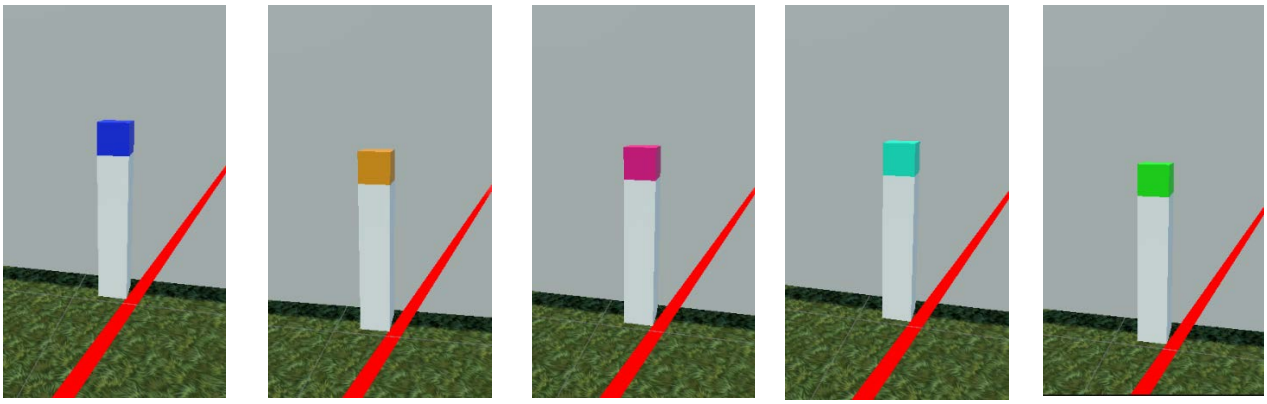


Slika 4. Kotač boja



Slika 5. Klizač za debljinu markera

Promjena boje ostvarena je s pomoću kotača boje. U projekt je dodana slika kotača boje i slika je na sličan način kao i klizač postavljena u scenu iznad lijevog kontrolera. Ako želimo promijeniti boju jednostavno drugim kontrolerom uperimo u sliku i pritiskom na gumb kontrolera dohvaćamo boju koju smo pokazali. Ovo je ostvareno skriptom na sličan način kao i crtanje, naime, koristimo zraku koja izlazi iz markera i tražimo presjek te zrake sa slikom kotača boje. Ako smo našli presjek, dohvaćamo vrijednosti boje teksture u pikselu koji odgovara točki na koju smo pokazali. Kada je ta vrijednost dohvaćena, mijenjamo boju materijala markera s drugom skriptom. Tako nije potrebno dodatno popravljati skriptu za crtanje jer ona već dohvaća boju materijala markera, i dobivamo vizualnu povratnu informaciju da smo uspješno promijenili boju jer se vrh markera isto promijenio.



Slika 6. Markeri u raznim bojama

3. Upute za korištenje

Aplikaciju je moguće pokrenuti s pomoću Unity Enginea. Potrebno je otvoriti projekt u Unity Engineu kao postojeći projekt. Naravno, potrebno je i spojiti s računalom VR sustav i napraviti potrebne korake u instalaciji programa za virtualnu stvarnost ovisno o marki sustava koja se koristi. Za sâmu aplikaciju ne bi trebalo biti potrebno išta dodatno raditi već je samo otvoriti i pritisnuti *Play* kako bi se pokrenula scena.

- Kada se scena pokrenula moguće se po njoj kretati s lijevim kontrolerom s pomoću *touchpada* na gornjoj strani kontrolera.
- S *touchpadom* na desnom kontroleru moguće je okretati se oko z osi u sceni.
- Na pritisak okidača na desnom kontroleru otvara se klizač za podešavanje debljine markera
- Na stisak kontrolera (bočne tipke koje rade na stisak ruke oko kontrolera) i upiranjem u marker, marker će biti dohvaćen u ruku koja odgovara kontroleru kojim je naciljan
- Na stisak kontrolera dok je marker u ruci i upiranjem u kotač boje dohvaća se boja koju smo naciljali i mijenja boju markera

Link na Git repozitorij: Yami0305/projektR

4. Literatura

1. Ziyang Zhang, Jurgen P. Schulze; University of California: VirtualForce: Simulating Writing and Sketching on a 2D-Surface in Virtual Reality (pdf)
2. Alexander Adrahtas, David Dumas, Alexander Guo, Gregory Schamberger: Drawing on Surfaces in VR
3. Open Brush – link na stranicu Open Brush alata za crtanje
4. Gravity Sketch | 3D sketching and design software - link na stranicu Gravity Sketch alata za crtanje
5. How to Create a Whiteboard in Unity VR - A Step-by-Step Guide - YouTube
6. Justin Barnett - Welcome To My Channel (youtube.com)

Proceduralna animacija humanoidnog lika
Tehnička dokumentacija
Verzija <1.0>

Studentski tim: Marko Prosenjak

Nastavnik: Željka Mihajlović

Sadržaj

Tehnička dokumentacija

Na koji način koristiti predložak?

Dokument se po potrebi može prilagoditi potrebama pojedinog projekta promjenom predloženih naslova predloženih poglavlja, kao i eventualnim dodavanjem novih poglavlja i potpoglavlja.

Cilj dokumenta je opisati rezultat rada studentskog tima, problem koji je riješen u okviru projekta, korištenu tehnologiju, mogućnosti i značajke dobivenog proizvoda i sl. Razinu detalja opisanu u ovom dokumentu studentski tim treba dogovoriti s nastavnikom.

Literatura:
U tekstu rada treba biti navedena literatura svugdje gdje je tekst, slika ili grafički prikaz preuzet ili se temelji na nekom pisanom predlošku. Literatura se navodi iza zaključka. U tekstu se literatura navodi unutar zagrada s navođenjem prvog autora i godine izdanja, npr. (Martinis, 1998).

Primjer citiranja knjige:
Prezime, inicijal(i) imena autora. Naslov: podnaslov. Podatak o izdanju. Mjesto izdavanja: Nakladnik, godina izdavanja.

Primjer citiranja članka u časopisu:
Prezime, inicijal(i) imena autora. Naslov članka: podnaslov. Naziv časopisa. Oznaka sveska/godišta, broj(godina), str. početna-završna.

Primjer citiranja rada sa konferencije:
Prezime, inicijal(i) imena autora. Naslov rada: podnaslov. Naslov zbornika, mjesto održavanja konferencije, (godina), str. početna-završna.

Primjer citiranja doktorskog, magistarskog ili diplomskog rada:
Prezime, inicijal(i) imena autora. Naslov. Vrsta rada. Ustanova na kojoj je rad obranjen, godina.

Primjer citiranja www izvora:
Ime(na) autora (ako je/su poznata), naslov dokumenta, datum nastanka (ako se razlikuje od datuma pristupa izvoru), naslov potpunog djela (italic), potpuna http adresa, datum pristupa dokumentu.

Ostale upute
U svim dokumentima obvezno primjenjivati SI jedinice. Slike, formule i tablice potrebno je numerirati. Opis tablice stavlja se iznad, a opis slike ispod nje. U opisu slike ili tablice pišu se samo podaci neophodni za njeno razumijevanje (npr. Slika 6. Pojačalo s promjenljivim pojačanjem). Dodatna objašnjenja daju se u tekstu uz povezivanje sa slikom ili tablicom. Osi i parametri na slikama i grafičkim prikazima trebaju biti obilježeni. Daljnji opis tog grafičkog prikaza treba se nalaziti u tekstu rada. Formule se obilježavaju brojevima u zagradi, uz desni rub stranice, a u tekstu se poziva na broj formule.

1. Opis razvijenog proizvoda

Razvijena je proceduralna animacija hodanja 3D humanoidnog lika unutar Unity okruženja, koristeći inverznu kinematiku (IK) i matematičke modele za simulaciju prirodnih pokreta. Za razliku od klasičnih animacija temeljenih na ključnim okvirima (keyframe animation), ovaj sustav ne koristi unaprijed pripremljene animacije, već generira pokrete u stvarnom vremenu na temelju logike i podataka iz virtualnog okruženja.

Glavni elementi proceduralne animacije uključuju kinematičke lance za noge (*Brackeys, 2020*), gdje svaka noga prati unaprijed određenu metu u prostoru. Ove mete se dinamički pomiču i prilagođavaju terenu, omogućujući prirodnije kretanje. Postignuta je realističnost hodanja kroz nekoliko ključnih mehanizama:

- Prilagođavanje stopala podlozi radi osiguravanja stabilnosti pri kretanju.
- Dinamičko podizanje i spuštanje trupa ovisno o visini podloge ispod stopala.
- Naizmjenična animacija nogu, koja omogućuje neprekinut i prirodan hod.

Kako bi se testirala prilagodljivost i funkcionalnost sustava, izrađena je 3D scena s različitim preprekama koje izazivaju sustav proceduralne animacije. Ove prepreke uključuju:

- Stepenice različitih visina, koje provjeravaju sposobnost prilagođavanja visini koraka.
- Neravan teren s nasumično postavljenim uzvišenjima i udubljenjima.
- Površine s nagibom, omogućujući penjanje i spuštanje.
- Kocke različitih dimenzija postavljene na ograničenom prostoru, testirajući preciznost položaja stopala.

Kretanje lika kontrolira se putem XBOX ONE joysticka, dok implementirana kamera određuje smjer kretanja. Model se uvijek rotira kako bi se usmjerio prema pogledu kamere, čime je postignut intuitivan način upravljanja.

Cilj projekta bio je stvoriti sustav proceduralne animacije koji omogućava realistično i prilagodljivo kretanje u različitim uvjetima. Postignuti rezultati potvrđuju sposobnost sustava da odgovori na dinamične promjene terena, stvarajući uvjerljiv osjećaj interakcije s okolinom bez potrebe za klasičnim ručno animiranim sekvencama. Sustav je potrebno nadograditi jer postoje slučajevi u kojima ponašanje odudara od očekivanog, te se time smanjuje uvjerljivost pokreta.

2. Tehničke značajke

a) Pokretanje i rotacija modela

Svaki vremenski okvir (eng. „frame“) se provjerava postoji li ulazna informacija koja dolazi iz joystick-a. Ako postoji, potrebno je ažurirati smjer kretanja modela. Novi smjer kretanja („forward vector“) se određuje uz pomoć vrijednosti dobivene iz joystick-a. Igrač se pokreće uz pomoć lijeve gljivice, stoga je potrebno očitati njenu x i y vrijednost (obje se kreću u rasponu [-1,1]). Uz pomoć tih vrijednosti moguće je odrediti pod kojim kutom igrač drži gljivicu. S obzirom na to da pozicija gljivice x=0, y=1 (gljivica pomaknuta prema gore) u većini videoigara označava smjer kretanja prema naprijed, model u slučaju držanja gljivice u toj poziciji, ne bi trebao mijenjati svoj smjer kretanja, odnosno treba ga rotirati za 0°. To znači da je potrebno zamijeniti x i y vrijednosti u funkciji „atan2“. Dobiveni kut koristi se za konstruiranje Quaternion-a, koji predstavlja rotaciju gljivice, a potrebno ga je pomnožiti s rotacijom kamere. S obzirom na to da se model uvijek kreće u smjeru u kojem kamera gleda, potrebno je uračunati njenu rotaciju (ako je igrač postavio gljivicu na poziciju (0,1), očekuje da će se model kretati u smjeru u kojem je kamera orijentirana, ako je postavio gljivicu u poziciju (1,0), očekuje da će se model kretati desno u odnosu na smjer u kojem kamera gleda). Ovime je ažurirana varijabla „targetRotation“, koja označava krajnji smjer kretanja modela (prema njemu je potrebno animirati). Animacija rotacije modela dobivena je rotacijom Quaternion-a, počevši od trenutačne rotacije, sve do „targetRotation“, uz korak od „rotationSpeed * Time.deltaTime“. Dobiveni Quaternion se pohranjuje kao nova trenutačna rotacija modela.

```
private void UpdateForwardDirection()
{
    // angle of thumbstick during input
    thumbstickAngle = Mathf.Atan2(movementDirection.x, movementDirection.y) * Mathf.Rad2Deg;
    RotateCharacter(thumbstickAngle);
}

private void RotateCharacter(float rotationAngle)
{
    // rotation of the camera must be taken in consideration, otherwise
    // rotation would be done in the world system instead of player's
    Quaternion thumbstickRotation = Quaternion.Euler(0, rotationAngle, 0).normalized;
    targetRotation = _camera.GetCameraYRotation() * thumbstickRotation;
}

AnimateRotation(transform.rotation, targetRotation);
private void AnimateRotation(Quaternion startRotation, Quaternion endRotation)
{
    transform.rotation = Quaternion.RotateTowards(startRotation, endRotation, rotationSpeed *
Time.deltaTime);
}
```

Ako je na joysticku očitan unos (igrač pokreće lijevu gljivicu), nakon ažuriranja smjera kretanja, model se pokreće prema naprijed brzinom „movementSpeed * Time.deltaTime“.

b) Animacija noge

Obje noge imaju IK mete („IK target“) koje na sebi imaju IKFootSolver.cs skripte. Uz pomoć tih skripti se postavljaju i animiraju noge (*Unity, 2021*).
U svakom vremenskom okviru skripta provjerava treba li se noga početi animirati. Noga se smije pomaknuti ako su zadovoljeni sljedeći uvjeti: na joystick-u treba postojati očitanje, druga noga se ne smije micati, trenutačna noga treba završiti s animiranjem, te se ne smije dogoditi da se noga pomiče prije nego se druga noga pomaknula (ako je noga upravo završila s animacijom, ne smije se ponovno krenuti animirati sve dok druga noga nije završila sa svojom animacijom). Završni uvjet koji treba biti ispunjen je da udaljenost od presjeka zrake (ispaljene s pozicije trupa prema dolje („Vector3.down“)) s podlogom i trenutačne pozicije stopala, mora biti veća od vrijednosti „stepDistance“, što znači da noga zaostaje za trupom, te ju treba pomaknuti (*Codeer, 2020*).

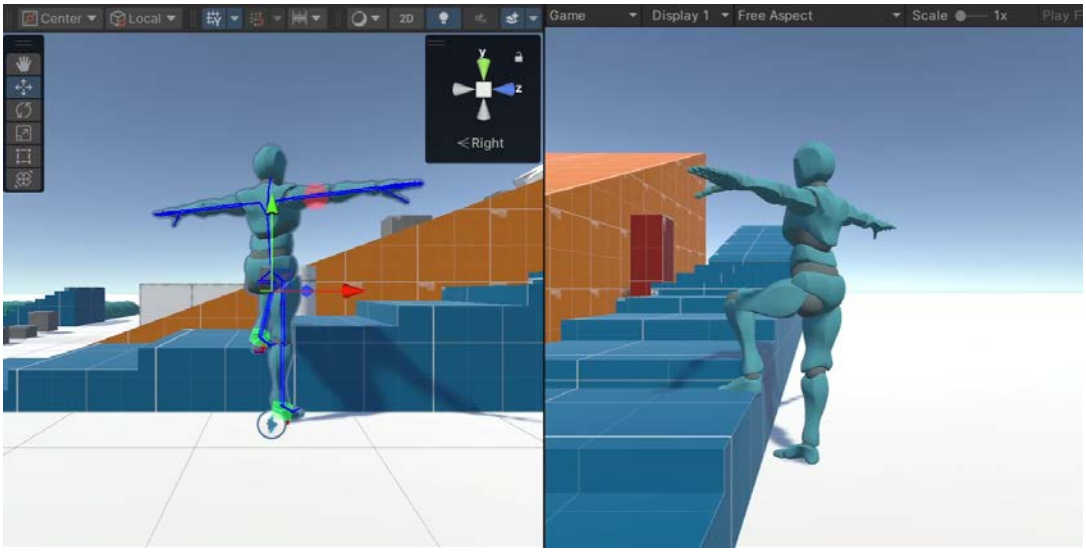
```
private bool ShouldMove()
{
    return proceduralMovement.DetectedMovementInput() &&
        !otherFoot.IsMoving() && animationCompleted >= 1f && !previouslyMoved &&
        (Vector3.Distance(bodyAlignedHit.point, currentPosition) > stepDistance);
}
```

Ako su svi uvjeti zadovoljeni, varijabla koja označava napredak animacije („animationCompleted“) postavlja se na 0, te se označava da se noga pomaknula, tako što se varijabla „previouslyMoved“ postavlja u „true“. Na kraju se u varijablu „oldBodyPos“ pohranjuje trenutačna pozicija trupa, a u varijablu „oldPosition“ se pohranjuje pozicija presjeka podloge sa zrakom, kojoj je ishodište pozicija stopala, a smjer kretanja je „Vector3.down“. Nakon toga noga ulazi u proces animacije, u kojem se zadržava sve dok je vrijednost varijable „animationComplete“ manja od 1. Ako se noga nalazi u procesu animacije, svaki vremenski okvir dolazi do ispaljivanja zrake u smjeru „Vector3.down“. Ishodište zrake je od „oldBodyPos“ udaljeno za „footSideOffset“ (x komponenta pozicije u lokalnom sustavu trupa – vrijednost za lijevu nogu će biti negativna, a za desnu pozitivna) u smjeru „body.right“ (vektor koji se pruža u smjeru pozitivne x osi trupa u globalnom koordinatnom sustavu), podignuta do visine koljena te pomaknuta prema naprijed za vrijednost puta koji se prijeđe za vrijeme animacije (uzima se u obzir trajanje animacije, te brzina kretanja modela).

```
Physics.SphereCast(oldBodyPos + body.right * footSideOffset + Vector3.up *
kneeHeightOffset + body.forward *
(proceduralMovement.GetMovementSpeed() / speed + stepLength)
          , sphereCastRadius, Vector3.down, out nHit, 1.5f, terrainLayer.value);
helperNewPos = nHit.point;
```

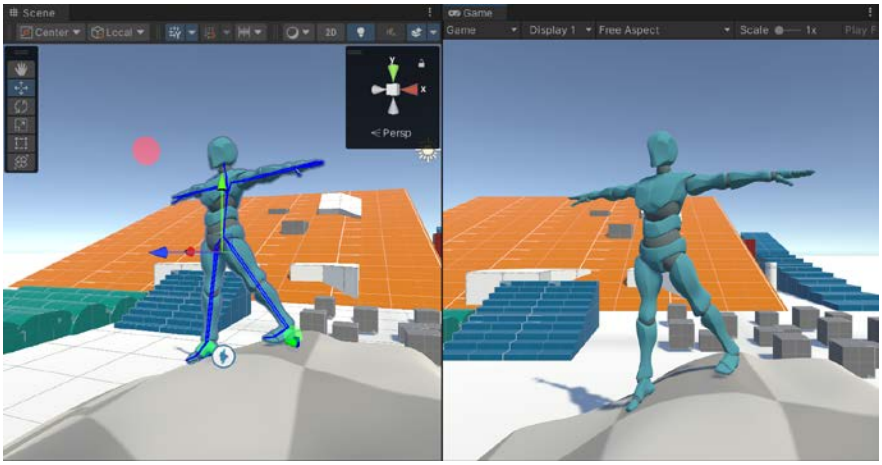
Na ovaj način ne dolazi do pogrešaka kod ispaljivanja zrake prilikom rotiranja modela (što je rezultiralo križanjem nogu, te zapinjanjem nogu kod naglih okreta - za kut veći od 90°), jer se koriste ažurne vrijednosti smjera pružanja vektora trupa (body.right i body.forward), dobivene nakon rotacije modela. Pozicija presjeka koristi se u funkciji „AnimateStep()“.

```
private void AnimateStep()
{
    currentPosition = Vector3.Lerp(helperOldPos, helperNewPos, animationCompleted);
    currentPosition.y += Mathf.Sin(animationCompleted * Mathf.PI) * stepHeight;
    currentNormal = Vector3.Lerp(oldNormal, newNormal, animationCompleted);
}
```



Slika 1 Postavljanje noge na stepenicu

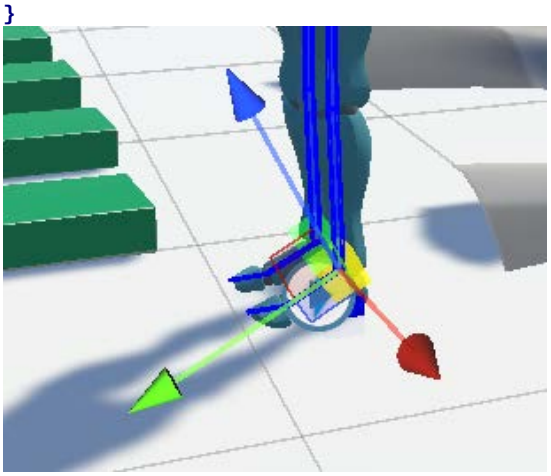
U funkciji se trenutačna pozicija mete izračunava uz pomoć linearne interpolacije između „helperOldPos“ (pozicija na kojoj se meta nalazila na početku animacije) i „helperNewPos“ (pozicija na kojoj bi se meta trebala naći na kraju animacije). S obzirom na to da obje pozicije predstavljaju presjek s podlogom, potrebno je podići metu kako bi se noga odvojila od podloge. Za to je korištena sinusna funkcija (period joj se mijenja ovisno o tome u kojem se trenutku animacija nalazi), te je pomnožena s visinom do koje se noga može podići (vrijednosti sin(x), $x \in [0, \frac{\pi}{2}] \cup [\frac{\pi}{2}, 1]$ preslikavaju se u vrijednosti [0, 1] U [1,0]). Također se linearno interpolira vektor normale stopala, koji se koristi za rotiranje stopala.



Slika 2 Hodanje po neravnom terenu

Rotacija stopala obavlja se u funkciji „AnimateFootRotation“. Quaternion koji se koristi za rotaciju stopala, dobiven je umnoškom druga dva Quaternion-a, koji predstavljaju rotaciju oko x, a potom y osi. Zbog početne rotacija stopala (definirane pri izradi modela, preuzetog sa stranice https://www.mixamo.com/#/), koordinatne osi stopala se ne poklapaju s koordinatnim osima svijeta. Stoga je potrebno odrediti Quaternion s kojim će se stopalo rotirati („footRotator“). On je dobiven kao umnožak Quaternion-a koji predstavlja rotaciju između normale podloge i Vector3.up (vektor u koordinatnom sustavu svijeta koji se pruža prema gore), s Quaternion-om koji predstavlja rotaciju između smjera u kojem se model kreće i „Vector3.forward“ (vektor u koordinatnom sustavu svijeta koji se pruža prema naprijed). Prvi Quaternion nam daje informaciju o rotaciji oko x osi, a drugi o rotaciji oko y osi. Na kraju je potrebno početnu rotaciju stopala pomnožiti s dobivenim Quaternion-om, kako bi se stopalo rotiralo i prianjalo uz podlogu.

```
private void AnimateFootRotation()
{
    footRotator = Quaternion.FromToRotation(Vector3.up, currentNormal) *
Quaternion.FromToRotation(Vector3.forward, body.forward);
    transform.rotation = footRotator * defaultToeRotation;
}
```

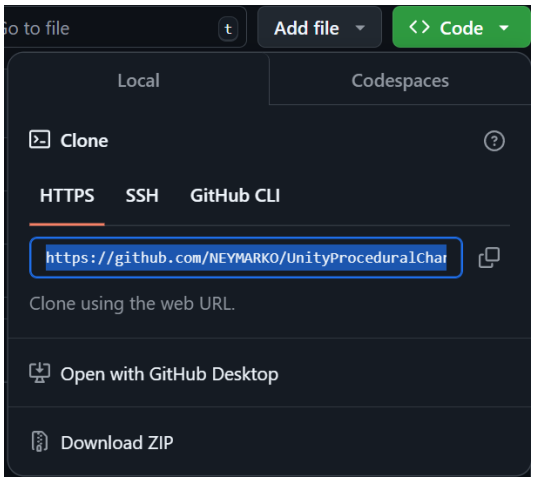


Slika 3 Početna rotacija stopala

3. Upute za korištenje

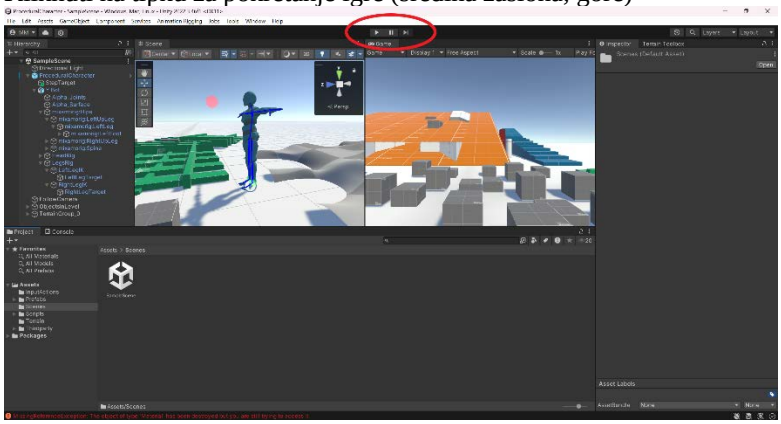
Za korištenje je potrebna Unity verzija „2022.3.46f1“ (ili više), te neki od sljedećih „joystick“-a: PS4 ili Xbox kontroleri („controllers“) za Windows

1. Otići na poveznicu https://github.com/NEYMARKO/UnityProceduralCharacter
2. Pritisnuti na gumb „Code“, te kopirati web URL za kloniranje repozitorija



Slika 4 web URL potreban za kloniranje repozitorija

3. Napraviti novi direktorij, ili odabrati već postojeći
   a. Pozicionirati se u odabrani direktorij naredbom „cd <putanja_do_direktorija>“
   b. Pokrenuti naredbu „git clone <web URL>“
4. Otvoriti UnityHub, te dodati klonirani projekt putem gumba „Add“ -> “Add project from disk“, nakon čega je potrebno odabrati „ProceduralCharacter“ direktorij unutar „UnityProceduralCharacter“ direktorija (to je klonirani direktorij, koji se nalazi u direktoriju u kojem je pokrenuta naredba „git clone“)
5. Otvoriti dodani projekt
6. Otvoriti „Assets\Scenes“ direktorij, te 2 puta kliknuti na „SampleScene“
7. Kliknuti na tipku za pokretanje igre (sredina zaslona, gore)



Slika 5 Gumb za pokretanje igre

8. Upravljati modelom uz pomoć lijeve (pomicanje modela) i desne gljivice (pomicanje kamere)

4. Literatura

Brackeys, *Make your Characters Interactive! - Animation Rigging in Unity*, 21/06/2020, https://www.youtube.com/watch?v=Htl7ysv10Qs, 23/09/2024

Codeer, *Unity procedural animation tutorial (10 steps)*, 28/03/2020, https://www.youtube.com/watch?v=e6Gjhr1IP6w, 05/10/2024

Unity, Creating procedural walk movement | Prototype Series, 17/02/2021, https://www.youtube.com/watch?v=acMK93A-FSY, 07/10/2024