SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 865

SIMULACIJA FLUIDA METODOM ČESTICA

Mario Jalšovec

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 865

SIMULACIJA FLUIDA METODOM ČESTICA

Mario Jalšovec

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 865

Pristupnik:	**Mario Jalšovec (0036529377)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Simulacija fluida metodom čestica**

Opis zadatka:

Proučiti fizikalne osnove ponašanja fluida. Proučiti osnove za implementaciju simulacije fluida česticama. Proučiti grafičko programsko sučelje Vulkan. Razraditi simulacijski model fluida uz implementaciju korištenjem grafičkog programskog sučelja Vulkan. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafičko programsko sučelje Vulkan i programski jezik C++. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 9. lipnja 2023.

Sadržaj

Uvod

Simulacija fluida u računalnoj grafici je postupak kojim se simulacijom pokušava dobiti realistično ponašanje tekućina i plinova koji u stvarnosti imaju jako kompleksno ponašanje. Ta kompleksnost proizlazi iz ogromnoga broja interakcija različitih molekula i čestica. Zbog te kompleksnosti nije moguće simulirati potpuno točno fizikalno ponašanje, ali zato postoje mnoge metode simulacije fluida kojima se može doći dovoljno blizu stvarnog ponašanja koje će biti dovoljno precizno za pojedinu upotrebu.

Najčešća upotreba ovih simulacija je u filmovima i video igrama u kojima je bitna realističnost. U zadnjih dvadesetak godina došlo je do značajnog napretka u grafičkom hardveru koji omogućuje vrlo precizne i detaljne simulacije.

Cilj ovog rada je razumjeti fiziku koja je potrebna za ovakve simulacije i na kojoj se svi algoritmi simulacije fluida temelje i onda primjenom tih algoritama ostvariti osnovnu simulaciju fluida i prikazati ostvarene rezultate.

1. Fizika simulacije fluida

1.1. Navier–Stokes jednadžbe

Navier–Stokes jednadžbe su parcijalne diferencijalne jednadžbe koje opisuju gibanje fluida kao što su tekućine i plinovi. Dobile su ime po francuskom fizičaru Claude-Louis Navier-u i engleskom fizičaru George Gabriel Stokes-u koji su ih stvarali dugi niz godina 19. stoljeća.

Problem ovih jednadžbi je da iako potpuno opisuju ponašanje fluida za njih ne postoji generalno rješenje ili algoritam kojim bi se za svaki slučaj i svaki dio prostora mogle egzaktno riješiti nego su za njihovo rješavanje potrebne numeričke metode koje procjenjuju njihova rješenja. [1]

$$\nabla \cdot \overline{u} = 0$$

$$\rho \frac{D\overline{u}}{Dt} = -\nabla p + \mu \nabla^2 \overline{u} + \rho \overline{F}$$

Slika 1.1 Navier-Stokes jednadžbe [1]

Prva jednadžba sa Slike 1.1 predstavlja jednadžbu kontinuiteta koja osigurava očuvanje mase. Druga jednadžba sa Slike 1.1 predstavlja konzervaciju gibanja fluida. U jednadžbi se koriste trodimenzionalni vektori zbog čega je drugu jednadžbu moguće zapisati kao tri jednadžbe, jednu za svaku dimenziju prostora.

Još uvijek nije dokazano da za ove jednadžbe postoji rješenje koje vrijedi za sve točke u prostoru, ali znamo da njihove aproksimacije daju prilično dobre rezultate u praksi što nam je dovoljno dobro za njihovu upotrebu. [1]

2. Računalne simulacije fluida

Kod računalnih simulacija fluida postoje dva glavna pristupa, Eulerov i Lagrangeov pristup. Eulerov pristup simulira fluid na određenim točkama u prostoru, uglavnom korištenjem strukture podataka mreže (engl. 'grid'). Lagrangeov pristup simulira fluid korištenjem čestica na temelju njihove međusobne interakcije. Oba pristupa imaju svoje prednosti i mane i zato niti jedan nije bolji u svim slučajevima i za sve primjene. [2]

2.1. Eulerov pristup

Glavna značajka Eulerovog pristupa je da gleda fluid na makro razini, nije usredotočen na pojedine čestice nego samo na veće dijelove fluida koji su najčešće podijeljeni u ćelije mreže. Ovakve simulacije su puno efikasnije za simulacije velikih fluida s velikim protokom i njihovim ponašanjem na većoj skali. Odličan je za simuliranje fluida koje nije moguće jako komprimirati kao što su naprimjer voda i protok vode i simuliranje protoka i širenja plina. Za vrijeme simulacije poprilično je lagano dobiti informacije o fizičkim značajkama fluida kao što su tlak, temperatura i gustoća.

Većina algoritma koji se temelje na ovom pristupu koriste pretpostavku da fluid koji simuliraju nije kompresibilan i zbog toga lagan način mogu modelirati proticaj kroz ćelije mreže. To je moguće zato jer po definiciji nekompresibilnosti znamo da onoliko fluida koliko uđe u ćeliju mora i iz nje izaći.

No ovakvi algoritmi imaju i neke mane, kao što ne praktična simulacija interakcije dva fluida kao naprimjer voda i zrak i isto tako imaju problema kod simuliranja čvrstih objekata koji se kreću kroz simulirani fluid.

2.2. Lagrangeov pristup

Za razliku od Eulerovog pristupa koji simulira fluid na makro razini, Lagrangeovog pristup simulira fluid na razini čestica i na temelju njihovih interakcija određuje se ponašanje cijeloga

fluida. Ovaj pristup je znatno precizniji u simulaciji fluida od Eulerovog pristupa i znatno je fleksibilniji oko toga što može simulirati.

Glavna mana Lagrangeovog pristupa je da je kod njega potrebno puno više računalnih resursi jer su simulacije znatno zahtjevnije zbog velikog broja čestica unutar simulacije.

2.3. Kombinirani pristup

U praksi se najčešće koristi kombinacija Eulerovog i Lagrangeovog pristupa jer se pokazalo da najviše primjena zahtijeva preciznost Lagrangeovog pristupa, ali da često nemamo pristup tolikim računalnim resursima potrebnim za njegovu simulaciju. Zbog toga nam je potrebna određena ušteda koju možemo dobiti kombinacijom ta dva pristupa. Jedna od metoda koja je nastala kombiniranjem Eulerovog i Lagrangeovog pristupa je PIC (engl. Particle In Cell) metoda. [4]

2.3.1. PIC metoda

PIC metoda od Eulerovog pristupa preuzima mrežu kao strukturu podatka koja sadrži ćelije gdje svaka od njih, preciznije svaki kut ćelije u sebi sadrži određena fizikalna svojstva koja karakteriziraju fluid. Neka od svojstva fluida koja mogu biti sadržana u ćelijama su brzina, tlak, temperatura, viskoznost i gustoća fluida. No PIC metoda kod simulacije koristi i čestice što preuzima iz Lagrangeovog pristupa. Svaka čestica ima određena fizikalna svojstva, glavna od njih su iznos i smjer brzine, ali može imati i neka druga svojstva kao što su masa, akceleracija i veličina za precizniju simulaciju.

Ovisno o tipu simulacije mogu postojati razini tipovi ćelija, no u osnovnim simulacijama koje simuliraju interakciju tekućine i zraka postoje tri tipa ćelija. To su ćelije tekućine, ćelije zraka i čvrste ćelije prepreka. Ćelije tekućine su one ćelije koje imaju barem jednu česticu u sebi, a ćelije zraka su one koje to nemaju, a u isto vrijeme nisu čvrste ćelije. U ovakvoj simulaciji vrlo je lagano simulirati prazne ćelije koje predstavljaju neki plin jer ih možemo samo preskočiti kod izračuna simulacije jer nemaju čestice u sebi i njihova svojstva se neće mijenjati. Simulacija prepreka se radi na način da na njihovu površinu postavimo čvrste ćelije u koje čestice ne mogu ući, pa time niti fluid ne može teći u njih.

Veza između čestica i ćelija je ta da u svakom koraku simulacije brzine pojedinih čestica se na određen način prebacuju u ćelije, a onda se pomoću određenih matematičkih operacija nad podatcima iz ćelija simulira ponašanje fluida. Na kraju nove brzine ser prenesu natrag na čestice.

Problem kod ove metode je da kod koraka prebacivanja brzina natrag na čestice gubimo stare orijentacije smjerova gibanja pojedinih čestica. To se događa zato što se kod izračuna novih brzina radi na razini jedne ćelije i kao rezultat dobivamo samo jedan vektor orijentacije brzine koji onda postavljamo kao smjer brzine svim česticama unutar te ćelije. To rezultira glatkim protokom fluida, ali ipak na cijenu manje preciznosti simulacije jer jedna ćelija ima puno manje stupnjeva slobode od ukupnog zbroja stupnjeva slobode koje imaju sve čestice unutar ćelije zajedno.

Zbog te limitacije uvodi se još jedna metoda koja proširuje PIC metodu, a to je FLIP metoda (engl. Fluid Implicit Particle) koja zadržava određen dio smjerova orijentacija čestica unutar ćelija koje se simuliraju. [4]

2.3.2. FLIP metoda

FLIP metoda pokušava riješiti problem gubljenja stupnjeva slobode pojedinih čestica tako da prije prebacivanja brzina na ćelije privremeno sprema prijašnje vrijednosti tih brzina. Nakon izračuna novih brzina, koja bi se u PIC metodi odmah prebacile na čestice, FLIP metoda na čestice dodaje razliku starih i novih brzina ćelija unutar kojih su sadržane i time čuva dio njihove prijašnje orijentacije.

Problem FLIP metode je da ipak stvara malo previše kaosa i da se njenim korištenjem gube određeni aspekti i ponašanja fluida.

U praksi je najbolje koristiti kombinaciju FLIP i PIC metode jer se time dobiva najbolje iz obije metode. Moguće je podešavati omjer utjecaja FLIP i PIC metoda i za svaku primjenu odabrati najbolji omjer koji najbolje odgovara potrebama simulacije. [4]

3. Grafički programski pogon

Za prikaz ove simulacije fluida nije korišten postojeći grafički pogon kao Unity ili Unreal Engine nego je izrađen vlastiti grafički pogon za 2D grafiku. Grafičko sučelje i programski jezik korišteni za izradu ovog grafičkog pogona su Vulkan i C++.

Grafičko sučelje Vulkan je bilo izabrano za izradu ovog grafičkog pogona zato jer je za njegovo korištenje potrebno naučiti mnogo stvari o računalnoj grafici i načinu na koji grafički pogoni rade u pozadini.

3.1. Vulkan grafičko sučelje

Vulkan je relativno novo grafičko sučelje napravljeno od Khronos Grupe u 2016. godini. Glavna značajka Vulkana je mogućnost za vrlo visoke performance zahvaljujući velikoj kontroli nad grafičkim hardverom koju Vulkan pruža programerima. Neke od drugih značajki koje Vulkan pruža su prenosivost na velikom broju operacijskih sustava i platformi, višedretvenost i potpuna kontrola nad memorijom.

No sva ta kontrola koju Vulkan pruža dolazi s cijenom velike složenosti i visokom barijerom ulaska zato jer je malo toga postavljeno za programera unaprijed i većina je na programeru da sam podesi sve što je potrebno, iako on to možda niti neće koristiti.

Za prvi trokut u Vulkanu potrebno je napisati oko 1000 linija koda, a u njih spada postavljanje validacijskih slojeva, povezivanje fizičkog i virtualnog uređaja, postavljanje grafičkog cjevovoda, postavljanje spremnika (engl. buffer) za slike i komande. [6]

3.2. Korištene biblioteke

Kod izrade grafičkog pogona osim Vulkana korištene su i neke druge grafičke i matematičke biblioteke kao što su GLFW, GLM i Dear ImGui koje su znatno olakšale određene dijelove implementacije.

3.2.1. Grafička biblioteka GLFW

GLFW (engl. Graphics Library Framework) je grafička biblioteka za programski jezik C++ koji služi kao programsko sučelje koje povezuje druga grafička sučelja kao što su Vulkan i OpenGL s operacijskim sustavom. Time omogućuje standardne funkcionalnosti kao što su komunikacija s mišem i tipkovnicom, povezivanje više zaslona i stvaranje grafičkih resursi kao što su sjenčari i razni spremnici. [7]

3.2.2. Grafička biblioteka GLM

GLM je matematička biblioteka za programski jezik C++ koja omogućuje mnoge matematičke operacije koje su potrebne u računalnoj grafici kao što su naprimjer operacije s vektorima i matricama. [8]

3.2.3. Grafička biblioteka Dear ImGui

Dear ImGui je grafička biblioteka otvorenoga koda koja se može koristiti u većini grafičkih sučelja uključujući Vulkan. Dear ImGui omogućava jednostavno stvaranje i kontrolu osnovnoga korisničkog sučelja. Neke od funkcionalnosti za koje je Dear ImGui bio korišten su prikazivanje opisnih informacija i informacija za ispravljanje pogrešaka tijekom razvoja kao što su broj slika u sekundi, broj čestica u simulaciji i pozicije određenih čestica. Dear ImGui je isto omogućio da u simulaciji postoje tipke i kontrola mišem za ostvarivanje interakcije s korisnikom. [9]

Dear ImGui je lagan i intuitivan za korištenje nakon što se poveže s grafičkim pogonom, no sama integracija u postojeći grafički pogon nije trivijalna i kod spajanja Dear ImGui sučelja s ovim grafičkim pogonom susretnuti su određeni problemi. To je zato jer ne postoji opsežna dokumentacija o tome kako povezati Dear ImGui u postojeći pogon na ispravan način, ali nakon dovoljno pokušaja Dear ImGui je uspješno bio integriran u ovaj grafički pogon i sve njegove mogućnosti su postale dostupne kroz grafički pogon.

3.3. Korisničko sučelje

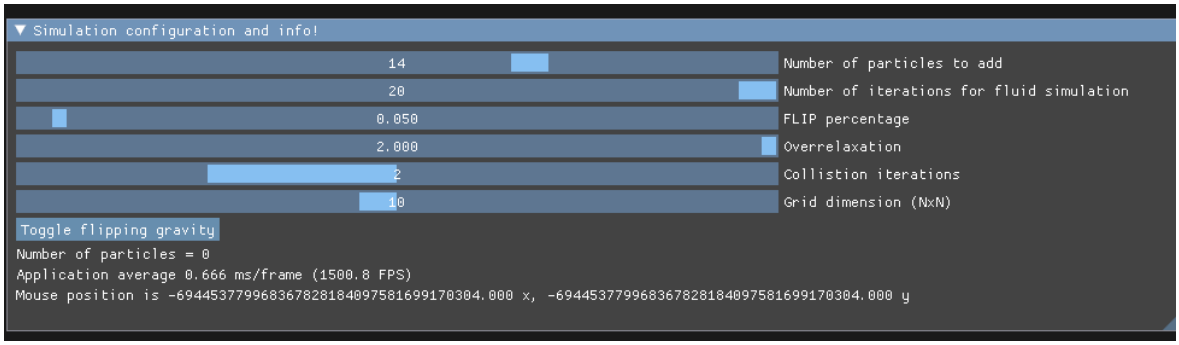
Koristeći grafičku biblioteku Dear ImGui napravljeno je jednostavno korisničko sučelje koje pruža određene informacije o simulaciji i element interaktivnosti s korisnikom koje se može vidjeti na slici 3.1.

Informacije koje korisničko sučelje pruža su broj slika u sekundi (engl. Frames Per Second, FPS), broj čestica u simulaciji i koordinate trenutne pozicije miša.

Pritiskom desne tipke miša bilo gdje prozoru simulacije dodaju se nove čestice na mjesto gdje pokazuje miš. Broj čestica koje će biti stvorene ovisi o parametru koji je moguće dinamički podešavati unutar simulacije.

Moguće je podešavati i određene parametre simulacije kao što su FLIP omjer, koji specificira koliki će biti omjer FLIP, a koliko PIC metode. Isto je tako moguće podešavati i parametar prekomjerne relaksacije (engl. Over relaxation) koji će biti objašnjen u kasnijim poglavljima.

Dimenzije mreže koja se koristi kod simulacije je isto moguće podešavati dinamički i time se najbolje vide razlike u performanci koje određene veličine mreža imaju. No više o rezultatima u kasnijem poglavlju.



Slika 3.1 Korisničko sučelje unutar simulacije

3.4. Programsko okruženje

Razvoj programskog pogona i simulacije rađen je primarno na Linuxu, preciznije na Ubuntu 22.04 inačici Linuxa koristeći CLion od JetBrains-a kao uređivač koda. Projekt je moguće kompilirati i pokrenuti koristeći CMake što omogućava dobru prenosivost između različitih okruženja.

3.5. Struktura grafičkog pogona

Grafički pogon je izrađen prateći lekcije na YouTube-u korisnika Brendan Galea i slijedi njegovu strukturu grafičkoga pogona. [10] Grafički pogon je podijeljen na više C++ klasa od

kojih svaki ima svoju zadaću i svaka je sastavljena od datoteke opisnika (engl. header) i datoteke s kodom.

3.5.1. Klasa prozor

Prozor je klasa koja je zadužena da čuva podatak o veličini prikaza na kojem se prikazuje izlaz iz grafičkog pogona, da pazi o tome je li došlo do promjene veličine prikaza i da o tome obavijesti ostale dijelove grafičkog pogona. Klasa prozor sadrži i pokazivač na strukturu *GLFWwindow* koja predstavlja prozor u grafičkoj biblioteci GLFW koja je onda zadužena da se dogovori s operacijskim sustavom kako taj prozor prikazati na fizičkom zaslonu.

3.5.2. Klasa uređaj

Uređaj (engl. Device) je klasa koja u sebi sadrži mnoge članske varijable i funkcije koje su zaduženje za postavljanje i povezivanje grafičkog sučelja Vulkan s fizičkim grafičkim uređajem. U Vulkanu ne postoje neke predodređene vrijednosti i zato je na programeru da sam napiše sav taj kod. Za ovaj rad taj kod je direktno preuzet iz Vulkan Tutorial [7] tečaja zato što je dovoljno dobar za potrebe izrade grafičkog pogona i simulacije.

3.5.3. Klasa cjevovod

Cjevovod (engl. Pipeline) je klasa s kojom se definira grafički protočni sustav grafičkog pogona. Grafički protočni sustav sastoji se od 5 osnovnih faza, od kojih se neke predefinirane kod stvaranja cjevovoda, a druge se mogu dinamički programirati. Glavni razlog zašto se grafički cjevovod dijeli u više jednostavnih, međusobno neovisnih faza je da bi se prolazi kroz grafički cjevovod mogli odvijati paralelno na grafičkom uređaju, što jako ubrzava izračunavanje slike.

Prva faza je sastavljanje geometrije iz ulaznih vrhova. Ta faza je unaprijed predefinirana. Na primjer jedna od stvari koja se definira je u koji oblik treba spojiti ulazne vrhove, jesu li to trokuti ili neki drugi oblici.

Druga faza je sjenčar vrhova (engl. Vertex Shader) čija je zadaća transformirati vrhove koristeći zadane matrice. U njemu se obično radi transformacija pogleda i perspektive. Njega je moguće programirati u sjenčarskim jezicima. U ovom grafičkom pogonu korišten je GLSL sjenčarski jezik.

```
#version 450

layout (location = 0) in vec2 position;
layout (location = 1) in vec3 color;

layout (push_constant) uniform Push{
    mat2 transform;
    vec2 offset;
    vec3 color;
} push;

void main(){
    gl_Position = vec4(push.transform * position + push.offset, 0.0, 1.0);
}
```

Slika 3.1 Sjenčar vrhova u grafičkom pogonu

Na slici 3.2 može se vidjeti kod sjenčara vrhova koji je korišten u grafičkom protočnom sustavu koji kao ulaz na lokaciji 0 prima jedan dvodimenzionalni vektor koji predstavlja poziciju, a na lokaciji 1 trodimenzionalni vektor koji predstavlja boju.

Pošto se radi o 2D grafičkom pogonu, još nije bilo potrebe raditi 3D transformacije pogleda i implementirati mogućnost za kameru i projekciju nego je dovoljna opcija za definiranje konstanti guranja (engl. Push Constant) koja omogućuje relativno jednostavnu i performantnu opciju za pomicanje objekata u 2D sceni. Za svaki poligon uz njegovu poziciju i boju šaljemo dodanu konstantu guranja u kojoj definiramo poziciju, pomak i boju trenutnog poligona. To nam omogućava da koristimo istu geometriju modela za više objekata u sceni koji nisu na istim pozicijama ili nemaju iste boje jer u sjenačaru automatski računamo novu poziciju poligona bazirano na njegovoj konstati guranja koja je specificirana na aplikacijskoj razini.

Treća faza je rasterizacija koja je isto predefinirana unaprijed, a u njoj se transformirana geometrija pretvara u diskretne jedinice koji će kasnije postati slikovni elementi na ekranu.

Četvrta faza je sjenčar fragmenata (engl. Fragment Shader) koji je isto kao sjenčar vrhova moguće programirati u sjenčarskom jeziku GLSL. Njime je moguće svakom slikovnom elementu na slici kontrolirati boju na sofisticiran način.

```
#version 450

layout (location = 0) out vec4 outColor;

layout (push_constant) uniform Push{
    mat2 transform;
    vec2 offset;
    vec3 color;
} push;

void main(){
    outColor = vec4(push.color, 1.0f);
}
```

Slika 3.1 Sjenčar fragmenata u grafičkom pogonu

Na slici 3.3 možemo vidjeti sjenčar fragmenata koji je korišten za dodjeljivanje boje svakom slikovnom elementu. U njemu samo na svaki slikovni element primjenjujem boju koja je definirana u grafičkom cjevovodu u strukturi podataka *Push* u trodimenzionalnom vektoru na zadnjem mjestu.

Zadnja faza grafičkog protočnog sustava je miješanje boja u kojem dolazi do miješanja boja slikovnog elementa koji je proizašao iz sjenčara fragmenata i slikovnog elementa koji je već bio u prijašnjoj slici.

Izlaz iz grafičkog cjevovoda je lista RGB boja koja predstavlja jednu sliku koja se sastoji od određenog broja slikovnih elemenata koje će sada biti moguće prikazati na fizičkom zaslonu.

3.5.4. Klasa zamjenski lanac

Zamjenski lanac (engl. Swap Chain) služi za sinkroniziranje slika koje izlaze iz grafičkog protočnog sustava s fizičkim zaslonom. Da bi se osigurao gladak prijelaz svake slike važno je da izlaz iz grafičkog cjevovoda bude sinkroniziran s brzinom iscrtavanja (engl. Refresh Rate) zaslona. Jer ako grafički cjevovod generira slike pre sporo onda možemo primijetiti zastajkivanje slike, a ako grafički cjevovod pre brzo generira slike koje ne možemo prikazati onda nepotrebno trošimo resurse koje smo mogli koristiti za nešto drugo.

Zamjenski lanac taj problem sinkronizacije rješava tako da uvijek ima dvije ili više slika u letu (engl. Frame in Flight) i pomoću njih uvijek može dati ažuriranu sliku zaslonu kada je to potrebno.

3.5.5. Klasa iscrtavatelj

Iscrtavatelj (engl. Renderer) u ovom grafičkom pogonu služi za povezivanje grafičkog dijela pogona s aplikacijskim. On pruža sučelje preko kojega omogućuje davanje naredbi grafičkom pogonu da crta objekte koji se s aplikacijske razine specificiraju kao lista 2D vrhova i boja. Struktura podataka pomoću koje se to radi se naziva spremnik naredbi (engl. Command Buffer) u koji je moguće pohraniti više objekata za nacrtati i tek na kraju dok smo na aplikacijskoj strani izgenerirali i ažurirali sve što smo trebali možemo dati naredbu grafičkom pogonu da nacrta sve u spremniku odjednom. Prednost toga je da znatno smanjuje vrijeme koje je potrebno za iscrtavanje scene jer nije potrebno za svaki objekt ili svakih par objekata prolaziti kroz grafički protočni sustav za crtanje.

3.5.6. Klasa model

Klasa Model predstavlja jedan geometrijski model objekta u grafičkom pogonu. Naprimjer za više objekata koji imaju geometriju kruga dovoljno je napraviti samo jedan model pomoću kojega će se prikazivati svi takvi objekti, a prikaz objekata će samo ovisiti o konstantama kao što su pozicija, rotacija, skala i boja određenog objekta. To se radi zbog memorijskih i performantnih razloga jer često imamo situaciju da imamo više istih geometrijskih modela u sceni koji se razlikuju samo pod skali ili rotaciji pa onda nema smisla za svaki spremati posebne vrhove.

3.5.7. Klasa igraći objekt

Igraći objekti (engl. Game Objects) su kao i ostalim grafičkim pogonima osnovni element kojima korisnik i korisnički dio aplikacije barata. Sve što želimo iscrtati na zaslonu pomoću grafičkog pogona definiramo pomoću igraćih objekata. Svaki igraći objekt ima neke osnovne varijable kao što su identifikator, pozicija, rotacija, skala, model, boja, brzina i druge s kojima je na lagan način moguće kontrolirati i prikazivati objekte bez brige o samom grafičkom pogonu i način na koji ih on iscrtava.

3.5.8. Klasa aplikacija

Aplikacija je korisnički dio ove simulacije koja korisniku omogućuje da koristi grafički pogon preko sučelja na jednostavan način bez da treba znati sve implementacijske detalje grafičkog pogona.

U aplikacijskom dijelu korisnik može definirati listu objekata koje će grafički pogon nacrtati. Korisnik onda pristupa i dinamički mijenja pozicije, veličine i boje objekata u aplikaciji, a grafički pogon ih ažurira na zaslonu.

Aplikacijski dio grafičkog pogona se koristi u simulaciji da prikazivanje rezultata. Ideja je da simulacijski aplikacije dio radi neovisno o grafičkom pogonu i grafički pogon da radi neovisno o simulacijskom djelu. Jedina poveznica koju imaju simulacija i grafički pogon je sama aplikacija.

4. Implementacija simulacije fluida

Ova simulacija je podijeljena u dva glavna dijela, simulator fizike (engl. Physics System) i simulator fluida (engl. Fluid Solver). Svaki korak fizike se određuje više puta u sekundi, točnije za svaku sliku koja se prikazuje i time se dobiva dojam kretanja pod uvjetom da imamo minimalno 24 slike po sekundi. U ovoj simulaciji cilj je imati više od 60 slika po sekundi zbog razloga koji će biti navedeni u kasnijim poglavljima.

4.1. Simulacija čestica

U ovoj simulaciji čestice su modelirane kao krugovi zato što je to najjednostavniji oblik za određivanje sudara i općenito primjenjivanje fizike. U grafičkom pogonu sve čestice su igraći objekti (engl. Game Object) i pogon ih tretira kao sve ostale objekte koji se prikazuju u njemu.

Jedina stvar koja je potrebna za potpun prikaz čestice je njezin centar i njezin radijus što značajno olakšava stvari kod izračuna simulacije i samog prikaza. Osim radijusa i centra svaka čestica ima svoju brzinu, prijašnju brzinu, akceleraciju, boju, poziciju u mreži i tip.

4.2. Simulator fizike

Ova klasa služi da prije iscrtavanja slike primjenjuje fiziku na sve objekte unutar grafičkog pogona. Po sadašnjim postavkama simulacije na sve objekte primjenjuje se gravitacija, a na sve čestice koje predstavljaju fluid dodatno se primjenjuje i fizika za njegovo simuliranje. Simulator fizike za svaku sliku ažurira pozicije svih čestica na temelju njihovih brzina koristeći postupak postepenog integriranja brzine pravocrtnog gibanja. Jednadžbe koje se za to koriste se mogu vidjeti na slici 4.1. Ostatak fizike delegira simulatoru fluida koji se brine da taj dio simulacije radi kako bi trebao.

Dulje su dane formule koje predstavljaju rješenje brzine i pozicije sustava koji se giba jednolikim ubrzanim gibanjem. [11]

$v = v_0 + at$

$s = s_0 + v_0t + \frac{1}{2}at^2$

```
53          for (auto &gameObject : RocketGameObject& : gameObjects) {
54
55              if (gameObject.gravityApplied) {
56                  gameObject.acceleration = gravity;
57              }
58
59              glm::vec2 position = gameObject.transform2d.translation;
60              glm::vec2 last_position = gameObject.last_position;
61              const glm::vec2 last_update_move = position - last_position;
62              const glm::vec2 new_position = position + last_update_move +
63                                             (gameObject.acceleration - last_update_move * (1.0f/deltaTime)) *
64                                             (deltaTime * deltaTime);
65              gameObject.last_position = position;
66              gameObject.transform2d.translation = new_position;
67          }
68
69      }
```

Slika 4.1 Isječak koda simulatora fizike za ažuriranje čestica

4.3. Mreža

Mreža u ovoj simulaciji ima dvije uloge. Jedna uloga je rješavanje kolizija između čestica koja je potrebna da bi osigurala da čestice ne mogu ući jedna u drugu što bi poremetilo prikaz simulacije jer čestice su jedino što se vidi prilikom prikaza. Druga uloga mreže je da su u njoj pohranjene vrijednosti koje su potrebne za simulaciju pojedinih ćelija nad ćelijama se vrše operacije prebacivanja brzine s i na čestice.

4.4. Rješavanje sudara

U ovoj kao i većini simulacija koje pokušavaju simulirati neku interakciju između čestica potrebno je riješiti problem detekcije i rješavanja sudara. U ovoj simulaciji može doći do dvije vrste sudara. Jedna vrsta je sudar čestice sa zidom, a drugi su međusobni sudari između čestica.

Obije vrste sudara se rješavaju uz pomoć mreže zato što velika većina potencijalnih sudara koje teoretski možemo provjeriti neće rezultirati sudarom i time bi radili nepotrebno velik broj provjera što bi rezultiralo manjim performansama.

Nakon detekcije sudara ono što želimo je osigurati da čestica ne bude unutar zida ili unutar druge čestice nakon sudara. Takvo rješavanje sudara dovoljno je dovoljno dobro za daljnje simuliranje gibanja čestica fluida jer se one gibaju relativno malom brzinom.

4.4.1. Rješavanje sudara između zidova

Mreža smanjuje broj provjera koje trebamo napraviti da bi riješili sudare čestica i zidova jer unaprijed znamo koje ćelije su rubne. Dalje znamo da je dovoljno provjeriti samo čestice unutar rubnih ćelija za potencijalne sudare sa zidovima i isto tako na temelju ćelije odmah znamo koji zid je potrebno provjeriti za sudare, a ostale možemo preskočiti.

```
void Grid::resolveCollisionsWithWalls(std::vector<RocketGameObject> &gameObjects) {

    for (int i = 0; i < grid_width; i++) {
        resolveCollisionsBetweenCellAndWalls( &: gameObjects,  cell: i,  direction: 0); // top
    }
    for (int i = 0; i < grid_width; i++) {
        int cell_index = (grid_height - 1) * grid_height + i;
        resolveCollisionsBetweenCellAndWalls( &: gameObjects,  cell: cell_index,  direction: 2); // bottom
    }

    for (int i = 0; i < grid_height; i++) {
        int cell_index = i * grid_height;
        resolveCollisionsBetweenCellAndWalls( &: gameObjects,  cell: cell_index,  direction: 3); // left
    }
    for (int i = 0; i < grid_height; i++) {
        int cell_index = (i + 1) * grid_height - 1;
        resolveCollisionsBetweenCellAndWalls( &: gameObjects,  cell: cell_index,  direction: 1); // right
    }

}
```

Slika 4.2 Funkcija za rješavanje sudara sa zidovima

```
uint32_t Grid::resolveCollisionsBetweenCellAndWalls(std::vector<RocketGameObject> &gameObjects, int cell,
                                                    int direction) { // 0 - top, 1 - right, 2 - bottom, 3 - left
    for (int object_id: grid[cell].objects) {
        RocketGameObject &gameObject = gameObjects[object_id];
        float radius = gameObject.radius;
        float margin = 0.001f;
        glm::vec2 center = gameObject.transform2d.translation;
        if (direction == 2) {
            if (center.y + radius >= 1.0f - margin) {
                float distance = center.y + radius - 1.0f + margin;
                gameObject.transform2d.translation.y -= distance;
            }
        }
        if (direction == 1) {
            if (center.x + radius >= 1.0f - margin) {
                float distance = center.x + radius - 1.0f + margin;
                gameObject.transform2d.translation.x -= distance;
            }
        }
        if (direction == 0) {
            if (center.y - radius <= -1.0f + margin) {
                float distance = center.y - radius + 1.0f - margin;
                gameObject.transform2d.translation.y -= distance;
            }
        }
        if (direction == 3) {
            if (center.x - radius <= -1.0f + margin) {
                float distance = center.x - radius + 1.0f - margin;
                gameObject.transform2d.translation.x -= distance;
            }
        }
```

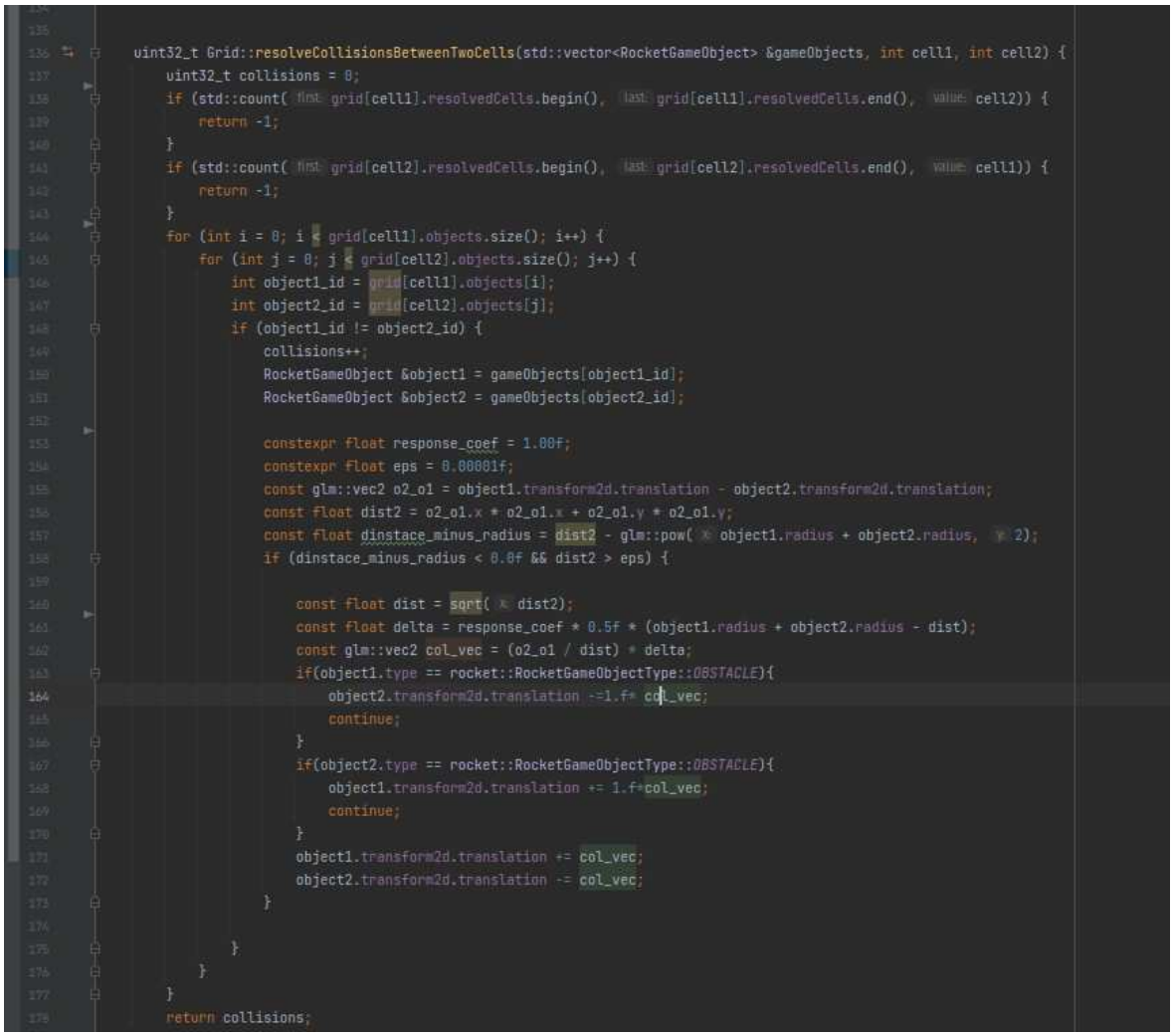
Slika 4.3 Funkcija za rješavanje sudara sa zidom unutar specifične ćelije

Funkcija sa slike 4.2 iterira kroz sve rubne ćelije i na rubnim ćelijama zove funkciju sa slike 4.3 koja provjerava je li došlo do sudara i ako je to slučaj onda u skladu s time koliko je čestica daleko ušla u zid ažurira njezinu poziciju da ne bude unutar zida. Pošto se radi o samo rubnim zidovima koji su obični pravokutnici detekcija sudara je jednostavna jer je samo potrebno provjeriti je li čestica prešla granice simulacije s x ili y osi. Izračun koliko je čestica ušla unutar zida je isto jednostavna, rješenje je samo koliko je x ili y koordinata manja od nula ili veća od jedan ovisno o zidu o kojem se radi.

4.4.2. Rješavanje sudara između čestica

Mreža olakšava i rješavanje međusobnih sudara između česticama zato što, uz pretpostavku da su čestice manje od jedne ćelije, znamo da čestica čije je središte unutar jedne ćelije može biti u sudaru samo s česticama iz 8 susjednih ćelija i česticama unutar svoje ćelije. To znatno smanjuje broj provjera koje trebamo obaviti.

Naravno omjer veličine čestica i veličine ćelija znatno ovisi o performansama koje ćemo dobiti ili izgubiti koristeći mrežu zato jer držanje ćelija ažuriranima oduzima određeno vrijeme pa više ćelija nije uvijek bolje.



```
uint32_t Grid::resolveCollisionsBetweenTwoCells(std::vector<RocketGameObject> &gameObjects, int cell1, int cell2) {
    uint32_t collisions = 0;
    if (std::count( first: grid[cell1].resolvedCells.begin(), last: grid[cell1].resolvedCells.end(), value: cell2)) {
        return -1;
    }
    if (std::count( first: grid[cell2].resolvedCells.begin(), last: grid[cell2].resolvedCells.end(), value: cell1)) {
        return -1;
    }
    for (int i = 0; i < grid[cell1].objects.size(); i++) {
        for (int j = 0; j < grid[cell2].objects.size(); j++) {
            int object1_id = grid[cell1].objects[i];
            int object2_id = grid[cell2].objects[j];
            if (object1_id != object2_id) {
                collisions++;
                RocketGameObject &object1 = gameObjects[object1_id];
                RocketGameObject &object2 = gameObjects[object2_id];

                constexpr float response_coef = 1.00f;
                constexpr float eps = 0.00001f;
                const glm::vec2 o2_o1 = object1.transform2d.translation - object2.transform2d.translation;
                const float dist2 = o2_o1.x * o2_o1.x + o2_o1.y * o2_o1.y;
                const float dinstace_minus_radius = dist2 - glm::pow( x: object1.radius + object2.radius, y: 2);
                if (dinstace_minus_radius < 0.0f && dist2 > eps) {

                    const float dist = sqrt( x: dist2);
                    const float delta = response_coef * 0.5f * (object1.radius + object2.radius - dist);
                    const glm::vec2 col_vec = (o2_o1 / dist) * delta;
                    if(object1.type == rocket::RocketGameObjectType::OBSTACLE){
                        object2.transform2d.translation -=1.f* col_vec;
                        continue;
                    }
                    if(object2.type == rocket::RocketGameObjectType::OBSTACLE){
                        object1.transform2d.translation += 1.f*col_vec;
                        continue;
                    }
                    object1.transform2d.translation += col_vec;
                    object2.transform2d.translation -= col_vec;
                }

            }
        }
    }
    return collisions;
```

Slika 4.1 Funkcija koja rješava sudare između dvije ćelije

Na slici 4.4 vidi se funkcija s kojom se rješavaju sudari između čestica u dvije ćelije. Potrebno je proći kroz sve čestice unutar prve ćelije i provjeriti je li došlo do sudara sa svakom od čestica iz druge ćelije. Sudar se određuje tako da znamo da su sve čestice krugovi, a provjera je li došlo do sudara između dva kruga se radi tako da provjerimo je li udaljenost njihovih središta bliža od zbroja radijusa.

Kada detektiramo da je došlo do sudara računamo vektor presjeka sudarenih čestica i svaku od čestica pomičemo za polovicu toga vektora, svaku u suprotnom smjeru. Time garantiramo da čestice neće biti jedna u drugoj nakon rješavanja sudara.

Jedan od problema koji nastaje ovim pristupom je da unutar jednog koraka simulacije računamo sve sudare čestica i neovisno o drugim čestica im ispravljamo poziciju što može dovesti do situacije gdje pomaknemo čestice A i B koje su bile u sudaru i onda jedna od njih nakon ispravljanja pozicije završi unutar čestice C kroz koju smo već bili iterirali i zaključili da nije bila u sudaru. Zbog toga u ovoj simulaciji rješavanje sudara s drugim česticama radi se više puta da bi se probao smanjiti taj efekt. Osim više iteracija ono što smanjuje ovaj efekt je iscrtavanje što više slika po sekundi i ograničavanje brzine. To je zato što prilikom ažuriranja jednog koraka simulacije, koji se radi jedanput prije iscrtavanje slike, čestice će napraviti manji pomak i to će umanjiti presjek čestica prilikom sudara. Podrazumijeva se pretpostavka da je brzina čestica ograničena na razumni iznos.

4.5. Ćelija

U ovoj simulaciji postoje tri tipa ćelija. Prvi tip su ćelije fluida koje se određuju pri svakom koraku simulacije na temelju toga ima li dotična ćelija barem jednu česticu u sebi, ako ima onda ta ćelija postaje ćelija fluida i nad njom kasnije primjenjujemo FLIP algoritam. Ćelije za koje se utvrdi da nemaju niti jednu česticu unutar sebe postaju ćelije tipa zrak i njih FLIP algoritam preskače.

Čvrste ćelije tipa prepreke su sve ćelije na rubu simulacije koje predstavljaju zidove i ćelije koje sadrže prepreke. One se ponašaju tako da u njih fluid ne može teći i one ne mijenjaju svoj tip kroz simulaciju bez korisnikove interakcije dok miče prepreku. Onda nove ćelije postaju čvrste, a stare dobivaju neki od druga dva tip.

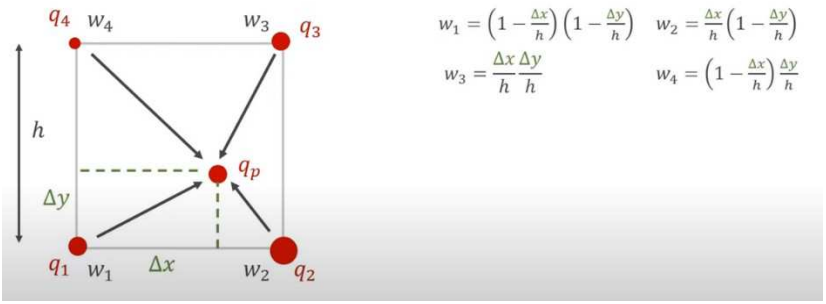
4.6. Simulator fluida

Simulator fluida sadrži instancu mreže i ima pristup česticama iz simulacije. Njegova zadaća je povezati simulaciju fluida preko mreže i stvarne vrijednosti pozicija i brzina čestica koja se ustvari prikazuju. Zbog strukture grafičkog pogona i simulacije sama mreža nema pristup instancama čestica koje grafički pogon koristi za krajnji prikaz nego samo vidi kopiju

njihovih pozicija i brzina za koje se simulator fluida brine da budu sinkronizirane. Na temelju tih podataka onda računa i ažurira vrijednosti koje će pohraniti u mrežu.

4.6.1. FLIP algoritam

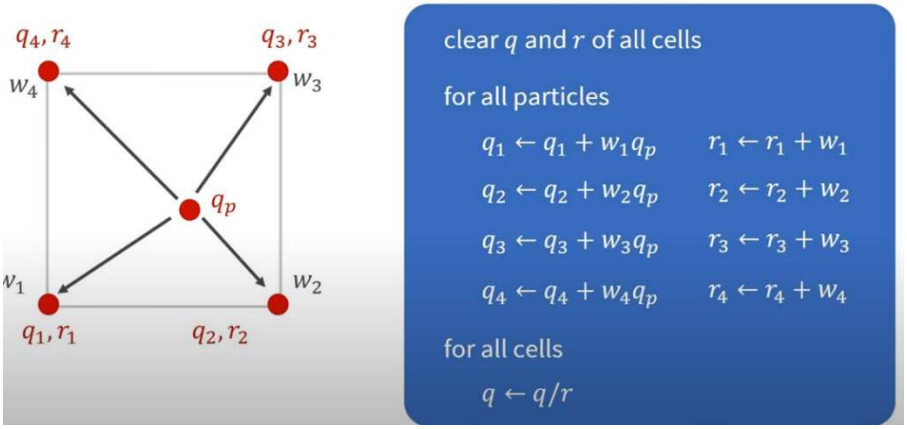
FLIP algoritam, isto kao i PIC algoritam ima tri osnovna koraka, s obzirom da je FLIP algoritam proširenje PIC algoritma jedina razlika između njih je u zadnjem koraku. Prvi korak je prenošenje brzina čestica na ćelije mreže. To se radi koristeći metodom bilinearne interpolacije. Čestica će se nalaziti negdje unutar ćelije. Ono što se radi je prijenos brzine čestice na sve kutove ćelije, ali to napraviti na temelju toga koliko je blizu određenom kutu te ćelije. Težine za svaki računaju se formulama na slici 4.5, te težine će biti potrebne i za treći korak. [4]



Slika 4.1 Računanje težina svakog kuta ćelije [4]

Jedino što je potrebno znati je centar čestice, koordinate daljnjeg lijevog kuta ćelije i visinu i veličinu ćelije, u ovoj simulaciji ćelije su uvijek kvadrati pa imaju istu visinu i duljinu

Nakon što su težine za svaki kut izračunate potrebno je prenijeti brzinu s čestice na ćeliju. To se radi pomoću algoritma na slici 4.6.

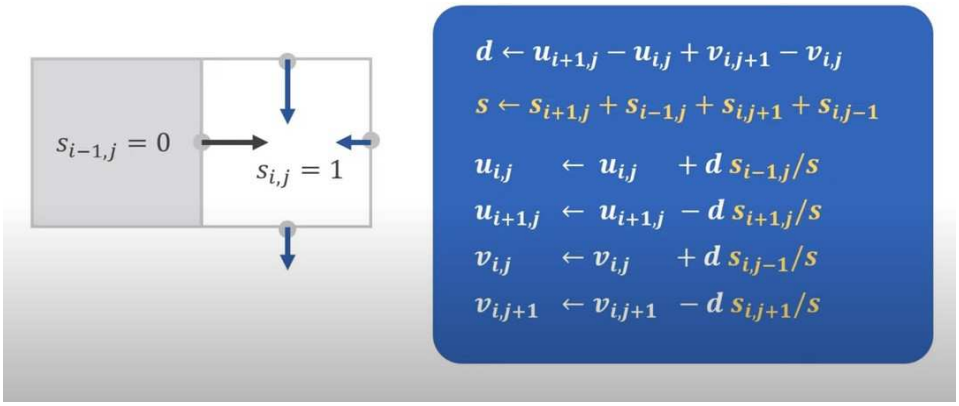


Slika 4.1 Računanje i prebacivanje brzina u ćeliju [4]

Prvo je potrebno proći kroz sve čestice unutar trenutne ćelije i na temelju izračunatih težina dodati brzine za svaki kut, oznaka q, i izračunati koliko je bila ukupna težina za određeni vrh, oznaka r. Na kraju kada se prođe kroz sve čestice unutar ćelije za svaki kut ćelije radi se dijeljenje brojave q i r. Time se dobiva brzina koju je potrebno prenijeti na taj kut ćelije.

No to nije stvarna brzina koju treba pamtiti za taj kut ćelije zato što čestice same po sebi imaju vrlo kaotično ponašanje i ništa ih još ne tjera da se ponašaju kao fluid. Zato je potrebno definirati još jedno svojstvo koje će ustvari biti zaslužno da tjera čestice da se ponašaju kao fluid, a to je svojstvo nekompresibilnosti. Za fluide kao voda to je poprilično dobra pretpostavka jer je vodu u prirodi moguće komprimirati samo u jako ekstremnim uvjetima kojima se ova simulacija ne bavi.

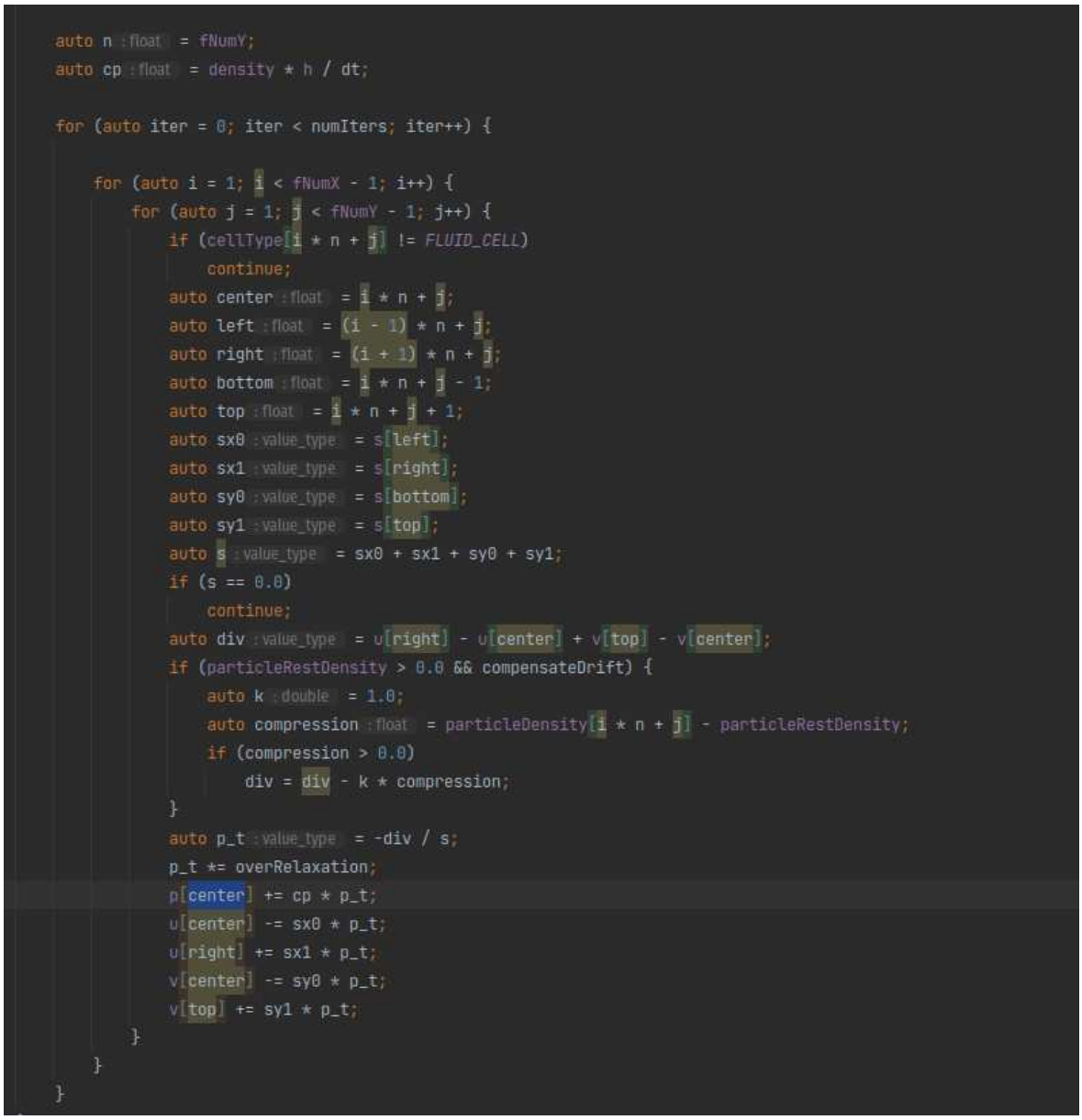
Uvjet nekompresibilnosti kaže da količina protoka koji uđe u ćeliju mora biti jednaka količini protoka koji izađe iz ćelije. To bi u praksi značilo da ako je u ćeliju ušla jedna čestica s brzinom v, onda je iz nje morala izaći druga čestica s istom brzinom, no to se ne može desiti samo od sebe nego je potrebno prepraviti brzine ćelija tako da svojstvo nekompresibilnosti bude zadovoljeno. Algoritam kojim se to postiže se može vidjeti na slici 4.7.



Slika 4.1 Izračun divergencija i protoka za svaku stranu ćelije [4]

Protok u ćeliju može ulaziti i izlaziti samo na četiri strane i potrebno je osigurati da zbroj svih protoka na te četiri strane bude jednak 0. Prvo se računa koliki je iznos protoka kojega ima viška ili manjka. To je označeno slovom d. Onda je potrebno osigurati da se ne računaju strane ćelija koje za susjeda imaju čvrstu ćeliju jer protok nije mogao doći od tamo i tamo neće moći izaći. Konačno za svaku stranu ćelije dodaje se ili oduzima suvišni protok podijeljen ukupnim brojem strana i time se osigurava da dok se svi protoci zbroje, rezultat toga će biti nula što je bio i cilj. No taj rezultat će biti 0 samo za trenutnu ćeliju koja će za postizanje tog uvjeta promijeniti susjedne ćelije koje onda u većinu slučajeva neće imati divergenciju jednaku nula. Zbog toga bi idealno bilo raditi neograničeno iteracija sve dok divergencija u svim ćelijama ne konvergira u nulu, naravno to nije moguće zbog ograničenih resursi i zato je broj iteracija ograničen i traži se neki drugi način za povećanje brzine konvergencije. [4]

Jedan od načina kojima se broj iteracija potrebne za konvergenciju smanjuje je množenje parametra divergencije d s faktorom između 1 i 2. Taj postupak se zove prekomjerna relaksacija (engl. over relaxation) koje se koristi kod smanjena broja iteracija kod rješavanja linearnih sistema iterativnim metodama. [4]



```
auto n :float  = fNumY;
auto cp :float  = density * h / dt;

for (auto iter = 0; iter < numIters; iter++) {

    for (auto i = 1; i < fNumX - 1; i++) {
        for (auto j = 1; j < fNumY - 1; j++) {
            if (cellType[i * n + j] != FLUID_CELL)
                continue;
            auto center :float  = i * n + j;
            auto left :float  = (i - 1) * n + j;
            auto right :float  = (i + 1) * n + j;
            auto bottom :float  = i * n + j - 1;
            auto top :float  = i * n + j + 1;
            auto sx0 :value_type  = s[left];
            auto sx1 :value_type  = s[right];
            auto sy0 :value_type  = s[bottom];
            auto sy1 :value_type  = s[top];
            auto s :value_type  = sx0 + sx1 + sy0 + sy1;
            if (s == 0.0)
                continue;
            auto div :value_type  = u[right] - u[center] + v[top] - v[center];
            if (particleRestDensity > 0.0 && compensateDrift) {
                auto k :double  = 1.0;
                auto compression :float  = particleDensity[i * n + j] - particleRestDensity;
                if (compression > 0.0)
                    div = div - k * compression;
            }
            auto p_t :value_type  = -div / s;
            p_t *= overRelaxation;
            p[center] += cp * p_t;
            u[center] -= sx0 * p_t;
            u[right] += sx1 * p_t;
            v[center] -= sy0 * p_t;
            v[top] += sy1 * p_t;
        }
    }
}
```

Slika 4.1 Isječak koda koji osigurava uvjet nekompresibilnosti.

Na slici 4.8 može se vidjeti da je za rješavanje uvjeta nekompresibilnosti potrebno iterirati više puta kroz sve ćelije unutar mreže i za svaku računati količinu protoka koja prolazi kroz nju. Na kraju se za svaku ćeliju ažuriraju izračunati protoci i prelazi se u sljedeću iteraciju koja će postepeno divergenciju približavati k nuli. Taj proces ubrzava množenje konstantom prekomjerne relaksacije.

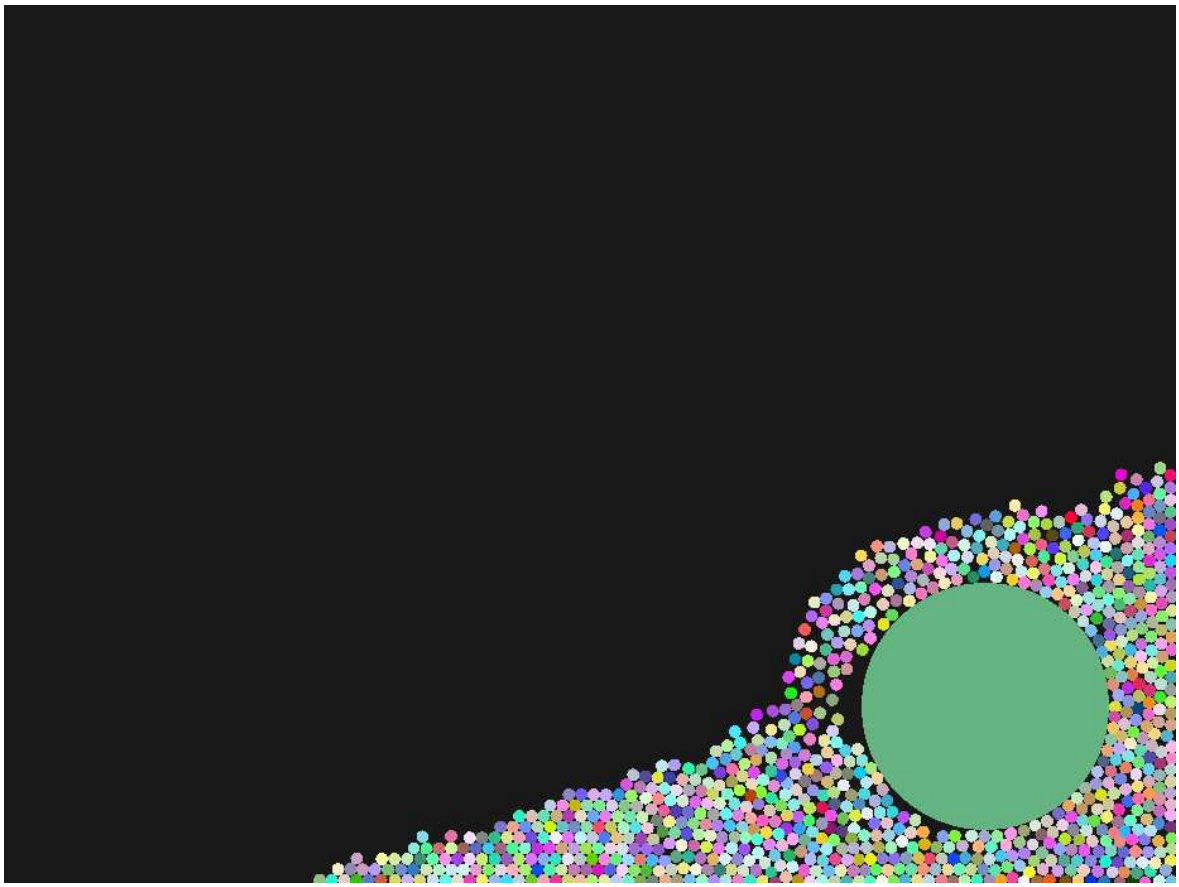
Konačni korak, ujedno i jedini u kojem se razlikuju PIC i FLIP metoda je prenošenje brzine koja je izračunata za ćelije natrag na čestice. U PIC metodi to se radi tako da se izračunata brzina za svaki kut ćelije pomnoži s prijašnje izračunatim težinama za taj kut i to se postavi

kao novu brzinu čestice. No to bi to rezultiralo da sve čestice unutar iste ćelije imaju istu orijentaciju brzine i time bi se izgubio dio podataka koje su čestice prije nosile.

FLIP metoda za razliku od PIC metode računa razliku starih brzina svakog kuta ćelije i novo izračunate brzine za svaki kut i onda dodaje tu razliku na postojeću brzinu čestice i time djelomično čuva staru brzinu i orijentaciju čestice. [4]

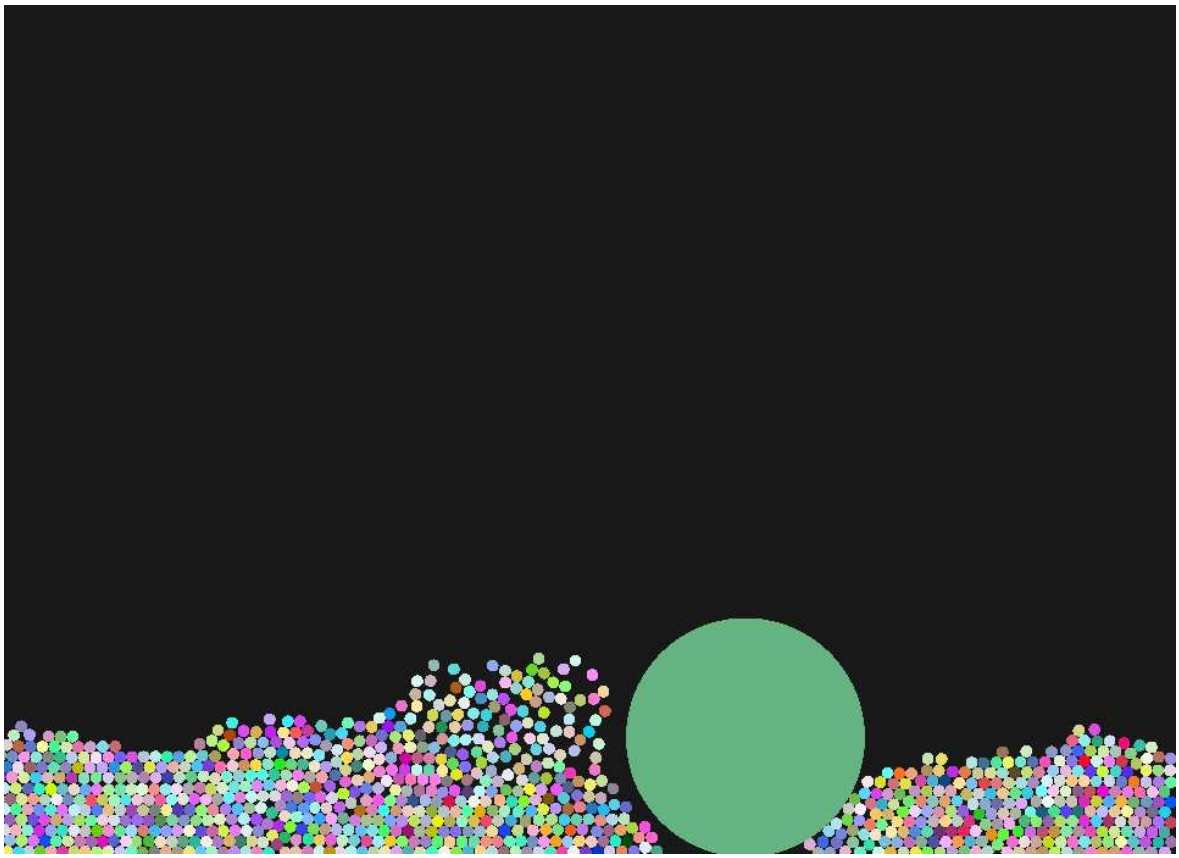
4.7. Interaktivna prepreka

Unutar simulacije postoji jedna veća čestica koja predstavlja interaktivnu prepreku koja omogućuje lakše testiranje i dodaje dodatni element interaktivnosti sa simulacijom. Prepreku je moguće pomicati mišem i postaviti je bilo gdje u simulaciji, na nju ne djeluje gravitacija pa je moguće da prepreka lebdi u zraku što je korisno kada je potrebno simulirati određene scenarije gdje je protok fluida ograničen ili nešto slično. Osim što je za prepreku gravitacija onemogućena sva ostala fizika potrebna za simulaciju fluida na prepreku djeluje isto kao i na ostale čestice.



Slika 4.1 Stacionarna interaktivna prepreka u simulaciji

Na slici 4.9 može se vidjeti velik broj manjih i jednu veliku česticu. Velika zelena čestica predstavlja prepreku i ona statički stoji u simulaciji sve dok je korisnik ne pomakne koristeći miš. Prilikom micanja prepreke ostale čestice se dinamički simuliraju, znači da je moguća direktna interakcija korisnika koji kontrolira prepreku i čestica unutar simulacije. To se može vidjeti na slici 4.10.



Slika 4.1 Interaktivna prepreka kada je korisnik pomiče u simulaciji

5. Rezultati

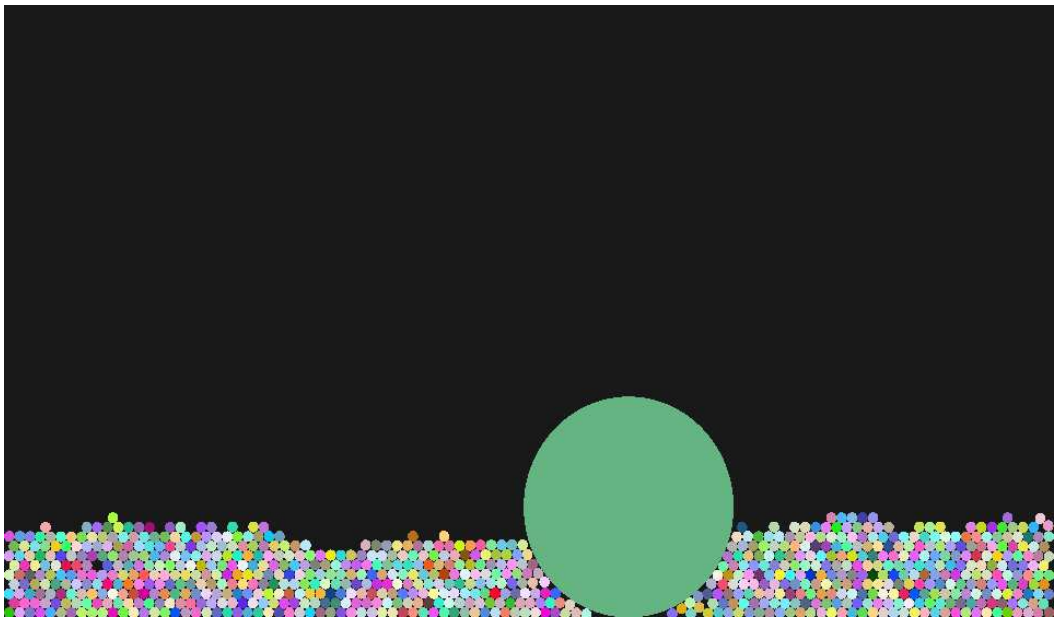
5.1. Grafički pogon

Jedan od ciljeva ovog završnog rada je bio izraditi 2D grafički pogon iz nule u programskom sučelju Vulkan. Sam grafički pogon je izrađen uz pomoć tečaja na YouTube-u koji prati Vulkan Tutorial pa grafički pogon nije izrađen potpuno samostalno iz nule, ali je u procesu izrade puno toga bilo naučeno o izradi grafičkog pogona i kako grafički pogoni rade ispod pozadine. Najteži dio kod izrade grafičkog pogona je bio integracija grafičke knjižnice Dear ImGui unutar postojećeg grafičkog pogona zato što je taj zadatak zahtijevao određeno dublje znanje o tome kako taj grafički pogon funkcionira. Integracija ne bi bila moguća bez dubljeg razumijevanja grafičkog pogona i grafičkog sučelja Vulkan.

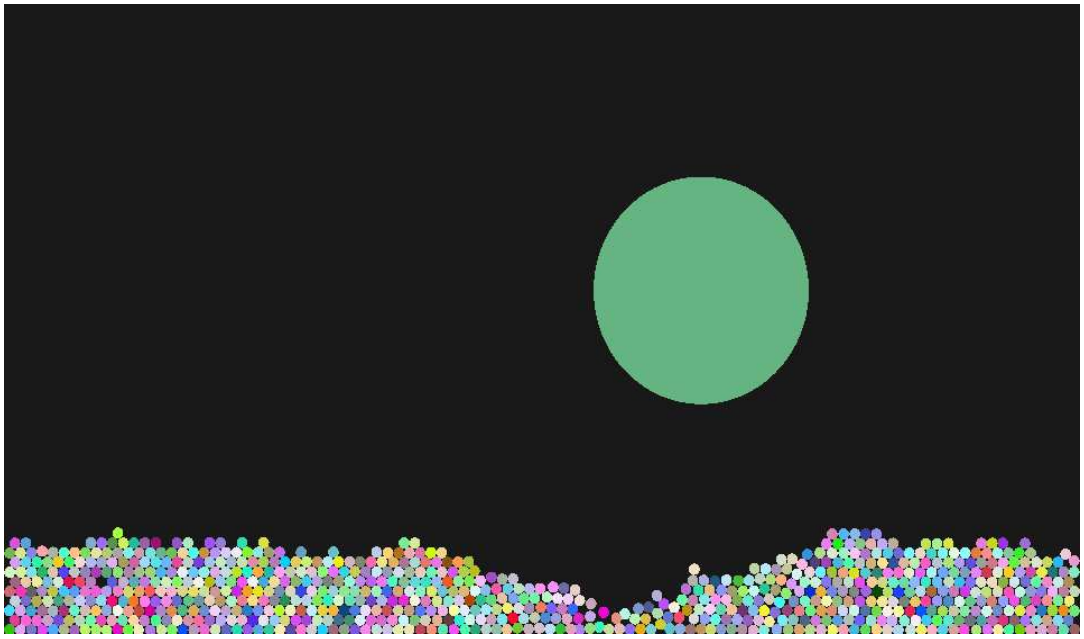
Rezultirajući grafički pogon je dovoljno dobar za prikaz ove simulacije i rezultati su zadovoljavajući. Nakon inicijalnog prikazivanja objekata u sceni i mogućnosti kontroliranja njihovih pozicija i boja više nije bilo potrebno brinuti se o grafičkom pogonu jer se sve moglo kontrolirati na aplikacijskoj razini.

5.2. Simulacija

Simulacija dobro radi kada je u stabilnom stanju i element interaktivnosti s preprekom donosi dodane mogućnosti korištenja simulacije. Na slici 5.1 može se vidjeti kako simulacija izgleda kada je prepreka postavljena unutar fluida i kada se fluid smirio i stacionaran je. Na slici 5.2 može se vidjeti kako fluid izgleda par trenutaka nakon što uklonimo prepreku iz njega, na toj slici fluid nije stacionaran.

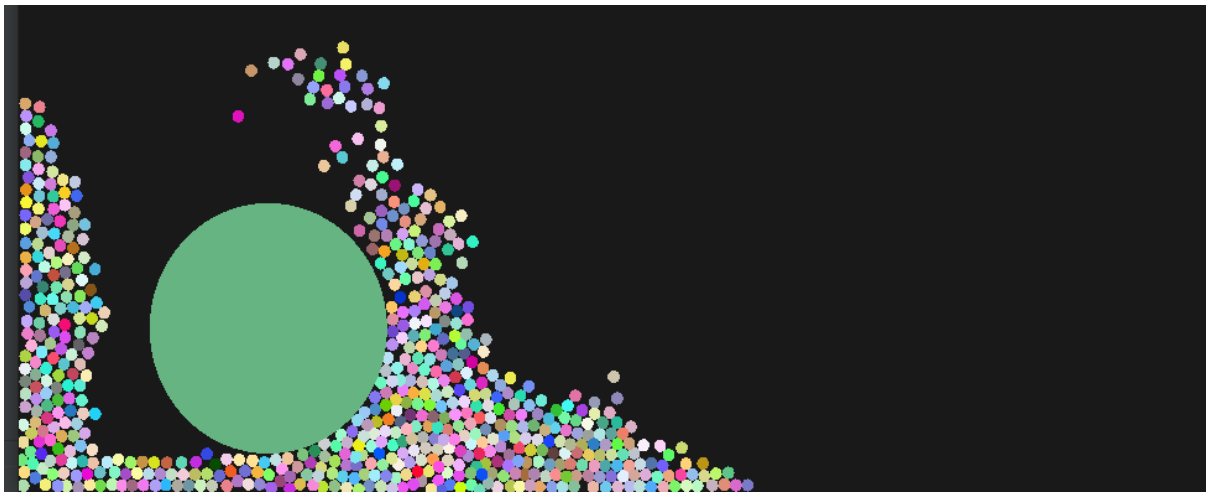


Slika 5.1 Simulacija u stacionarnom stanju s preprekom u sredini

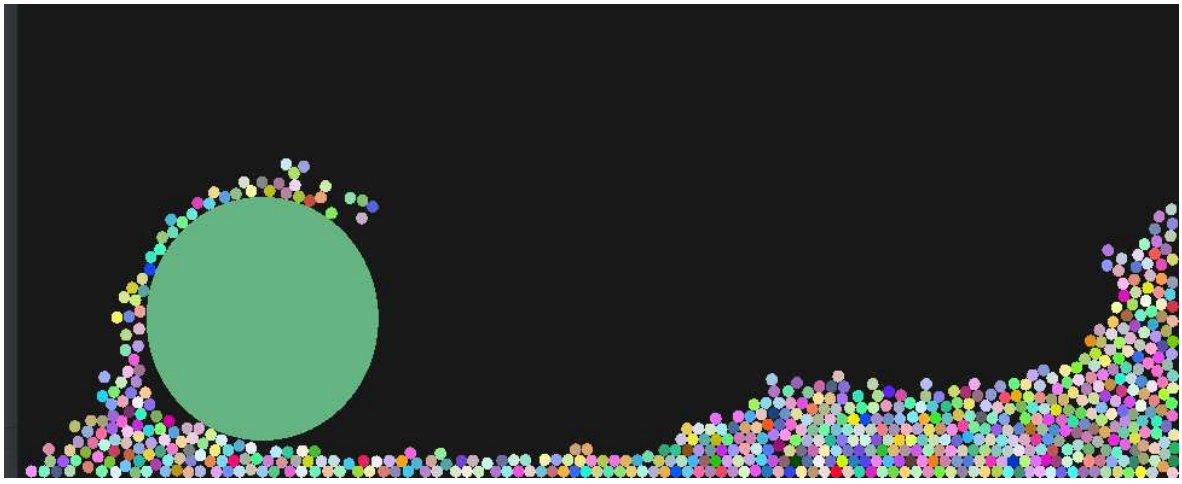


Slika 5.2 Simulacija kada prepreku maknemo iz fluida

Za svrhu testiranja simulacije dana je opcija za uključivanje periodičkog mijenjanja gravitacije prema lijevo i desno koje stvara gibanje fluida bez potrebe za dodatnom korisnikovom interakcijom.



Slika 5.3 Interakcija fluida i prepreke prilikom gibanja fluida



Slika 5.4 Interakcija fluida i prepreke prilikom gibanja fluida

Na slici 5.3 i 5.4 može se vidjeti kako izgleda kada je uključeno automatsko mijenjanje gravitacije s lijeva na desno. Na slici 5.3 visi se stanje sustava kada je gravitacija podešena na lijevo, a na slici 5.4 kada je gravitacija podešena na desno.

Moguće je vidjeti da sustav čestica ima određeno ponašanje fluida, ali zbog ograničene količine čestica koje simulacija može podnijeti za razumne performance to ponašanje nije dovoljno dobro stvarnom ponašanju fluida za neku konkretnu upotrebu. No u ovom radu je cilj bio proučiti fiziku simulacije i razumjeti kako te simulacije rade u pozadini, a ne nužno napraviti simulaciju koja će se moći koristiti za neku stvarnu primjenu.

Mijenjanje parametara simulacije kao što su korišteni postotak FLIP algoritma ili koeficijent prekomjerne relaksacije vrlo malo utječu na simulaciju i utjecaj nije očit golim okom. Pretpostavka zašto se to dešava je zato što su čestice pre velike i nije ih dovoljno da se dobiju efekti koji bi ti parametri trebali kontrolirati.

Teoretski mijenjanje postotka FLIP algoritma bi trebalo regulirati koji će postotak vektora gibanja individualnih čestica biti sačuvan, ako je taj postotak postavljen na nulu individualna gibanja čestica neće biti uopće sačuvana i sve čestice unutar jedne ćelije će se gibati u istom smjeru. To bi trebalo rezultirati glatkim uniformnim tokom unutar jedne ćelije no u ovoj simulaciji to nije vidljivo.

Još jedna pretpostavka je što su čestice veće njihovo pomicanje prilikom rješavanja sudara ima veći utjecaj na fiziku od samog simulatora fluida. Taj zaključak proizlazi iz toga da kada se kod većih čestica isključi izračun koji simulira fiziku fluida i samo se rješavaju sudari ne vidi se znatna razlika između ta dva simulirana sustava.

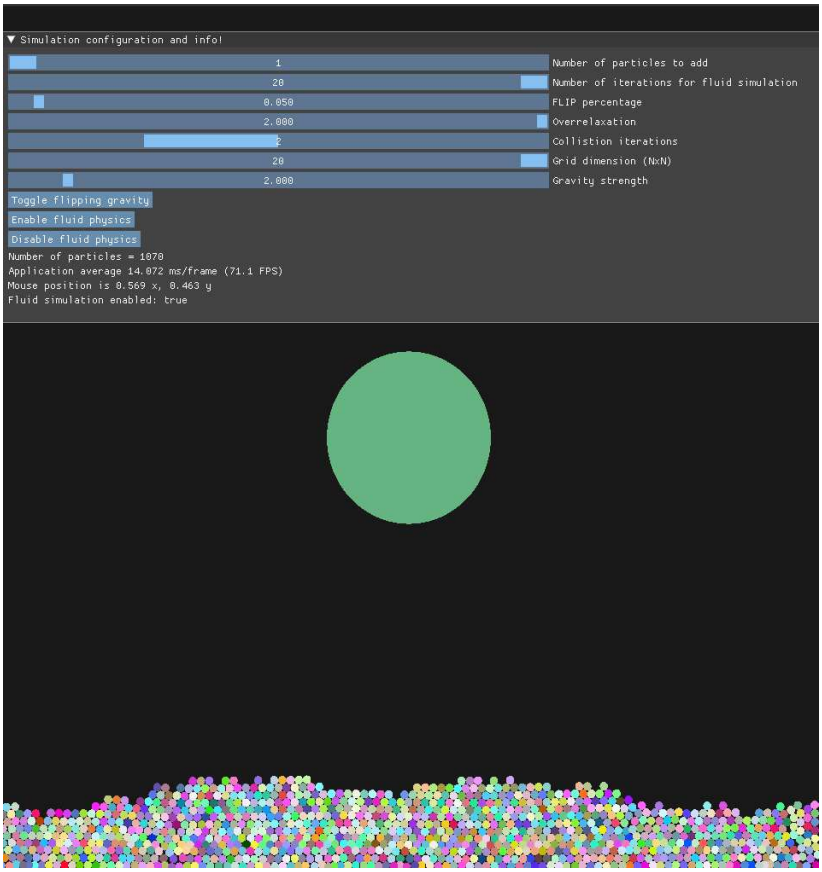
Primjena fizike fluida je vidljiva kada su čestice manje veličine, ali onda simulacija već dolazi do problema performanci i stabilnosti simulacije. Ova simulacija ne može simulirati dovoljno velik broj čestica da volumen fluida bude dovoljno velik za neko bolje testiranje.

Jedan od većih problema je da simulacija postaje nestabilna kada slike po sekundi padnu ispod 60. To se može vidjeti na slikama 5.5 i 5.6. Na slici 5.5 simulacija se vrti na 72 sličice po sekundi, dok se na slici 5.6 ona vrti na 54 sličice po sekundi i postaje nestabilna, a čestice se počinju ponašati kaotično i nepredvidivo.

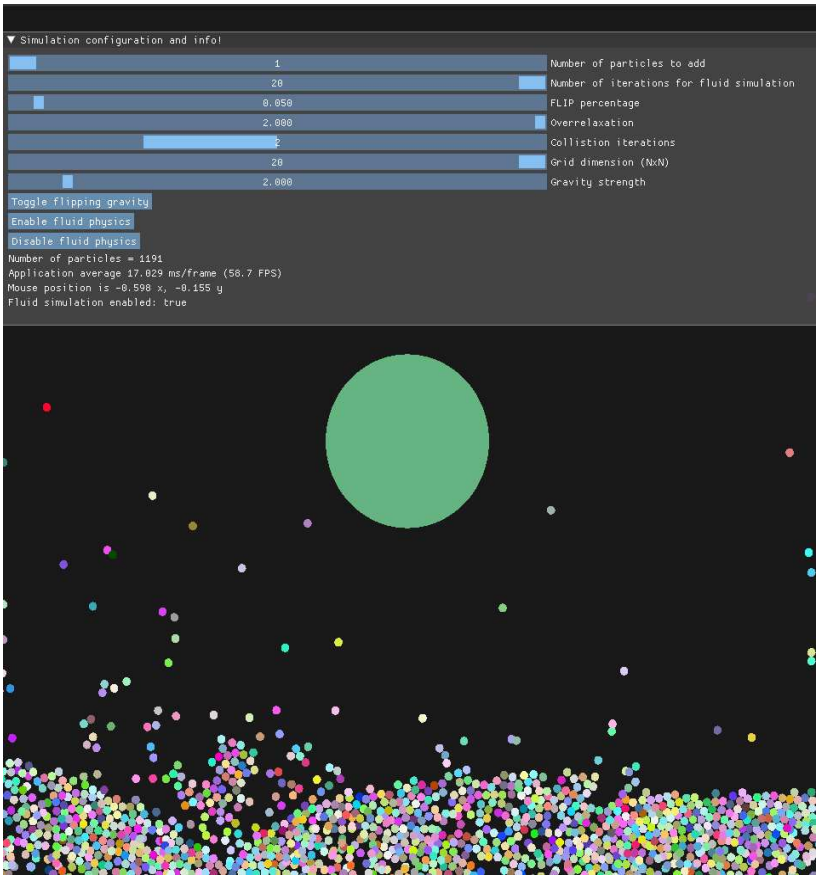
Pretpostavka zašto se to događa je u načinu rješavanja sudara koji ovisi o brzini iscrtavanja. Dok se desi situacija kada se simulacija vrti na pre malo sličica u sekundi događa se to da čestice previše ulaze jedna u drugu i potreban je veliki pomak da se to preklapanje riješi. To pak povećava šansu da te čestice koje pomičemo završe unutar neke druge čestice i onda se desi lančana reakcija gdje određene čestice dobiju kaotično kretanje.

Jedno od rješenja ovoga problema bi bila optimizacija simulacije čime bi se omogućio veći broj čestica i postavljanje razumnog limita na broj čestica koji može biti u simulaciji. Drugi, bolji način bi bio odvojiti računanje fizike od grafičkog sustava gdje onda rezultati simulacije ne bi ovisili o brzini iscrtavanja. Jedan potencijalni problem do kojega može doći ako bi se išlo tim pristupom je kada izračun fizike bude sporiji od brzine iscrtavanja. Onda bi više slika

koje grafički pogon iscrtava za simulaciju bilo isto i dobili bi efekte zastajkivanja iako imamo dovoljno velik broj slika u sekundi.



Slika 5.5 Simulacija iznad 60 FPS



Slika 5.6 Simulacija ispod 60 FPS

5.3. Performance

Zbog razloga koji je bio objašnjen u prošlom poglavlju performance su prilično bitne u ovoj simulaciji jer stabilnost i time točnost simulacije ovisi o njima. Glavni faktor koji utječe na performance je broj čestica zato što broj čestica povećava vrijeme potrebno za izračun fizike, a izračun fizike direktno utječe na performance i broj slika u sekundi.

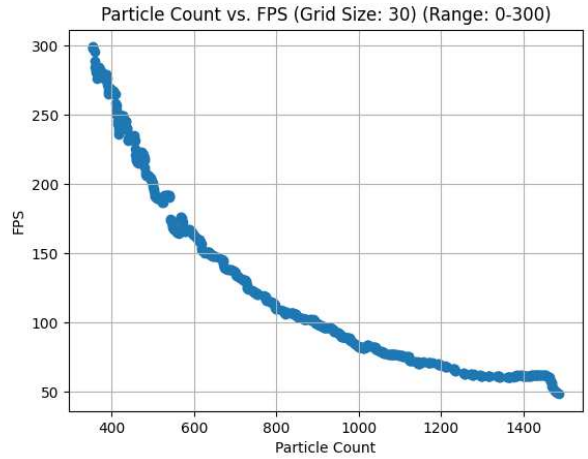
Dio koji uzima najviše vremena kod izračuna fizike je detekcija sudara zato što je u teoriji potrebno usporediti svaku česticu sa svakom drugom česticom u simulaciji, način na koji je to optimirano je koristeći mrežu koja smanjuje broj provjera koje se trebaju obaviti, ali tih provjera je još uvijek jako puno i znatno utječu na performance.

Druga stvar koja utječe na brzinu izračuna fizike je sam izračun za fiziku fluida jer je potrebno proći kroz sve ćelije i kroz sve čestice unutar tih ćelija više puta. Kod testiranja

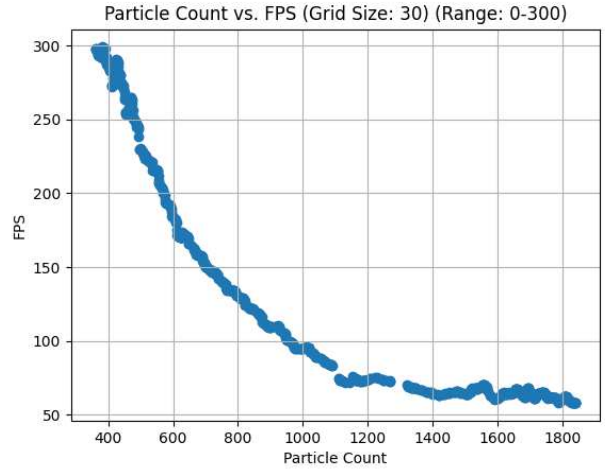
korišteno je 20 iteracija da bi izračunali nove pozicije i brzine čestica, što uzima sve više resursi s povećavanjem broja čestica.

Uspoređene su performance simulacije kada se računa sva fizika, kada se računaju samo sudari, kada se računa samo fiziku fluida i kada se fizika uopće ne računa kao baznu liniju. Kod testiranja korištena je veličinu mreže 30x30 i veličinu čestica takvu da je promjer čestice tri puta manji od dužine ćelije.

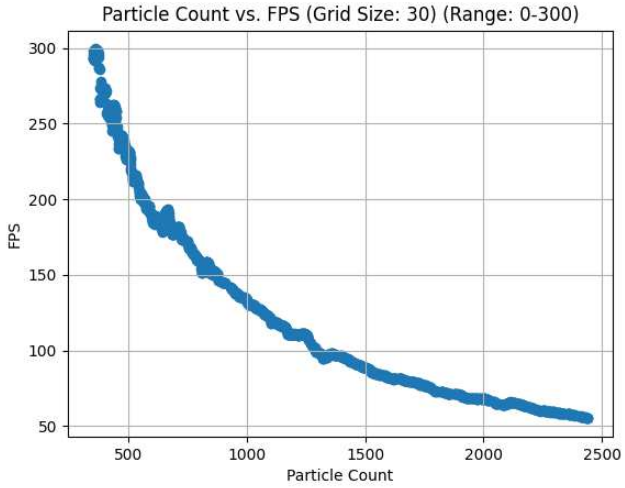
Svi testovi su bili rađeni na prijenosnom računalu koje ima 16 GB radne memorije, Intel Core i7 1165G7 procesor i integriranu Intel iRISx grafičku karticu.



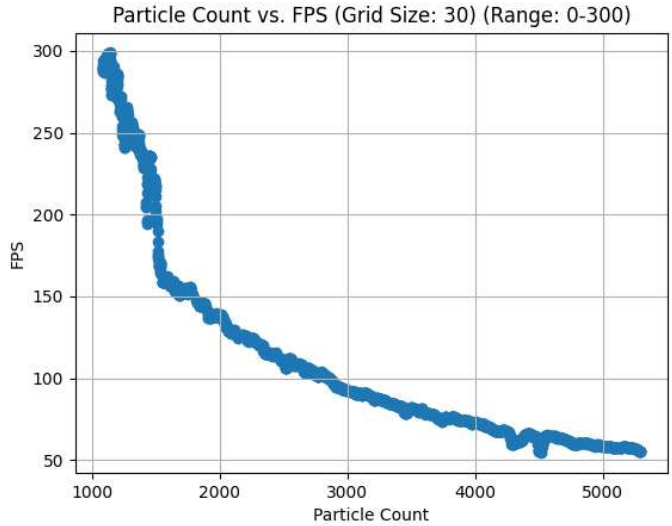
Slika 5.1 Performance simualcije sa svim izračunima fizike



Slika 5.1 Performance simulacije s isključenom fizikom fluida



Slika 5.1 Performance simulacije s isključenim rješavanjem sudara



Slika 5.1 Performance simulacije bez izračuna fizike

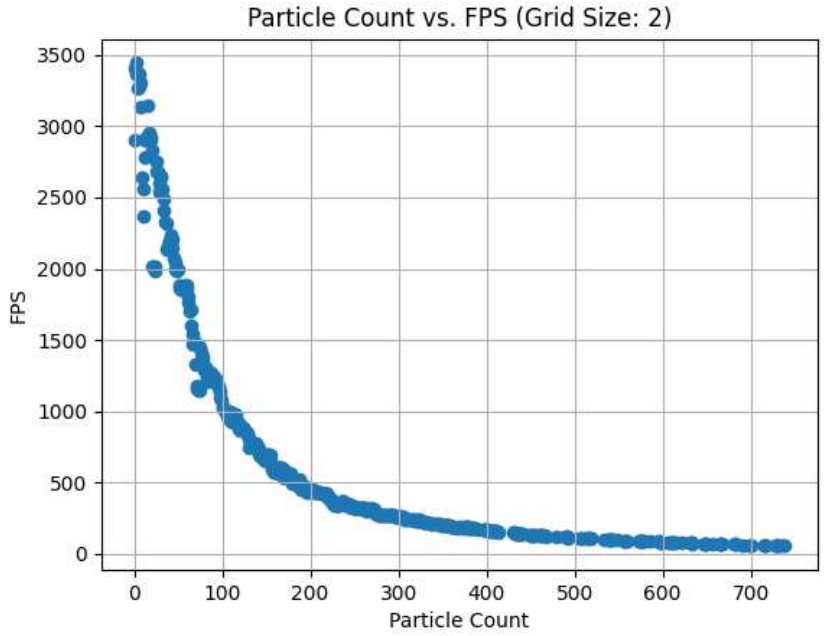
Kao što se može vidjeti sa slike 5.10 broj čestica koje ovaj grafički pogon može prikazati na testnom računalu prije nego padne ispod 60 slika po sekundi je oko 4500, a dodavanje bilo kakvog izračuna će uvijek smanjiti taj broj. Na slici 5.9 može se vidjeti kako se performance mijenjaju kada se doda izračun fluida u simulaciju, tada broj slika po sekundi padne ispod 60 kada se dođe do oko 2200 čestica, znači da je već izgubljeno pola performansi koje su početno bile prisutne.

Kada simulacija računa samo fiziku za rješavanje sudara, što je prikazano na slici 5.8, može se vidjeti da već na 1500 čestica simulacija pada ispod 60 slika po sekundi. U usporedbi s

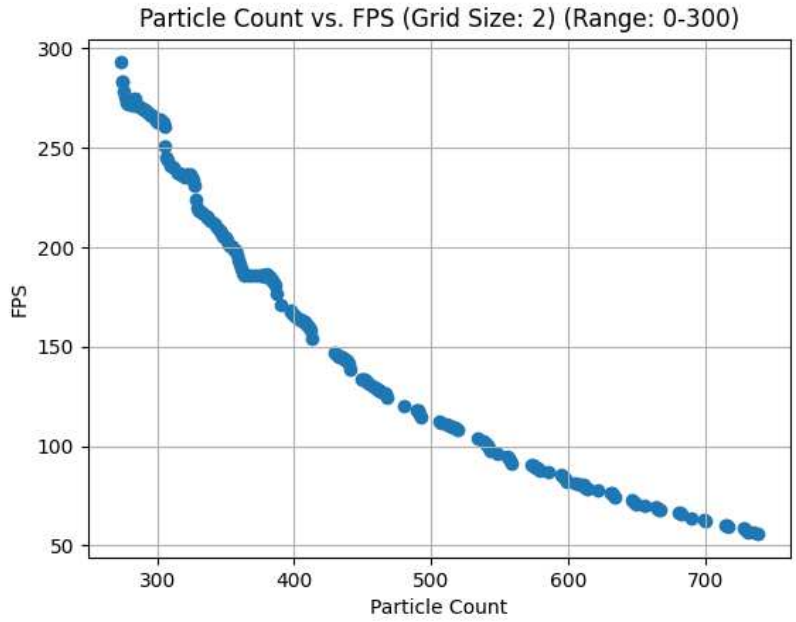
izračunom samo fizike fluida može se zaključiti da je rješavanje sudara sporije od izračuna fizike fluida za od 30% do 50%, ali su još uvijek na sličnoj razini što se tiče složenosti.

Na slici 5.7 može se vidjeti performance simulacije gdje se ona vrti s uključenim svim izračunima fizike. Ovdje se može vidjeti da je maksimalni broj čestica koje se mogu simulirati ovom simulacijom na više od 60 slika po sekundi između 1000 i 1200. Taj broj nije dovoljno velik za bilo kakvu ozbiljnu simulaciju, ali je dovoljno dobar za ovaj rad gdje je cilj bio samostalno istraživanje i implementacija simulacije fluida.

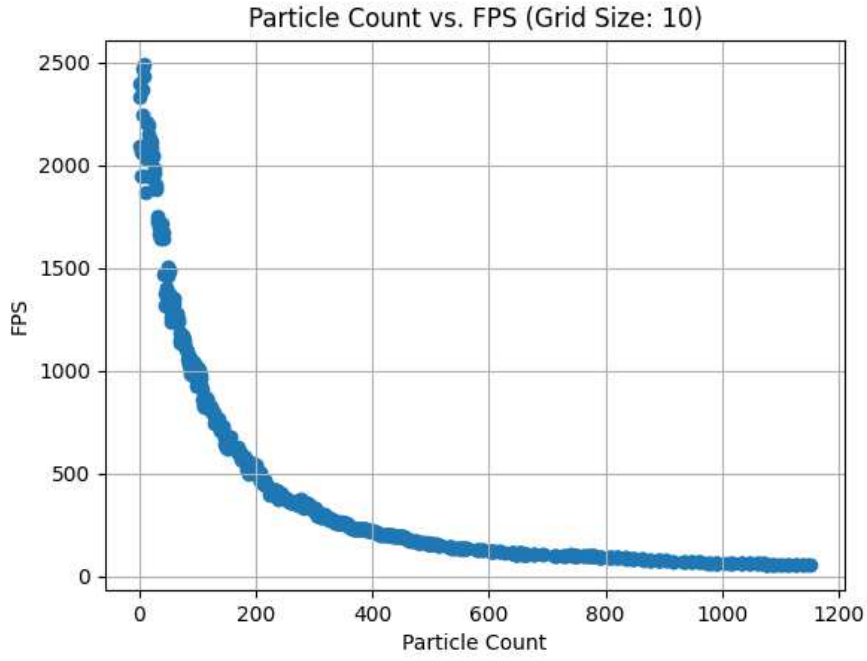
Drugi faktor osim broja čestica koji utječe na performance simulacije je broj ćelija unutar mreže. S jedne strane što je više ćelija to će biti potrebno raditi manje provjera sudara unutar jedne ćelije i susjednih ćelija jer će u njima biti manje čestica, ali s druge strane postojat će više ćelija kroz koje je potrebno iterirati. Zbog toga je potrebno pronaći broj ćelija koji daje najbolje performance simulacije i sa strane detekcije sudara i računanja fizike fluida. U nastavku će biti analizirane razne postavke veličine mreže i njihov utjecaj na performance simulacije.



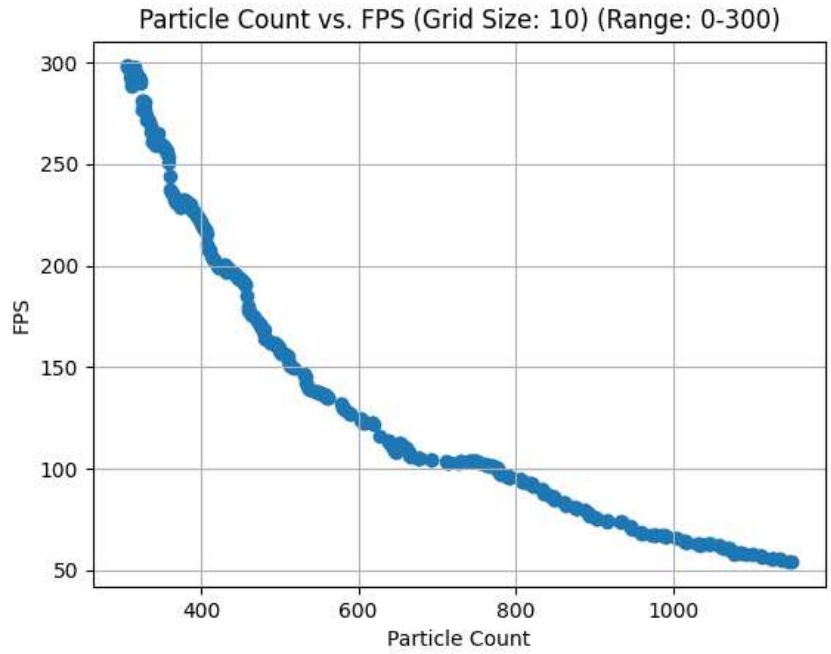
Slika 5.1 Performance 2x2 mreže



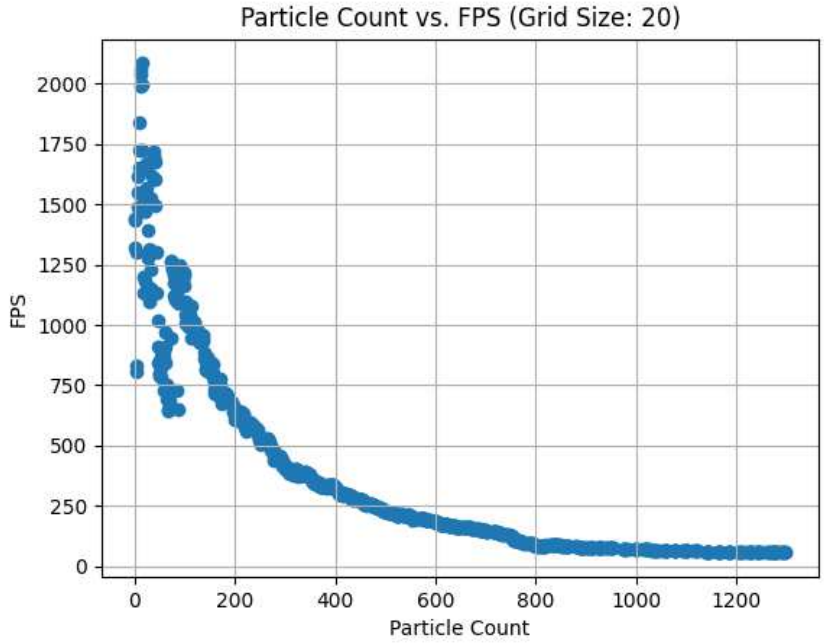
Slika 5.1 Performance 2x2 mreže približeno na 300 FPS-a



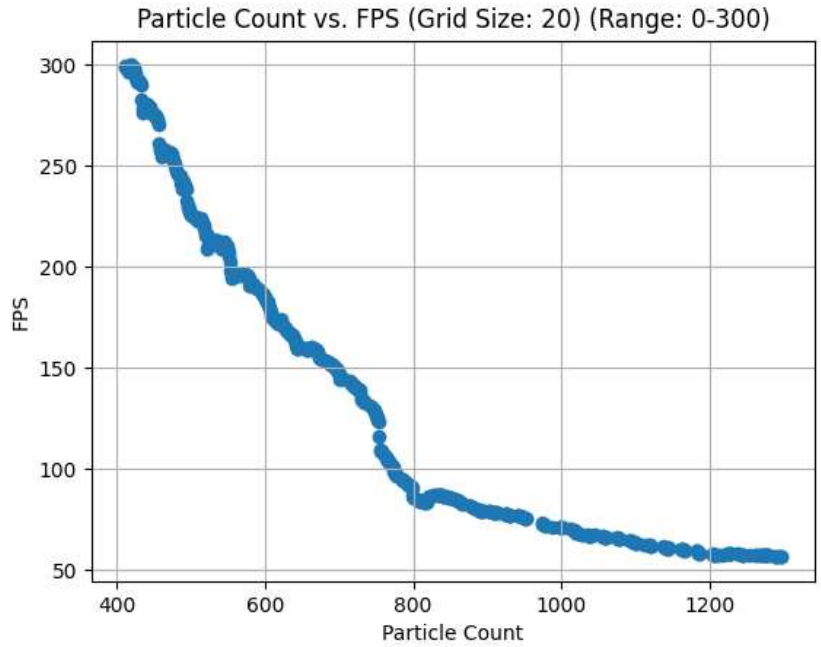
Slika 5.1 Performance 10x10 mreže



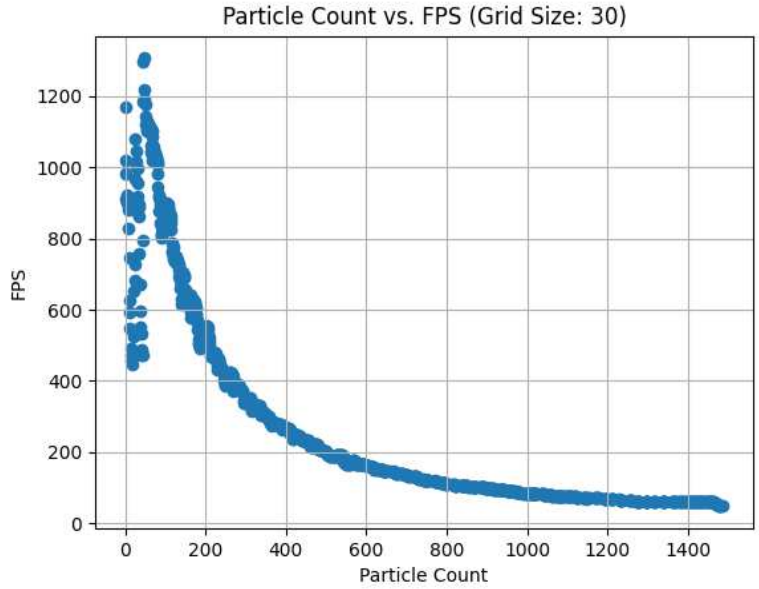
Slika 5.1 Performance 10x10 mreže prilično na 300 FPS-a



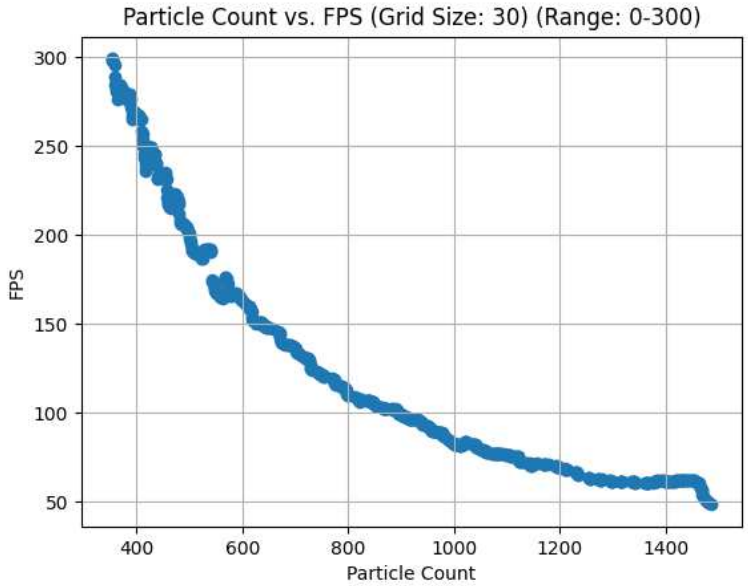
Slika 5.1 Performance 20x20 mreže



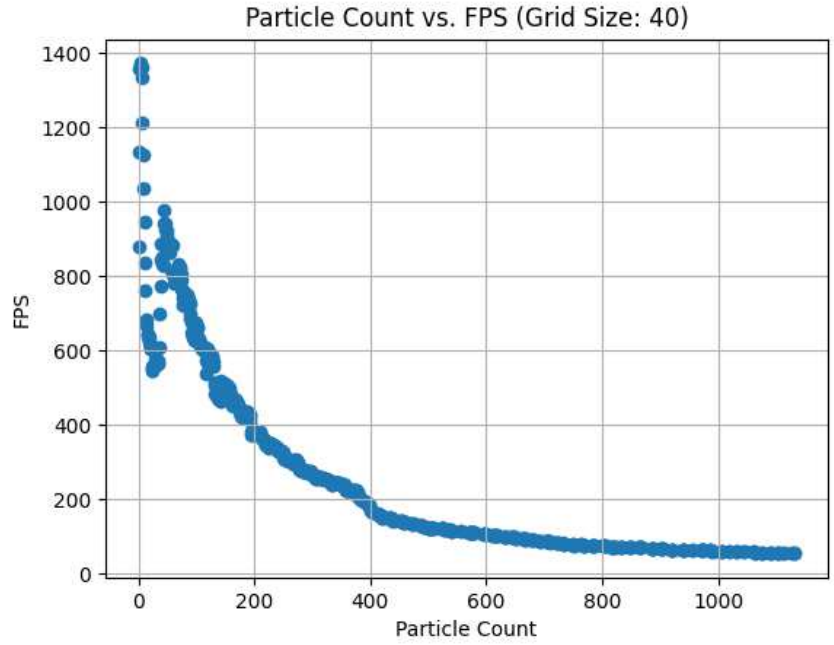
Slika 5.1 Performance 20x20 mreže približeno na 300 FPS-a



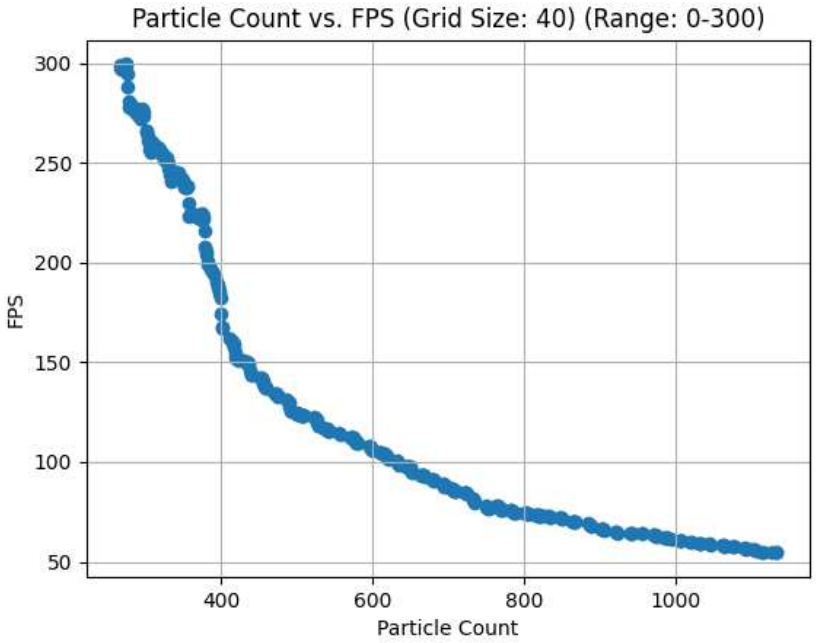
Slika 5.1 Performance 30x30 mreže



Slika 5.1 Performance 30x30 mreže približeno na 300 FPS-a:



Slika 5.1 Performance 40x40 mreže



Slika 5.1 Performance 40x40 mreže približeno na 300 FPS-a

Na slici 5.11 se može vidjeti kako broj čestica ovisi o broju slika po sekundi kada se koristi mreža 2x2, u tome slučaju ne dobivamo nikakve prednosti mreže. Može se primijetiti da performance simulacije kada je u simulacjii manje od 50 čestica dostižu i 3500 slika po sekundi i onda otprilike eksponencijalno pada s brojem čestica. Nije dostupno dovoljno podataka da bi se moglo točno interpolirati krivulju, ali na početku krivulja izgleda eksponencijalno, a kasnije dok se razmatra veći broj čestica krivulja izgleda kao da linearno pada. To se može vidjeti na slici 5.12 koja prikazuje isti graf kao slika 5.11 na području od 300 slika po sekundi na manje. Sa slike 5.12 može se vidjeti da simulacija pada ispod 60 slika po sekundi već nakon 650 čestica.

Dok se veličinu mreže poveća na 10 puta 10, što će nam dati ukupno 100 ćelija može se vidjeti promjena performanci. Na slici 5.13 može se vidjeti da je početni broj slika po sekundi u usporedbi s mrežom 2 puta 2 pao s 3500 na 2500. To je očekivano zato što sada i na malom broju čestica odmah postoji već 100 ćelija kroz koje je potrebno iterirati u svakom koraku simulacije i to usporava simulaciju i na početku. Za veći broj čestica, što se može vidjeti na slici 5.14, vidljivo je da je broj slika u sekundi znatno veći u usporedbi s dva puta dva

mrežom i da je u ovoj konfiguraciji mreže moguće doći i do 1000 čestica prije nego simulacija padne ispod 60 slici.

Kada se veličinu mreže poveća na 20 puta 20, slika 5.15 i 5.16, broj slika u sekundi za mali broj čestica je oko 2000, a simulacija pada ispod 60 slika u sekundi za 1200 čestica.

Najbolje performance za velik broj čestica postignuto je za veličinu mreže 30 puta 30, što se može vidjeti na slikama 5.17 i 5.18, gdje simulacija može izdržati i 1400 čestica prije nego padne ispod 60 slika u sekundi.

Slike 5.19 i 5.20 prikazuju performance simulacije kada se koristi veličina mreže veća od 30 puta 30, u ovom slučaju to je 40 puta 40. Tamo se može vidjeti da su se performance za velik broj čestica pogoršale i da opet ne možemo ići na više od 1000 čestica da bi održali 60 slika u sekundi kao što je bilo i kod mreže 20 puta 20.

5.4. Moguća proširenja

Za simulaciju i grafički pogon postoje mnoga poboljšanja i proširenja koje bi bilo moguće dodati.

5.4.1. Grafički pogon

Neka od proširenja grafičkog pogona bi bila neka vrsta kamere preko koje bi se čuvao omjer zaslona, neki predefinirani geometrijski objekti kao što su krugovi, kvadrati, zvijezde. Još jedna korisna stvar bi bila jednostavno crtanje ravnih i zakrivljenih linija i mogućnost za dodavanje teksta unutar prozora.

No glavno proširenje grafičkog pogona bi bila mogućnost prikazivanja scene u tri dimenzije što bi onda omogućilo i trodimenzionalne simulacije.

5.4.2. Simulacija fluida

Najvažnije poboljšanje bi bilo optimizacija simulacije da je moguće simulirati veći broj čestica i uklanjanje ovisnosti simulacije o grafičkom pogonu, time i broju slika u sekundi.

Optimizacija simulacije bi se mogla postići boljim korištenjem struktura podataka s kojima se manipulira česticama i mrežom fluida i smanjivanjem kompleksnosti i složenosti logike

unutar izračuna određenih stvari unutar simulacije. Još jedan razlog zašto je simulacija relativno spora je zato što se odvija u samo jednoj dretvi na procesoru. Svi izračuni se obavljaju sekvencijski jedan za drugim iako neki izračuni nisu međusobno ovisni i mogili bi se računati paralelno. Tako da bi još jedan način optimizacije bio podijeliti posao koji simulator radi na više dretvi. Tu bi naravno bilo potrebno pronaći dio posla koji se isplati i može paralelizirati. Naprimjer računanje sudara između ćelija bi se mogao podijeliti u na više nezavisnih poslova što ne bi utjecalo na točnost simulacije zato što taj dio ionako nije potpuno precizan i potrebno ga je napraviti više puta. Dio izračuna za mrežu fluida se isto može paralelizirati i time ubrzati i taj dio simulacije.

Drugo poboljšanje koje bi se moglo uvest je uklanjanje ovisnosti simulacije od grafičkog pogona koje se isto može postići koristeći višedretvenosti, jedna ili više dretvi bi bile zadužene za grafički pogon i za samo prikazivanje objekata u sceni, a jedan ili više simulacijskih dretvi bi bile zadužene za simulaciju i njene izračune koje bi onda ažurirale pozicije objekata potpuno neovisno o grafičkom pogonu. Grafički pogon bi isto tako samo koristio ažurirane pozicije objekata i iscrtavao ih na zaslonu potpuno neovisno o simulaciji. U slučaju kada bi simulacija bila brže od pogona ne bi dolazilo do ikakvih problema, jer bi se uvijek znalo da grafički pogon iscrtava ažurirane objekte, a u slučaju kada bi simulacija bila sporija onda bi grafički pogon više slika iscrtavao iste slike objekata za koje je zadužena simulacija, no i to je poboljšanje na trenutne konfiguracije zato što grafički pogon može iscrtavati i neke druge stvari koje nisu u simulaciji i više ne treba čekati simulaciju da bude gotova s jednim korakom.

Još jedna stvar koja bi se mogla poboljšati kod simulacije je pružiti korisniku neku veću razinu interakcije. Naprimjer moglo bi se dodati da korisnik može ručno dodavati i simulirati neke druge prepreke kao naprimjer cijevi ili zidove. Isto bi bilo dobro dodati opciju da korisnik može saznati neka fizička svojstva o fluidu koja simuliramo, kao naprimjer tlak, temperaturu, brzinu ili gustoću. Time bi simulacija dobila veću korisnost i primjenjivost nego što je ima sada.

6. Zaključak

Fluidi se u teoriji opisuju pomoću Navier Stokes jednadžbi, no u praksi nije moguće na lagan način ili uopće riješiti te jednadžbe i zato ih nije moguće direktno koristiti kod računalnih simulacija. Zbog toga se koriste razni algoritmi koji pokušavaju dovoljno dobro aproksimirati rješenja tih jednadžbi, a oni se dijele u Eulerove i Lagrangeove metode. PIC metoda kombinira oba pristupa jer za simulaciju fluida koristi i mrežu i zasebne čestice i time dobiva najbolje iz oba pristupa simulaciji fluida. FLIP algoritam je proširenje PIC metode koje daje veći naglasak na pojedina gibanja čestica unutar simulacije i u praksi daje prilično dobre rezultate.

7. Literatura

[1] Alessandro Bazzi, The Navier-Stokes Equations, Cantor's Paradise (2020, rujan). Poveznica: https://www.cantorsparadise.com/the-navier-stokes-equations-461f7453d79e ; pristupljeno 15.04.2023.

[2] Gustav Tschirschnitz, Pierre Sabrowski, COMPUTATIONAL FLUID DYNAMICS METHODS EXPLAINED (2020, studeni). Poveznica: https://www.dive-solutions.de/blog/cfd-methods ; pristupljeno 15.04.2023.

[3] Matthias Müller, 17 - How to write an Eulerian fluid simulator with 200 lines of code. (2022, prosinac). Poveznica: https://youtu.be/iKAVRgIrUOU ; pristupljeno: 24.04.2023.

[4] Matthias Müller, 18 - How to write a FLIP water / fluid simulation running in your browser. (2023, siječanj). Poveznica: https://youtu.be/XmzBREkK8kY ; pristupljeno 30.04.2023.

[5] Alexander Overvoorde, Vulkan Tutorial. (2023, travanj). Poveznica: https://vulkan-tutorial.com/resources/vulkan_tutorial_en.pdf ; pristupljeno: 05.04.2023.

[6] Gemma Ryles, What is Vulkan? All the facts on the Direct X alternative. (2022, lipanj). Poveznica: https://www.trustedreviews.com/explainer/what-is-vulkan-2946841 ; pristupljeno: 07.05.2023.

[7] Unknown, GLFW. (2022, srpanj). Poveznica: https://www.glfw.org/ ; pristupljeno: 14.04.2023.

[8] Unknown, OpenGL Mathematics. (2020, svibanj). Poveznica: https://glm.g-truc.net/0.9.9/index.html ; 14.04.2023.

[9] Unknown, Dear ImGui, (2014, kolovoz). Poveznica: https://github.com/ocornut/imgui ; pristupljeno: 30.04.2023.

[10] Brendan Galea, Vulkan (c++) Game Engine Tutorials, (2020, studeni). Poveznica:

https://youtube.com/playlist?list=PL8327DO66nu9qYVKLDmdLW_84-yE4auCR ; pristupljeno: 07.03.2023.

[11] Glen Elert, Equations of Motion, (2009, listopad). Poveznica: https://physics.info/motion-equations/ ; pristupljeno: 10.05.2023.

Sažetak

U ovom radu je objašnjen i predstavljen dvodimenzionalni grafički pogon napravljen s grafičkim sučeljem Vulkan koji je onda korišten za potrebe prikaza simulacije fluida koristeći kombinaciju Eulerovog i Lagrangeovog pristupa implementacijom FLIP algoritma.

Detaljno je objašnjena implementaciju grafičkog pogona, simulacija fluida i njezina interaktivnost. Rezultati simulacije i njezinih performansi su detaljno analizirani pod utjecajem raznih parametara i opisana su moguća proširenja grafičkoga pogona i simulacije.