SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2298

FIZIKALNO TEMELJENA SIMULACIJA ČVRSTIH TIJELA

Leon Katić

Zagreb, lipanj 2026.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2298

FIZIKALNO TEMELJENA SIMULACIJA ČVRSTIH TIJELA

Leon Katić

Zagreb, lipanj 2026.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 2. ožujka 2026.

ZAVRŠNI ZADATAK br. 2298

Pristupnik:	**Leon Katić (0036559392)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Fizikalno temeljena simulacija čvrstih tijela**

Opis zadatka:

Jedna od osnovnih simulacija u grafičkim pogonima je fizikalno temeljena simulacija čvrstih tijela (engl. rigid body simulation). Cilj ovog rada je istražiti fizikalne osnove ponašanja čvrstih tijela. Proučiti načine izračuna kolizije u 3D prostoru. Napraviti programsku implementaciju, odnosno biblioteku programskih rutina za izračun fizikalno temeljene simulacije čvrstih tijela. Posebno obratiti pažnju na mogućnost interaktivnog djelovanja korisnika na objekte u simuliranom okruženju. Provesti testiranje na nizu primjera te analizirati i ocijeniti ostvarene rezultate. Razmotriti upotrebljivost dobivenih rezultata i moguće smjernice proširenja. Izraditi odgovarajući programski proizvod koristeći programski jezik C++ i grafičko programsko sučelje OpenGL. 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: 12. lipnja 2026.

Sažetak

U ovom radu opisana je implementacija fizikalne simulacije sudara čvrstih tijela u trodimenzionalnom prostoru. Simulacija je izrađena u programskom jeziku C++ uz korištenje biblioteka OpenGL, GLFW i GLM za renderiranje i matematičke operacije. Teorijska osnova rada obuhvaća vektorski račun, transformacijske matrice kamere te fizikalni model gibanja i sudara. Implementirana su dva tipa detekcije i razrješavanja sudara — sfera-sfera i sfera-statični objekt — pri čemu se za pronalaženje najbliže točke na trokutu koristi algoritam temeljen na Voronoijevim regijama prema Ericsonovom djelu *Real-Time Collision Detection*. Razrješavanje sudara temelji se na očuvanju količine gibanja uz parametar elastičnosti sudara. Korisniku je omogućeno interaktivno upravljanje kamerom te ispucavanje sfera u scenu sastavljenu od statičnih objekata.

Ključne riječi: fizikalna simulacija, detekcija sudara, OpenGL, C++

Summary

This paper describes the implementation of a rigid body collision simulation in three-dimensional space. The simulation is built in C++ using the OpenGL, GLFW and GLM libraries for rendering and mathematical operations. The theoretical foundation covers vector algebra, camera transformation matrices and the physical model of motion and collision. Two types of collision detection and resolution are implemented — sphere-sphere and sphere-static object — where the closest point on a triangle is determined using an algorithm based on Voronoi regions from Ericson's *Real-Time Collision Detection*. Collision resolution is based on conservation of momentum with an elasticity parameter. The user is provided with interactive camera control and the ability to launch spheres into a scene composed of static objects.

Keywords: physical simulation, collision detection, OpenGL, C++

Sadržaj

Uvod

Fizikalne simulacije imaju dugu povijest primjene u inženjerskoj praksi. Koriste se u širokom spektru područja — od simulacije leta svemirskih letjelica i projektiranja građevinskih konstrukcija do razvoja video igara i računalno generiranih vizualnih efekata u filmskoj industriji. Osnovni cilj svake fizikalne simulacije je što vjernije dočarati ponašanje fizikalnih sustava u stvarnom svijetu, no potpuna preciznost ostaje računalno izrazito zahtjevna, a u praksi nedostižan cilj. Iz tog razloga fizikalni pogoni, računalni programi koji aproksimativno izračunavaju fizikalne pojave unutar zadanog sustava, predstavljaju temelj suvremenih simulacija.

Jedan od temeljnih problema u fizikalnim simulacijama je detekcija i razrješavanje kolizija između krutih tijela. Ovaj problem posebno je izazovan kada se radi o proizvoljnim trodimenzionalnim oblicima jer je potrebno u svakom trenutku odrediti ne samo je li došlo do sudara, već i u kojoj točki, s kojom silom te kako tijela treba odmaknuti kako bi simulacija ostala stabilna. Dodatnu složenost unosi potreba za izvođenjem u stvarnom vremenu, što znači da sve fizikalne proračune treba završiti unutar nekoliko milisekundi po sličici.

Cilj ovog rada je implementacija jednostavnog fizikalnog pogona koji simulira koliziju čvrstih sfera međusobno te njihovu interakciju sa statičnim trodimenzionalnim objektima, uključujući i konkavne i konveksne oblike. Pogon je implementiran u programskom jeziku C++, a za prikaz trodimenzionalne scene koristi se OpenGL, pri čemu se funkcije učitavaju pomoću GLAD biblioteke, dok se upravljanje prozorom i ulazom ostvaruje kroz GLFW okvir. Simulacija je izrađena te se izvodi na operativnom sustavu Linux. Za matematičke operacije nad vektorima i matricama koristi se GLM biblioteka. Simulacija se izvodi u stvarnom vremenu s podrškom za interaktivno upravljanje kamerom i dinamičko dodavanje objekata u scenu.

1. Teorijska osnova

Kako bi bilo moguće realizirati fizikalnu simulaciju u stvarnom vremenu, potrebno je razumjeti nekoliko temeljnih koncepata na kojima ona počiva. Simulacija opisana u ovom radu je interaktivna, što znači da se svi fizikalni proračuni moraju izvršavati dovoljno brzo da korisnik ne primjećuje kašnjenje — idealno 60 ili više sličica u sekundi. Da bi se to postiglo, koriste se aproksimativne metode koje žrtvuju određenu razinu fizikalne točnosti u korist brzine izvođenja.

Fizikalni sustav u ovom radu sastoji se od dva tipa objekata: dinamičnih sfera koje su podložne gravitaciji i međusobnim sudarima, te statičnih trodimenzionalnih mreža koje služe kao nepomične prepreke u sceni. Za svaki vremenski korak simulacije potrebno je ažurirati pozicije i brzine dinamičnih objekata, detektirati sve kolizije te ih razriješiti.

1.1. Vektori i vektorski račun

Jedan od glavnih temelja ove simulacije su vektori i vektorski račun. Vektor je matematički objekt koji ima smjer i iznos, a u trodimenzionalnom prostoru predstavlja se s tri komponente, kao što je prikazano izrazom (1). U kontekstu fizikalne simulacije vektori se koriste za reprezentaciju pozicija objekata, njihovih brzina, ubrzanja te normalnih vektora površina potrebnih za razrješavanje kolizija.

Dvije najvažnije vektorske operacije u ovoj simulaciji su skalarni produkt i vektorski produkt. Skalarni produkt dvaju vektora računa se prema izrazu (2) i daje skalar koji opisuje koliko su vektori međusobno poravnati — koristi se primjerice pri izračunu komponente brzine u smjeru normale površine prilikom sudara. Vektorski produkt, opisan izrazom (3), daje novi vektor okomit na oba ulazna vektora, pri čemu njegova duljina opisuje koliko su ta dva vektora međusobno okomita. Koristi se, primjerice, pri izračunu vektora desne strane kamere iz smjera pogleda i vektora gore.

Preostale dvije operacije su duljina vektora i normalizacija. Duljina vektora računa se prema izrazu (4), kao kvadratni korijen zbroja kvadrata svih komponenti. Normalizacija vektora, opisana izrazom (5), daje vektor duljine 1 kolinearan s originalnim vektorom, dijeljenjem vektora s njegovom duljinom.

$$v_1 = (v_{1x}, v_{1y}, v_{1z}), \quad v_2 = (v_{2x}, v_{2y}, v_{2z}) \tag{1}$$

$$v_1 \cdot v_2 = v_{1x} v_{2x} + v_{1y} v_{2y} + v_{1z} v_{2z} \tag{2}$$

$$v_1 \times v_2 = (v_{1y} v_{2z} - v_{1z} v_{2y}, \quad v_{1z} v_{2x} - v_{1x} v_{2z}, \quad v_{1x} v_{2y} - v_{1y} v_{2x}) \tag{3}$$

$$|v_1| = \sqrt{v_{1x}^2 + v_{1y}^2 + v_{1z}^2} \tag{4}$$

$$\hat{v}_1 = \frac{v_1}{|v_1|} \tag{5}$$

1.2. Kamera i transformacijske matrice

Za vizualni dio fizikalne simulacije potrebne su dvije ključne matrice: matrica transformacije iz globalnog koordinatnog sustava u lokalni sustav kamere te matrica perspektivne projekcije za projiciranje 3D točaka u 2D ravninu prikaza.

Kamera je definirana očištem *c* i gledištem *g*, koji su predstavljeni trodimenzionalnim vektorima, te vektorom u_w koji zadaje smjer „gore" u sceni. Na temelju tih podataka izračunavaju se tri ortonormirana vektora prema izrazu (6): vektor *f* koji opisuje smjer gledanja kamere, desni vektor *r* te vektor „gore" u lokalnom sustavu kamere *u*.

Matrica translacije M_T, dana izrazom (7), pomiče koordinatni sustav u odnosu na očište kamere c. Matrica rotacije M_R, dana izrazom (9), rotira koordinatni sustav scene u lokalni koordinatni sustav kamere koristeći izračunate vektore $\hat{r}$, $\hat{u}$ i $\hat{f}$ kao retke matrice. Na kraju, točke u lokalnom sustavu kamere projiciraju se u 2D ravninu prikaza pomoću matrice perspektivne projekcije M_P, opisane izrazom (10).

Konačna matrica kojom se globalne koordinate točke transformiraju u projicirane koordinate dobiva se prema izrazu (11), kao produkt inverza matrice translacije, inverza matrice rotacije te matrice perspektivne projekcije.

$$\mathrm{f}=\mathrm{g}-\mathrm{c},\ \ \mathrm{r}=\mathrm{f}\times u_w,\ \ \mathrm{u}=\mathrm{r}\times f \tag{6}$$

$$M_T=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ c_x & c_y & c_z & 1\end{pmatrix} \tag{7}$$

$$M_T^{-1}=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -c_x & -c_y & -c_z & 1\end{pmatrix} \tag{8}$$

$$M_R=\begin{pmatrix}\hat{r}_x & \hat{r}_y & \hat{r}_z & 0\\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0\\ \hat{f}_x & \hat{f}_y & \hat{f}_z & 0\\ 0 & 0 & 0 & 1\end{pmatrix} \tag{9}$$

$$M_P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{|f|} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

$$T_{\text{projekcija}} = \text{T}_{\text{global}} \cdot M_T^{-1} \cdot M_R^{-1} \cdot M_P \tag{11}$$

1.3. Fizikalni model

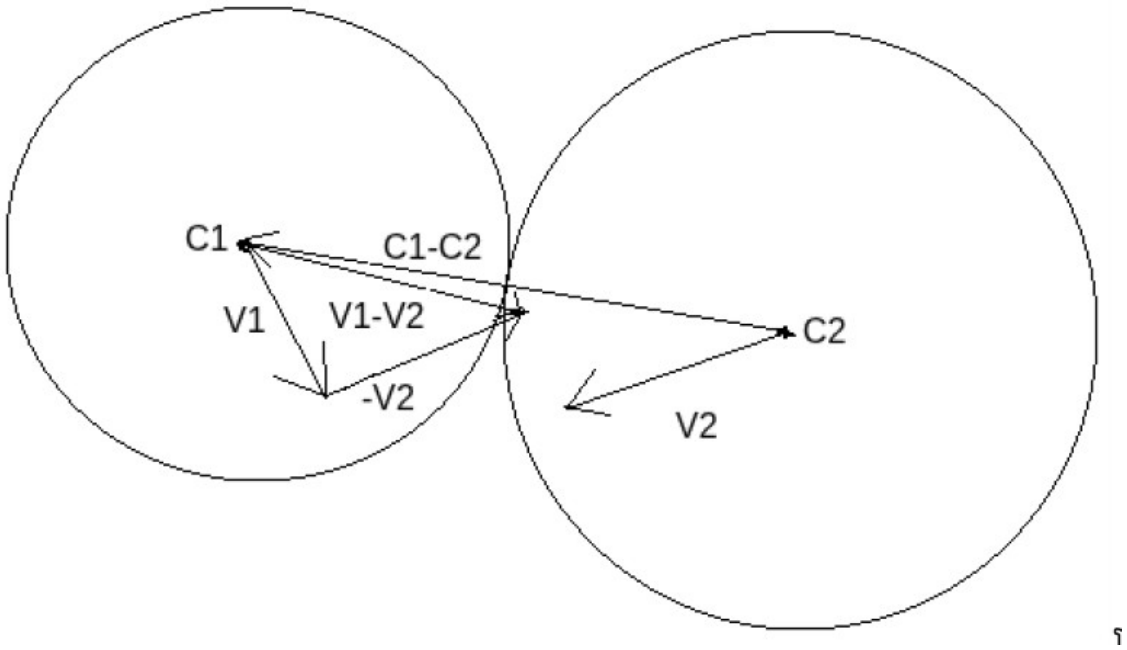
Da bi se ostvarilo kretanje sfera i njihovo međusobno odbijanje, potrebna je odgovarajuća fizikalna podloga. Svaka sfera opisana je vektorom centra, vektorom brzine, radijusom i masom. Cijela simulacija odvija se pod utjecajem gravitacije, a kretanje svake sfere određeno je isključivo njezinim vektorom brzine prema izrazu (12).

$$v_{\text{sfera}} = \text{v}_{\text{poč}} - (0, \text{gt}, 0) \tag{12}$$

Kolizija dviju sfera detektira se usporedbom udaljenosti njihovih centara c_1 i c_2 sa zbrojem njihovih radijusa r_1 i r_2, prema nejednakosti (13). Intuitivno, ako je udaljenost centara veća od zbroja radijusa, sfere se ne dodiruju; ako su jednaki, sfere se izvana dotiču; a ako je udaljenost strogo manja, sfere se međusobno preklapaju.

$$r_1 + \text{r}_2 \geq |c_2 - c_1| \tag{13}$$

Za razrješavanje kolizije potrebno je odrediti nove vektore brzine na temelju masa, centara i brzina sfera u trenutku sudara, uz pretpostavku da je zbroj radijusa jednak udaljenosti centara. Kao mjera usklađenosti vektora brzina s linijom koja spaja centre sfera uvodi se relativna brzina, izračunata prema izrazu (14) kao skalarni produkt razlike brzina i normaliziranog smjera sudara. Intuitivno, relativna brzina je najveća kada su vektori brzine suprotnih orijentacija i poravnati s osi sudara. Uvjet za odbijanje je da relativna brzina bude negativna — ako nije, sfere se u trenutku dodira već udaljavaju jedna od druge i odbijanje nije potrebno.



Sl. 1.3.1 Trenutak kolizije

$$c_r = c_1 - c_2, \quad v_{\text{rel}} = (v_1 - v_2) \cdot \hat{c}_r \tag{14}$$

Nadalje se uvodi koeficijent restitucije k, koji opisuje koliko se sfere snažno odbijaju pri sudaru, pri čemu vrijedi $0 < k < 1$. Vrijednost $k = 1$ odgovara savršeno elastičnom sudaru bez gubitka energije, dok $k = 0$ odgovara potpuno neelastičnom sudaru. Sfere imaju mase m_1 i m_2.

Na temelju relativne brzine, koeficijenta restitucije i masa sfera izračunava se impuls sudara J prema izrazu (15). U nazivniku se nalazi recipročna reducirana masa sustava dviju sfera. Impuls se zatim primjenjuje na vektore brzine obiju sfera prema izrazu (16) pri čemu se prvoj sferi brzina povećava u smjeru vektora c_r a drugoj smanjuje, razmjerno njihovim masama. Na taj način zadovoljeno je načelo očuvanja količine gibanja.

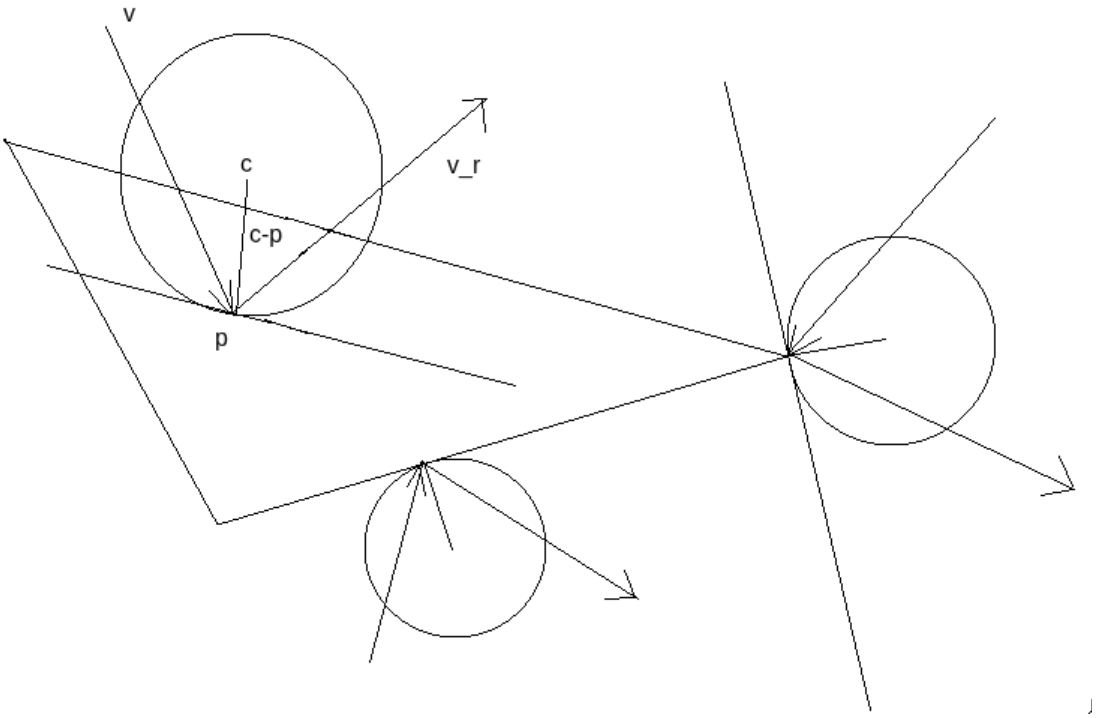
$$J=\frac{-(1+k)\cdot v_{rel}}{\frac{1}{m_1}+\frac{1}{m_2}} \tag{15}$$

J = -(1 + k) cdot {v_{rel}} over {1 over m_{1} + 1 over m_{2}}

$$v_1'=v_1+\left(\frac{J}{m_1}\right)\cdot\hat{c}_r,\ \ v_2'=v_2-\left(\frac{J}{m_2}\right)\cdot\hat{c}_r \tag{16}$$

Da bi se dobilo odbijanje od statičnih objekata, potrebno je pronaći najbližu točku na trokutu statične mreže od centra sfere. Na slici (Sl. 1.3.2) prikazana su tri slučaja: odbijanje od površine unutar trokuta, odbijanje od točke na bridu trokuta te odbijanje od samog vrha trokuta.

Za pronalaženje te točke koristi se algoritam iz poglavlja 5.1.5 izvora [1], koji za trokut određen trima vrhovima i proizvoljnu točku u prostoru vraća najbližu točku na trokutu. Algoritam se temelji na Voronoijevim regijama trokuta — prvo se provjerava pripadaju li projekcija točke regiji nekog vrha, u tom slučaju se vraća taj vrh. Zatim se provjerava pripada li regiji nekog brida, u tom slučaju se vraća ortogonalna projekcija točke na taj brid. Ako nijedan od tih uvjeta nije zadovoljen, točka se projicira na površinu trokuta i određuje se baricentričnim koordinatama.



Sl. 1.3.2 Trenutak kolizije s trokutom

Kada je određena najbliža točka na trokutu, provjerava se je li uopće došlo do sudara. Sfera se sudarila s trokutom ako vrijedi uvjet (17), gdje je *c* centar sfere, *p* najbliža točka na trokutu od centra sfere, a *r* radijus sfere. Uvjet je intuitivan — analogno detekciji sudara dviju sfera, uspoređuje se udaljenost između relevantnih točaka s radijusom sfere.

$$|c - p| \leq r \tag{17}$$

Nakon potvrde sudara određuje se reflektirani vektor brzine. Vektor od točke *p* prema centru *c* normalizira se prema izrazu (18) i označava s *n*. Iz istog izraza definirana je i relativna brzina v_{rel} kao skalarni produkt vektora brzine *v* i normale *n*, koja mjeri koliko se vektor brzine poklapa sa smjerom normale površine.

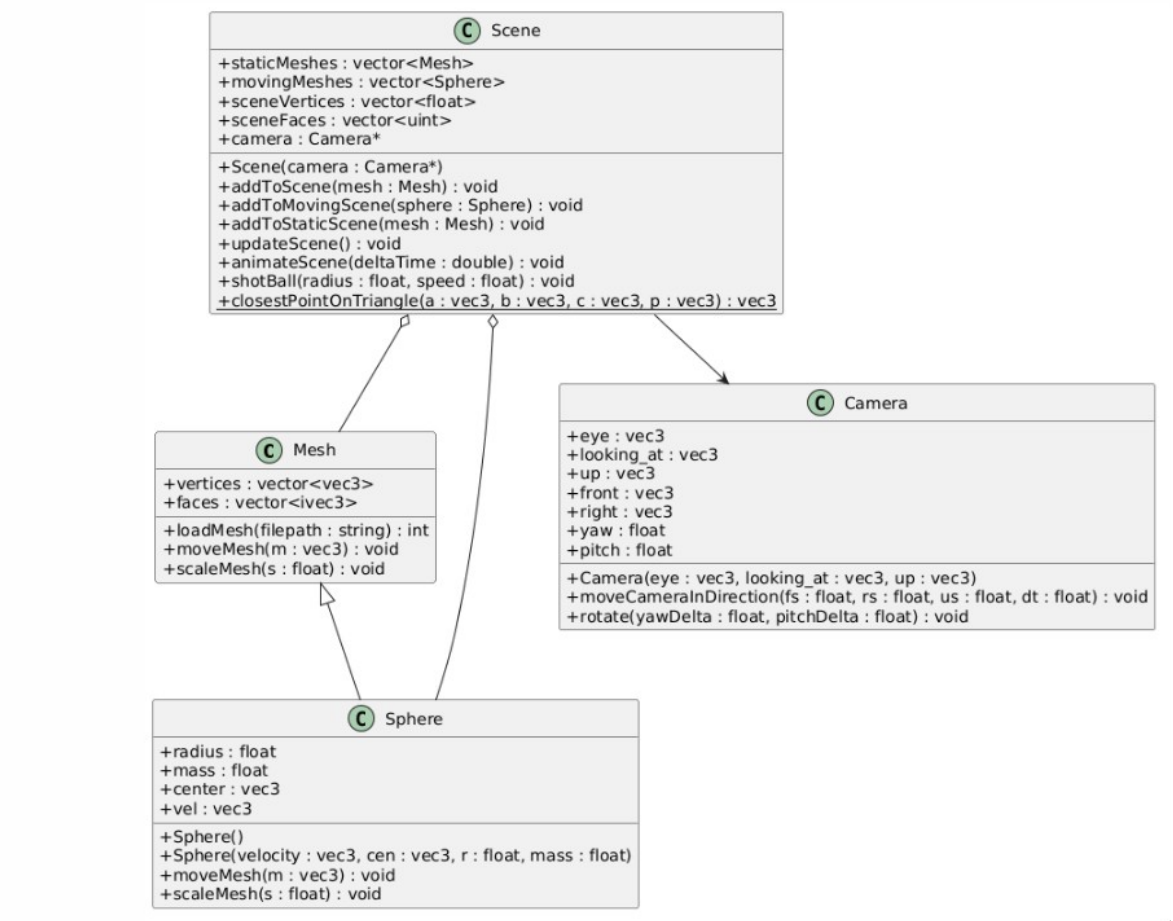
$$\mathrm{n} = \frac{c - p}{|c - p|}, \quad v_{\mathrm{rel}} = \mathrm{v} \cdot n \tag{18}$$

Ako je v_{rel} manji od nule, sfera se giba prema površini pa je odbijanje potrebno. U suprotnom, sfera se već udaljava od površine i nema potrebe mijenjati vektor brzine. Reflektirani vektor brzine v_r izračunava se prema izrazu (19), gdje k označava koeficijent restitucije jednak onome kod sudara sfera-sfera. Za $k = 1$ reflektirani vektor v_r jednake je duljine kao ulazni vektor v, što odgovara savršeno elastičnom sudaru bez gubitka energije.

$$v_r = \mathrm{v} - \left(1 + \mathrm{k}\right) \cdot v_{\mathrm{rel}} \cdot n \tag{19}$$

2. Arhitektura sustava

Programski dio rada izrađen je na objektno orijentiran način. Postoje tri glavne klase: *Mesh*, *Camera* i *Scene*, te klasa *Sphere* koja nasljeđuje klasu *Mesh*. Dijagram klasa prikazan je na slici (Sl. 2.1). Pomoću tih klasa postavlja se cijelo virtualno okruženje — dodaju se kamera i objekti u scenu, a objekti se učitavaju iz datoteka u .obj formatu. Za reprezentaciju 3D vektora koristi se tip podataka *glm::vec3* i *glm::ivec3* iz biblioteke OpenGL Mathematics (GLM), koja pruža sve osnovne vektorske i matrične operacije navedene u poglavljima 1.1. i 1.2.



Sl. 2.1 UML dijagram klasa

2.1. Klasa *Mesh*

Svaka instanca klase *Mesh* sastoji se od liste vrhova i liste trokuta. Za obje liste koristi se standardna biblioteka *std::vector*. U listi *vertices* nalaze se trodimenzionalni vektori koji reprezentiraju pozicije vrhova, a u listi faces nalaze se cjelobrojni vektori čije tri komponente predstavljaju indekse vrhova iz liste vertices, indeksirano od nule.

Učitavanje podataka iz .obj datoteke obavlja metoda loadMesh(Sl. 2.1.1), koja kao argument prima putanju do datoteke. Metoda čita datoteku redak po redak te na temelju prefiksa određuje vrstu podatka — prefiks v označava vrh koji se dodaje u listu vertices, a prefiks f označava trokut čiji se indeksi vrhova dodaju u listu faces. Ovu klasu koristimo za objekte koji su statični u simulaciji.

```
int loadMesh(const std::string &filepath) {
  cout << "Loading file: " << filepath << endl;
  ifstream file(s: filepath);
  if (!file.is_open()) {
    cout << "Failed to open file: " << filepath << endl;
    return 1;
  }

  string line;
  while (getline(&is: file, &str: line)) {
    istringstream ss(str: line);
    string prefix;
    ss >> prefix;

    if (prefix == "v") {
      float x, y, z;
      if (ss >> x >> y >> z) {
        this->vertices.push_back(x: glm::vec3(x, y, z));
      }
    } else if (prefix == "f") {
      string v1, v2, v3;
      if (ss >> v1 >> v2 >> v3) {
        int idx1 = stoi(str: v1.substr(pos: 0, n: v1.find(c: '/')));
        int idx2 = stoi(str: v2.substr(pos: 0, n: v2.find(c: '/')));
        int idx3 = stoi(str: v3.substr(pos: 0, n: v3.find(c: '/')));
        this->faces.push_back(x: glm::ivec3(x: idx1 - 1, y: idx2 - 1, z: idx3 - 1));
      }
    }
  }
  cout << "Loaded " << vertices.size() << " vertices, " << faces.size()
       << " faces" << endl;
  return 0;
}
```

Sl. 2.1.1 Metoda loadMesh

Za translaciju i skaliranje objekata uvedene su dvije metode: *moveMesh* i *scaleMesh*. Metoda *moveMesh* prima vektor translacije kao argument te svaki vrh iz liste vertices pomiče za taj vektor. Metoda *scaleMesh* prima skalarnu vrijednost kao argument te svaki vrh iz liste vertices skalarno množi s tom vrijednošću. Važno je napomenuti da se skaliranje vrši oko ishodišta koordinatnog sustava — ako objekt nije centriran u ishodištu, skaliranje će uz promjenu veličine uzrokovati i pomak objekta.

2.2. Klasa *Sphere*

Klasa *Sphere* nasljeđuje klasu *Mesh* te proširuje ju novim atributima: radijusom, masom, vektorom brzine i vektorom centra. Konstruktor klase *Sphere* automatski poziva metodu *loadMesh* s datotekom *sphere.obj*, koja je dobivena izvozom Blender icosphere centrirane u ishodištu s radijusom 1 u .obj formatu, pa je defaultna vrijednost radijusa 1.

Budući da klasa Sphere uvodi eksplicitan atribut centra i radijusa koji moraju ostati konzistentni s listom vertices, metode moveMesh i scaleMesh se nadjačavaju. Nadjačana metoda moveMesh uz translaciju svih vrhova pomiče i atribut centra za isti vektor, a nadjačana metoda scaleMesh uz skaliranje svih vrhova skalira i atribut radijusa. Ovu klasu koristimo za objekte koji su dinamični u simulaciji.

2.3. Klasa *Camera*

Klasa *Camera* opisuje kameru u sceni i sastoji se od vektora očišta *eye*, gledišta *looking_at*, vektora *up*, vektora smjera gledanja *front* i desnog vektora *right*, te kutova *yaw* i *pitch* koji opisuju rotaciju kamere. Vektor *up* označava u kojem smjeru je gore u svijetu, u ovom slučaju taj vektor će bit (0,1,0) što znaći da y os označava gore-dolje. Konstruktor prima očište, gledište i vektor gore te iz njih izračunava vektore *front* i *right* prema formulama iz poglavlja 1.2.

Za kretanje kamere koristi se metoda *moveCameraInDirection* koja prima brzine u smjeru naprijed, desno i gore te vremenski korak *deltaTime*. Važno je napomenuti da se kretanje naprijed i desno vrši samo u horizontalnoj ravnini — ignorira se y komponenta vektora *front* i *right* — čime se postiže kretanje nalik hodanju po tlu.

Za rotaciju kamere koristi se metoda *rotate* koja prima promjene kutova *yaw* i *pitch*. Kut *yaw* opisuje rotaciju kamere oko vertikalne osi (lijevo-desno), a kut *pitch* opisuje rotaciju oko horizontalne osi (gore-dolje). Kut *pitch* je ograničen na interval (−89°, 89°) kako bi se izbjeglo preokretanje kamere u slučaju da *pitch* dosegne ±90°. Na temelju ažuriranih kutova novi vektor *front* izračunava se sfernim koordinatama (Kôd 2.3.1)

```
void rotate(float yawDelta, float pitchDelta) {
yaw += yawDelta;
pitch += pitchDelta;
if (pitch > 89.0f)
pitch = 89.0f;
if (pitch < -89.0f)
pitch = -89.0f;
glm::vec3 newFront;
newFront.x =cos(glm::radians(yaw))*cos(glm::radians(pitch));
newFront.y = sin(glm::radians(pitch));
newFront.z = sin(glm::radians(yaw))*cos(glm::radians(pitch));
front = glm::normalize(newFront);
right = glm::normalize(glm::cross(front, up));
looking_at = eye + front;
}
```

Kôd 2.3.1 – Metoda rotate

Intuitivno, *cos(pitch)* određuje koliko je vektor *front* naklonjen u horizontalnoj ravnini, a *sin(pitch)* određuje njegovu vertikalnu komponentu. Kut *yaw* zatim raspoređuje horizontalnu komponentu između osi x i z. Nakon što se izračuna novi vektor *front*, desni vektor *right* ažurira se kao vektorski produkt vektora *front* i *up*, analogno formuli (6) iz poglavlja 1.2. Vektor *up* ostaje nepromijenjen kroz rotaciju jer kamera ne rotira oko osi gledanja.

2.4. Klasa *Scene*

Klasa *Scene* središnja je klasa koja upravlja cijelim virtualnim okruženjem. Sastoji se od dvije liste objekata — *staticMeshes* koja sadrži statične objekte scene i *movingMeshes* koja sadrži sfere podložne fizikalnoj simulaciji — te pokazivača na kameru. Uz to, klasa održava dvije zajedničke liste *sceneVertices* i *sceneFaces* koje sadrže sve vrhove i trokute cijele scene u formatu pogodnom za slanje na GPU. Svaki vrh u listi *sceneVertices* predstavljen je sa šest vrijednosti — prve tri označavaju koordinate vrha (x, y, z), a sljedeće tri označavaju boju vrha (r, g, b). Lista *sceneFaces* sadrži indekse vrhova grupirane u trojke, pri čemu svaka trojka definira jedan trokut. Detalji korištenja tih lista za renderiranje opisani su u poglavlju o renderiranju.

Za dodavanje objekata u scenu koriste se metode *addToStaticScene* i *addToMovingScene*. Obje metode objekt dodaju u odgovarajuću listu te pozivaju internu metodu *addToScene* koja vrhove i trokute dodanog objekta upisuje u zajedničke liste *sceneVertices* i *sceneFaces*. Pri tome se vodi računa o pomaku indeksa kako bi indeksi u listi *sceneFaces* ispravno pokazivali na vrhove u listi *sceneVertices*.

Budući da se pozicije sfera mijenjaju svakim korakom simulacije, zajedničke liste *sceneVertices* i *sceneFaces* potrebno je osvježiti prije svakog iscrtavanja. To obavlja metoda *updateScene* koja liste prvo isprazni, a zatim ih ponovno popuni podacima svih statičnih i dinamičnih objekata u njihovim trenutnim pozicijama.

Metoda *shootBall (*Kôd 2.4.1) omogućuje ispucavanje sfere iz pozicije kamere u smjeru gledanja. Masa sfere izračunava se iz radijusa prema formuli za masu kugle uz gustoću 0.9, a početna brzina postavlja se u smjeru vektora *front* kamere.

```
void shootBall(float radius, float speed) {
float mass = 0.9 * (4.0f / 3.0f) * M_PI * radius * radius *
radius;
Sphere sphere(speed * camera->front, camera->eye, radius,
mass);
this->addToMovingScene(sphere);
}
```

Kôd 2.4.1 – Metoda shootBall

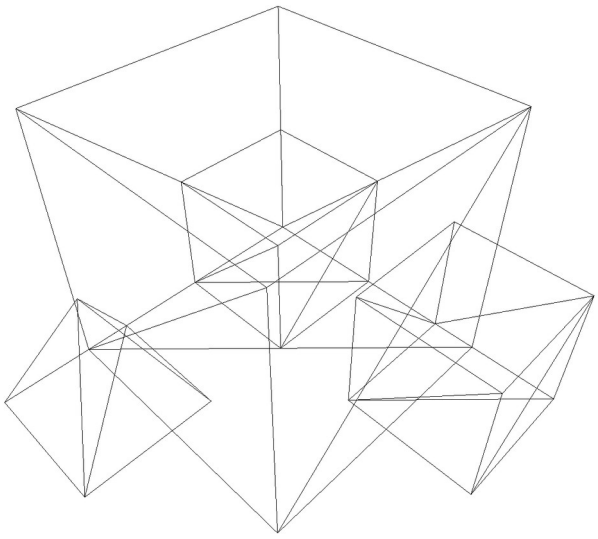
Klasa *Scene* također sadrži dvije metode vezane uz fizikalnu simulaciju. Statična metoda *closestPointOnTriangle* implementira algoritam opisan u poglavlju 1.3, a metoda *animateScene* provodi jedan korak fizikalne simulacije za sve dinamične objekte scene, što je detaljno opisano u narednom poglavlju o fizikalnoj simulaciji.

3. Fizikalna simulacija

Za pokretanje simulacije potrebna je glavna petlja koja se izvršava svaki vremenski okvir. U svakoj iteraciji petlje objekti se pomiču prema svojim vektorima brzine, provjeravaju se i razrješavaju kolizije sfera-sfera i sfera-statični objekt, te se ažuriraju zajedničke liste vrhova i trokuta za renderiranje. Uz fizikalnu simulaciju, petlja obrađuje i korisnički unos — upravljanje kamerom te ispucavanje sfera u scenu postavljenu od statičnih objekata.

3.1. Petlja animacije i vremenski korak

Glavna petlja animacije nalazi se u funkciji main te se izvršava sve dok korisnik ne zatvori prozor. Prije pokretanja petlje u scenu(Sl. 3.1.1) se pomoću metode *addToStaticScene* dodaju statični objekti — tri kocke bez gornje stranice i jedna piramida — koji čine okruženje po kojemu se sfere gibaju i odbijaju.



Sl. 3.1.1 Scena statičnih objekata

Svaka iteracija petlje počinje mjerenjem vremenskog koraka *deltaTime*, koji predstavlja vrijeme proteklo između dvije uzastopne iteracije. Korištenje *deltaTime* pri ažuriranju pozicija i brzina osigurava da se simulacija odvija jednako brzo neovisno o brzini izvođenja programa — bez toga bi simulacija bila brža na bržim računalima i sporija na sporijima.

Kretanje kamere obrađuje se funkcijom *glfwGetKey*. Tipkama W, A, S, D kamera se giba naprijed, lijevo, nazad i desno u horizontalnoj ravnini, tipkom Space se diže, a tipkom Left Shift spušta. Držanjem Left Control udvostručuje se brzina kretanja. Za svaki pritisnuti smjer poziva se metoda *moveCameraInDirection* s odgovarajućom brzinom i *deltaTime* kao argumentima, koja ažurira poziciju očišta kamere *eye* i gledišta *looking_at* prema smjerovima *front*, *right* i *up* opisanim u poglavlju 1.2. Rotacija kamere obrađuje se funkcijom *glfwGetCursorPos* koja vraća trenutnu poziciju miša — razlika između trenutne i prethodne pozicije miša izračunava se kao *dx* i *dy* te se prosljeđuje metodi *camera.rotate* kao promjena po *yaw* i *pitch*. Desnom tipkom miša aktivira se skrivanje kursora, a tipkom Escape kursor se vraća.

Veličina ispucane sfere mijenja se kotačićem miša putem callback funkcije *scrollCallback* koja ažurira globalnu varijablu *scrollOffset*. Lijevom tipkom miša ispucava se veća sfera brzinom 0.2, a desnom tipkom manja sfera brzinom 0.07, pri čemu je između dva uzastopna ispucavanja uvedena zaštita od 150 ms kako se ne bi ispucalo previše sfera odjednom. Tipkom Q brišu se sve dinamične sfere iz scene, a srednja tipka miša pauzira i nastavlja simulaciju.

Na kraju svake iteracije pozivaju se redom *animateScene* za fizikalnu simulaciju, *updateScene* za ažuriranje podataka za GPU te OpenGL pozivi za renderiranje, što je detaljnije opisano u poglavlju 4.

3.2. Razrješavanje sudara

Fizikalna simulacija odvija se unutar metode *animateScene*. Na početku svake iteracije simulira se gravitacija — y komponenta brzine svake sfere smanjuje se za konstantni iznos gravitacijskog ubrzanja proporcionalno *deltaTime*. Time se postiže kontinuirano ubrzavanje sfere prema dolje kroz svaki frame, analogno slobodnom padu u stvarnom svijetu. Nakon primjene gravitacije svaka sfera pomiče se prema svom trenutnom vektoru brzine prema izrazu (12), također skalirano s *deltaTime*.

Razrješavanje sudara provodi se u pet iteracija kako bi se smanjile numeričke pogreške nastale zbog diskretnog vremenskog koraka — u jednom koraku simulacije sfera se može toliko pomaknuti da duboko prodre u drugi objekt, pa je više iteracija potrebno za ispravnu korekciju pozicija i brzina.

Za svaki par dinamičnih sfera provjerava se uvjet sudara prema izrazu (13). Ako je uvjet zadovoljen i relativna brzina prema izrazu (14) negativna, što znači da se sfere gibaju jedna prema drugoj, izračunava se impuls prema izrazu (15) te se ažuriraju vektori brzina obiju sfera prema izrazima (16) i (17). Uz korekciju brzine provodi se i pozicijska korekcija (Kôd 3.2.1) — sfere se razmiču duž normale sudara razmjerno svojim masama kako bi se uklonilo međusobno preklapanje nastalo zbog diskretnog vremenskog koraka.

```
float penetration = rsum - dist;
if (penetration > 0) {
glm::vec3 correction = (penetration / (m1 + m2)) * n;
movingMeshes[i].moveMesh(correction * m2);
movingMeshes[j].moveMesh(-correction * m1);
}
```

Kôd 3.2.1 – Računanje korekcije sudara sfera-sfera

Za svaku sferu prolazi se kroz sve trokute svih statičnih objekata. Za svaki trokut pronalazi se najbliža točka *p* od centra sfere algoritmom opisanim u poglavlju 1.3. Ako je uvjet sudara prema izrazu (17) zadovoljen, izračunava se normala *n* prema izrazu (18) te se provjerava relativna brzina. Ako je sfera usmjerena prema površini, ažurira se vektor brzine prema izrazu (19). Analogno sudaru sfera-sfera, provodi se i pozicijska korekcija pomicanjem sfere duž normale za iznos penetracije kako bi se uklonilo preklapanje(Kôd 3.2.2).

```
glm::vec3 p =
closestPointOnTriangle(v1, v2, v3, movingMeshes[i].center);
float dist = glm::length(movingMeshes[i].center - p);
float pen = dist - movingMeshes[i].radius;
if (pen < 0) {
glm::vec3 n = glm::normalize(movingMeshes[i].center - p);
glm::vec3 vel = movingMeshes[i].vel;

float vrel = glm::dot(vel, n);
if (vrel < 0) {
movingMeshes[i].vel = vel - (1.0f + restitution) * vrel * n;
}
movingMeshes[i].moveMesh(-pen * n);
}
```

Kôd 3.2.2 – Računanje reflektirane brzine i korekcije sudara sfera-trokut

Poznato ograničenje ovog pristupa je problem tuneliranja (*eng. tunneling*) — pri dovoljno velikim brzinama sfera može u jednom koraku simulacije prijeći toliki put da potpuno prođe kroz statični objekt, a da detekcija sudara nije aktivirana jer se položaj sfere provjerava samo na kraju svakog koraka. Ovaj problem spominje se u izvoru [1] u poglavlju 2.4.3 u kontekstu diskretne detekcije sudara. U ovoj implementaciji problem je djelomično ublažen petljom od pet iteracija razrješavanja sudara po koraku, no nije potpuno riješen. Potpuno rješenje zahtijevalo bi kontinuiranu detekciju sudara (*eng. continuous collision detection*) koja provjerava cijelu putanju sfere između dva koraka, a ne samo njezin položaj na kraju koraka.

4. Izrada prikaza

Za izradu prikaza scene koristi se OpenGL — API za hardverski ubrzano renderiranje koji omogućuje izvođenje grafičkog koda direktno na GPU-u. U ovoj simulaciji koristi se OpenGL verzija 4.6 zajedno s bibliotekama GLAD za učitavanje OpenGL funkcija i GLFW za upravljanje prozorom i korisničkim unosom.

4.1. Učitavanje sjenčara

Sjenčari su programi koji se izvršavaju direktno na GPU-u i definiraju kako se vrhovi i pikseli renderiraju. U ovoj simulaciji koriste se dva sjenčara — vertex sjenčar i fragment sjenčar — koji se učitavaju iz datoteka *shader.vert* i *shader.frag* funkcijom *make_shader*.

Funkcija *make_shader* poziva pomoćnu funkciju *make_module* za svaki sjenčar posebno. Funkcija *make_module* otvara datoteku sjenčara, čita njen sadržaj u string te ga prosljeđuje OpenGL funkciji *glShaderSource*. Sjenčar se zatim kompajlira pozivom *glCompileShader*, a u slučaju greške ispisuje se poruka o grešci putem *glGetShaderInfoLog*. Nakon što su oba sjenčara uspješno kompajlirana, funkcija *make_shader* stvara program sjenčara pozivom *glCreateProgram*, prilaže oba sjenčara pozivom *glAttachShader* te ih povezuje pozivom *glLinkProgram*. Kompajlirani sjenčar moduli se nakon toga brišu jer su njihovi podaci već preneseni u program sjenčara.

4.2. Vertex i fragment sjenčar

Vertex sjenčar (Kôd 4.2.1) izvršava se jednom za svaki vrh objekta. Prima dva ulazna atributa na lokacijama 0 i 1 — *vertexPos* koji sadrži trodimenzionalnu poziciju vrha i *vertexColor* koji sadrži RGB boju vrha, što odgovara formatu liste *sceneVertices* opisanom u poglavlju 2. Uz to prima dvije uniformne matrice — *view* i *projection* — koje se postavljaju iz glavne petlje pozivima glUniformMatrix4fv. Konačna pozicija vrha u prostoru ekrana izračunava se množenjem pozicije vrha matricom pogleda i matricom projekcije prema izrazu (11) iz poglavlja 1.2 i upisuje se u ugrađenu varijablu gl_Position. Boja vrha prosljeđuje se dalje fragment sjenčaru kroz izlaznu varijablu *fragmentColor*.

```
#version 460 core

layout (location = 0) in vec3 vertexPos;
layout (location = 1) in vec3 vertexColor;

out vec3 fragmentColor;

uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * vec4(vertexPos, 1.0);
    fragmentColor = vertexColor;
}
```

Kôd 4.2.1 – datoteka shader.vert

Matrica pogleda i matrica perspektivne projekcije izračunavaju se svake iteracije fizikalne petlje i šalju se na GPU kao uniformne varijable *view* i *projection*.

Matrica pogleda izračunava se pozivom *glm::lookAt* koji prima tri argumenta — poziciju očišta kamere *eye*, gledište *looking_at* i vektor gore *up*. Funkcija interno izračunava matricu translacije i matricu rotacije opisane izrazima (7) i (9) te vraća njihov produkt prema izrazu (11) iz poglavlja 1.2.

Matrica perspektivne projekcije izračunava se pozivom *glm::perspective* koji prima kut vidnog polja od 45°, omjer širine i visine prozora 1920/1080, te vrijednosti bliže i dalje ravnine rezanja 0.1 i 100.0. Objekti koji se nalaze izvan tog raspona dubine neće biti prikazani. Funkcija interno izračunava matricu perspektivne projekcije analognu matrici M_P opisanoj izrazom (10) iz poglavlja 1.2.

Obje matrice šalju se na GPU pozivom g*lUniformMatrix4fv* te su dostupne vertex sjenčaru kao uniformne varijable *view* i *projection*, gdje se koriste za transformaciju pozicije svakog vrha prema izrazu (11).

Fragment sjenčar (Kôd 4.2.2) izvršava se jednom za svaki piksel koji pokriva neki trokut. Prima interpoliranu boju *fragmentColor* od vertex sjenčara — OpenGL automatski interpolira vrijednosti između vrhova trokuta za svaki piksel. Izlazna boja piksela *outColor* postavlja se na tu interpoliranu boju uz alpha vrijednost 1.0, što označava potpunu neprozirnost.

```
#version 460 core

in vec3 fragmentColor;
out vec4 outColor;

void main() {
    outColor = vec4(fragmentColor, 1.0);
}
```

Kôd 4.2.2 – datoteka shader.frag

4.3. Objekti VAO,VBO,EBO

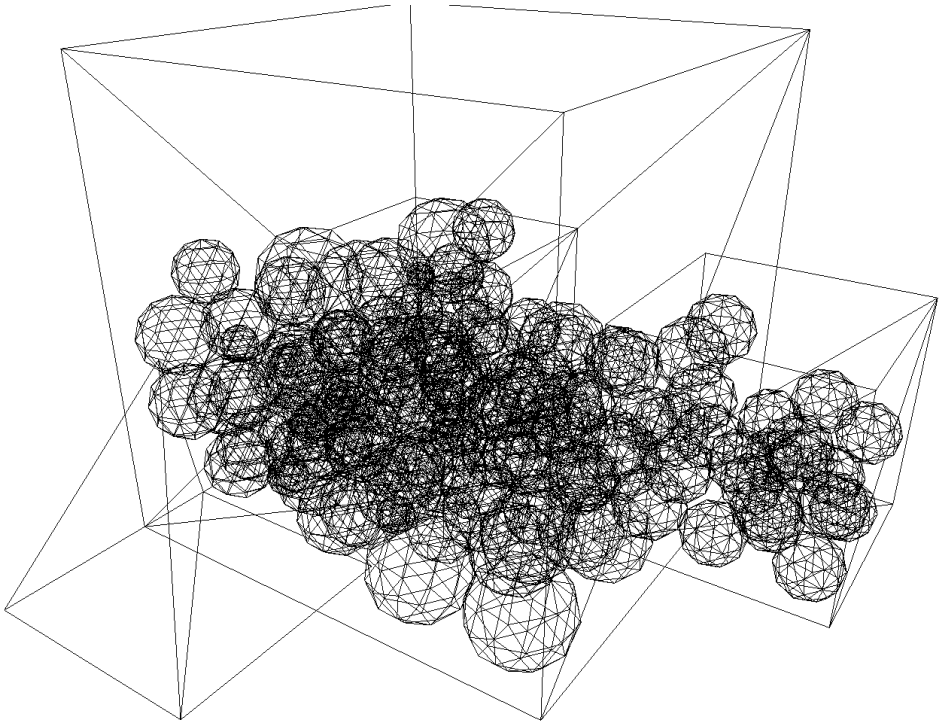
Podaci o vrhovima i trokutima šalju se na GPU putem tri objekta — VAO (*Vertex Array Object*), VBO (*Vertex Buffer Object*) i EBO (*Element Buffer Object*). VBO sadrži listu *sceneVertices* s pozicijama i bojama svih vrhova, a EBO sadrži listu *sceneFaces* s indeksima vrhova koji definiraju trokute. VAO pamti kako su podaci u VBO-u organizirani — u ovom slučaju svaki vrh zauzima 6 floatova, pri čemu prvih 3 odgovaraju poziciji i šalju se na lokaciju 0 vertex sjenčara, a sljedećih 3 odgovaraju boji i šalju se na lokaciju 1. Ova organizacija postavlja se pozivima *glVertexAttribPointer* i *glEnableVertexAttribArray*. Budući da se pozicije sfera mijenjaju svakim frameom, podaci se svake iteracije iznova šalju na GPU pozivima *glBufferData*(Sl. 4.3.1). Na kraju iteracije pozivom *glDrawElements* OpenGL iscrtava sve trokute scene koristeći podatke iz VAO-a, a gotov frame prikazuje se na ekranu pozivom *glfwSwapBuffers*.Slanje podataka na GPU putem VBO i EBO

```
glBindVertexArray(array: VAO);
glBindBuffer(target: GL_ARRAY_BUFFER, buffer: VBO);
glBufferData(target: GL_ARRAY_BUFFER, size: sizeof(float) * scene.sceneVertices.size(),
             data: scene.sceneVertices.data(), usage: GL_STATIC_DRAW);
glBindBuffer(target: GL_ELEMENT_ARRAY_BUFFER, buffer: EBO);
glBufferData(target: GL_ELEMENT_ARRAY_BUFFER,
             size: sizeof(unsigned int) * scene.sceneFaces.size(),
             data: scene.sceneFaces.data(), usage: GL_STATIC_DRAW);
```

Sl. 4.3.1 Slanje podataka na GPU putem VBO i EBO

5. Rezultati

Što se tiče performansi, nakon ispaljivanja 100 sfera program počinje osjetno usporavati. Također, ako su kugle dovoljno malog radijusa, a frekvencija osvježavanja (framerate) niska zbog vremenskog pomaka *deltaTime*, kolizije postaju sve manje precizne. U određenim slučajevima dolazi do pojave tuneliranja (tunneling), odnosno prolaska kugli kroz statične objekte, što je ranije spomenuto u poglavlju 3.2. Program je testiran na računalu Lenovo Legion 5 15ARH7H s NVIDIA RTX 3060 Mobile grafičkom karticom i AMD Ryzen 5 6600H procesorom takta 4,57 GHz na Arch Linuxu.



Sl. 5.1 Simulacija od 130 kugla i 4 statična objekta

Zaključak

U ovom radu implementirana je interaktivna fizikalna simulacija sudara čvrstih tijela u trodimenzionalnom prostoru. Cilj rada bio je izraditi simulaciju koja fizikalno točno modelira gibanje sfera pod utjecajem gravitacije te njihovo međusobno odbijanje i odbijanje od statičnih objekata, uz vizualni prikaz u stvarnom vremenu.

Teorijska osnova rada obuhvatila je vektorski račun kao temelj svih fizikalnih i grafičkih izračuna, transformacijske matrice potrebne za postavljanje kamere i projiciranje trodimenzionalne scene na dvodimenzionalnu ravninu prikaza te fizikalni model gibanja i sudara temeljen na impulsu i očuvanju količine gibanja. Za detekciju sudara sfere i statičnog objekta implementiran je algoritam pronalaženja najbliže točke na trokutu temeljen na Voronoijevim regijama prema Ericsonovom radu *Real-Time Collision Detection* [1], koji ispravno obrađuje sva tri slučaja — odbijanje od površine trokuta, od brida i od vrha.

Programska arhitektura organizirana je u četiri klase — *Mesh, Sphere, Camera* i Scene — koje zajedno upravljaju učitavanjem objekata, fizikalnom simulacijom i prijenosom podataka na GPU. Renderiranje je ostvareno putem OpenGL pipeline-a s VAO, VBO i EBO strukturama te jednostavnim vertex i fragment sjenčarima koji transformiraju vrhove u prostor ekrana i bojaju ih prema zadanim bojama vrhova.

Simulacija postiže fizikalno prihvatljive rezultate za standardne scenarije — sfere se ispravno odbijaju jedna od druge i od statičnih objekata uz realistično ponašanje pri različitim masama i veličinama. Ipak, postoji poznato ograničenje vezano uz diskretnu prirodu simulacije — problem tuneliranja, opisan u poglavlju 3.2, pri čemu sfera pri dovoljno velikim brzinama može proći kroz statični objekt bez aktiviranja detekcije sudara. Potpuno rješenje ovog problema zahtijevalo bi implementaciju kontinuirane detekcije sudara, što predstavlja prirodan smjer daljnjeg razvoja ovog rada. Uz to, simulacija podržava isključivo sfere kao dinamične objekte, pa bi proširenje na proizvoljne konveksne oblike bio još jedan mogući nastavak.

Izradom ovog rada stečeno je praktično iskustvo u primjeni fizikalnih principa u računalnoj simulaciji, razvoju grafičkih aplikacija s OpenGL-om te organizaciji većeg C++ projekta. Rezultat je funkcionalna interaktivna simulacija koja demonstrira temeljne koncepte fizikalnih simulacija u stvarnom vremenu.

Literatura

[1] C. Ericson, *Real-Time Collision Detection*. San Francisco, CA: Morgan Kaufmann, 2005.

[2] Wikipedia, "Physics engine," *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Physics_engine. [Accessed: 10. 6. 2026.]

[3] Github link na source kod od rada: https://github.com/katic123/ZavRad

Izjava o korištenju umjetne inteligencije

Osvrt na korištenje umjetne inteligencije: U izradi ovog završnog rada korišteni su alati umjetne inteligencije Claude (Anthropic) i DeepSeek. Alati su korišteni isključivo kao pomoćni alati za jezično uređivanje teksta završnog rada — poboljšanje stila, gramatike i čitljivosti već napisanog sadržaja. Cjelokupna ideja rada, arhitektura sustava, implementacija programskog koda te struktura i sadržaj pojedinih poglavlja rezultat su isključivo vlastitog rada i razumijevanja. Pri implementaciji fizikalnog modela i algoritama povremeno je korištena umjetna inteligencija za provjeru ispravnosti pojedinih dijelova koda, no sve konačne odluke o dizajnu i implementaciji donesene su samostalno. Nisu uočene halucinacije niti netočnosti koje bi utjecale na sadržaj rada.

Izjava o korištenju umjetne inteligencije: Potvrđujem da sam tijekom izrade ovog završnog rada koristio umjetnu inteligenciju u skladu s Politikom primjerenog korištenja umjetne inteligencije na Fakultetu elektrotehnike i računarstva, te da umjetna inteligencija nije zamijenila moje kritičko razmišljanje niti moj doprinos.