UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Simularea interacțiunilor fizice
TODO:Subtitlu (ex: versiunea 2018)

Cristian-Andrei SANDU

Coordonator științific:

Prof. dr. ing. Costin-Anton BOIANGIU

BUCUREȘTI

2018

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Diploma Project Title (eg: Diploma project template)
TODO:Subtitle (eg: 2018 version)

Cristian-Andrei SANDU

Thesis advisor:

Prof. dr. ing. Costin-Anton BOIANGIU

BUCHAREST

2018

CUPRINS

SINOPSIS

Lucrarea de față are obiectivul de a prezenta o serie de fenomene fizice ce țin de cinematica corpurilor solide, diferiți algoritmi și metode numerice utilizate pentru a simula aceste fenomene și punerea lor în aplicare sub forma unui simulator de interacțiuni mecanice, capabil să aproximeze și să afișeze în timp real mișcarea realistică a unui număr obiecte pe ecran. Rezultatul este o aplicație OpenGL capabilă să ruleze o serie de demo-uri și care oferă utilizatorului posibilitatea de a interacționa cu acestea(să vizualizeze scena, să poată introduce obiecte noi și să poată modifica parametri).

ABSTRACT

The aim of this thesis is to provide a closer look at a series of physical phenomena pertaining to the motion of solid objects, as well as to describe various algorithms and numerical methods used in motion simulation and implementing them inside a physics engine able to approximate and render the realistic motion of a number of objects in real time. The achieved result is an OpenGL app able to run a series of demos and also provide the user with means for interacting with them(view the scene, insert new objects or change simulation parameters).

MULȚUMIRI

TODO:(opțional) Aici puteți introduce o secțiunea specială de mulțumiri / acknowledgments.

1 INTRODUCERE

Simularea interacțiunilor fizice pe un computer se realizează pe baza unui **motor de fizică**(eng. *physics engine*) care are rolul de a prelua starea scenei la un moment discret de timp t_i și de a determina starea la momentul t_{i+1}. Întrucât acest lucru se realizează într-un spațiu discret, se dorește de fapt obținerea unei aproximări cât mai bună a fenomenelor fizice din realitate. Pentru că fizica este un domeniu extrem de vast, lucrarea de față are ca subiect doar simularea interacțiunilor mecanice dintre corpuri, lăsând aprofundarea altor tipuri de forțe și fenomene la latitudinea cititorilor interesați de domeniu.

1.1 Context

Proiectul a luat naștere ca urmare a interesului autorului pentru mecanică și grafică pe calculator și dorința de a aprofunda pașii necesari implementării unui sistem informatic robust, realistic, ușor de folosit și plăcut vederii. Aplicabilitatea unui astfel de simulator se reflectă într-o multitudine de domenii: jocuri video, proiectare și testare de sisteme mecanice(complexitatea variind de la angrenaje simple până la mașini, avioane), balistică(de natură militară sau civilă), didactică(prin oferirea unei perspective ușor de urmărit și de înțeles în studiul mecanicii).

1.2 Problema

Se dorește obținerea unei aplicații care să ofere utilizatorului capabilitatea de a rula simulări pentru niște scenarii definite programatic și editabile în timpul execuției printr-o interfață grafică. Se disting, prin urmare, două subprobleme de rezolvat:

1. **motorul de fizică** care să simuleze mișcarea și interacțiunile corpurilor din scenă
2. **interfața cu utilizatorul**

1.3 Obiective

În continuarea celor spuse anterior, sunt delimitate următoarele obiective atinse în elaborarea lucrării de față și a aplicației asociate:

- alegerea unei reprezentări robuste pentru starea(din punct de vedere cinematic) unui obiect al scenei
- integrarea mărimilor secundare(de ex. accelerația, viteza) în vederea obținerii stării noi a obiectului
- detecția potențialelor coliziuni între obiecte
- generarea punctelor de contact între obiectele aflate în coliziune
- rezolvarea contactelor generate cu un răspuns realistic
- desenarea obiectelor pe ecran
- implementarea unui algoritm de ray-casting pentru selectarea unui obiect de pe ecran cu ajutorul mouse-ului
- implementarea unei interfețe grafice pentru controlul simulării și modificarea de elemente ale scenei

1.4 Soluția propusă

În vederea atingerii tuturor obiectivelor de mai sus, este propusă o aplicație OpenGL, capabilă să preia input-ul utilizatorului și să deseneze un număr de scenarii demonstrative. Backend-ul (motorul de fizică în sine) va urmări o arhitectură clasică, folosită cu succes în alte proiecte asemănătoare(ex: box2D[2], bullet[3]). Funcționarea simulatorului este asigurată de o buclă infinită în care la fiecare iterație sunt realizate, pe rând: tratarea input-ului utilizatorului, integrarea(actualizarea stării) corpurilor solide, detecția coliziunilor, rezolvarea lor, desenarea în contextul OpenGL.

1.5 Rezultatele obținute

TODO:Descriere pe scurt a rezultatelor obținute, eventual de ce acestea sunt importante față de alte soluții sau studii.

1.6 Structura lucrării

În continuare voi prezenta pe scurt fiecare secțiune a acestei lucrări, care urmărește, în mare, șablonul oficial.

2. **Motivație și analiza cerințelor**: sunt detaliate atât motivația realizării proiectului propus, cât și funcționalitățile oferite de aplicație, în raport cu cerințele care trebuie acoperite.
3. **Metode existente**: sunt analizate metodele disponibile pentru atingerea fiecăruia dintre obiectivele propuse, modul în care acestea sunt folosite în soluții similare și o evaluare a acestor metode. Fiecare subproces al simulatorului va avea propria subsecțiune.
4. **Soluția propusă**: sunt motivate alegerile și deciziile luate la nivel structural, iar soluția va fi descrisă pe larg, din punct de vedere teoretic.
5. **Detalii de implementare**: este prezentată arhitectura aplicației și orice detalii de implementare considerate a fi relevante(algoritmi folosiți, etapele dezvoltării - cu dificultăți întâmpinate și soluții descoperite)
6. **Evaluare**: analiză a performanțelor aplicației și a gradului de atingere a obiectivelor propuse
7. **Concluzii**: este sumarizat întregul proiect, trecând din nou peste elementele constituente(obiective, implementare, rezultate obținute); în plus, sunt oferite perspective pentru dezvoltarea ulterioară a proiectului.

2 MOTIVAȚIE

Motivul pentru care am ales să realizez proiectul de față este una personală, aceea de a aprofunda tehnicile matematice și programatice folosite într-o simulare realistică a interacțiunilor mecanice dintre corpuri 3D. Pe parcursul realizării acestuia, am avut în vedere și posibilitatea îmbunătățirii uneia sau mai multora dintre aceste tehnici.

Faptul că în sfera dezvoltării de aplicații grafice sau jocuri accentul se pune mai ales pe partea practică – e de preferat o aplicație care rulează mai bine și care este plăcută ochiului decât una foarte riguroasă din punct de vedere teoretic, a dat naștere multor soluții mai mult sau mai puțin "hacky", dar foarte interesante. Am ale astfel să nu urmăresc obținerea de rezultate teoretice remarcabile, și să mă concentrez pe înțelegerea și implementarea unor astfel de metode.

Proiectul poate fi considerat, în fapt, o "testare a apelor" în domeniul simulării de fizică în timp real, un exercițiu pentru abilitățile mele de programare eficientă, robustă, orientată pe obiecte, dar și o îmbunătățire a cunoștințelor mele de C++.

Consider că ce am dobândit în urma realizării acestui proiect mă va ajuta să înțeleg mai bine subtilitățile din spatele unui motor de fizică state-of-the-art, astfel încât, pe viitor, să fiu capabil de a contribui la proiecte open-source deja existente(bullet[3]) sau, de ce nu, să efectuez muncă în cercetare sau în industrie în acest domeniu, la un nivel ceva mai complex și actual.

TODO: ar mai trebui ceva aici?

3 STUDIU DE PIAȚĂ / METODE EXISTENTE

Pentru început, ar trebui menționat că un motor de fizică este foarte rar întâlnit de sine stătător, el constituind de cele mai multe ori o parte esențială a unei aplicații mult mai complexe(care cuprinde și alte motoare/instrumente necesare funcționării). De aceea, voi ignora sisteme precum motoarele pentru dezvoltarea jocurilor(Unreal Engine, Source, Unity etc.) sau simulatoare științifice și mă voi concentra strict pe expunerea particularităților motoarelor de fizică de sine stătătoare.

3.1 Aplicații similare

Deoarece cele mai multe aplicații nu au nevoie de toate particularitățile unui motor de fizică complex și foarte general, o practică des întâlnită este ca companiile să își implementeze unul propriu, optimizat pentru cerințele specifice ale aplicațiilor dezvoltate. Chiar și așa, putem aminti câteva dintre cele mai populare middleware-uri:

- **Box2D**[2] este un motor de fizică open source, capabil să simuleze corpuri solide în 2D. Oferă suport pentru detecție continuă a coliziunilor, poligoane convexe și cercuri, corpuri compuse, soluționarea contactelor cu frecare, articulațiilor etc. TODO: menționează arbore dinamic pentru broadphase. Scris în C++, a fost ulterior portat și în alte limbaje și este apreciat pentru simplitatea lui și folosit de dezvoltatori independenți, și a fost inclus chiar și în Unity ca opțiune pentru motorul de fizică 2D.
- **Bullet**[3] este probabil cel mai cunoscut motor de fizică open source în lumea 3D. Oferă suport pentru detecția discretă sau continuă a coliziunilor pentru toate primitivele de bază, dar și meshe convexe, simularea corpurilor deformabile, articulații și constrângeri complexe, mișcarea vehiculelor etc. Este folosit în jocuri, robotică, efecte speciale în filme și este inclus în software precum Godot, Blender Game Engine sau Unity 3D.

- Din sfera closed source, poate fi amintit **NVIDIA PhysX**, unul dintre cele mai populare motoare de fizică în industria jocurilor video - inclus în Unreal Engine 3+, Unity 3D și folosit de companii ca EA, THQ, 2K Games. Physx folosește accelerare hardware pe GPU, plăcile video GeForce de la NVIDIA fiind capabile să ofere o creștere exponențială a puterii de procesare a simulărilor fizice.

3.2 Soluții curente

Secțiunea următoare va pune în evidență starea actuală a dezvoltării de motoare de fizică, oferind o perspectivă asupra unor algoritmi și metode numerice care sunt folosite în prezent. Gruparea acestora urmărește componentele standard ale unui simulator de interacțiuni fizice.

3.2.1 Câteva noțiuni matematice referite și utilizate de algoritmii de mai jos

- O **funcție suport** a unei mulțimi pe o direcție este definită ca:

$$h_A : \mathbb{R}^n \mapsto \mathbb{R}, \text{ cu } A \subset \mathbb{R}^n,\ h_A(d) = \sup\{d \cdot a \mid a \in A\}$$

  Este utilizată în determinarea **punctelor suport**(cel mai îndepărtat punct al unui obiect într-o anumită direcție). Formele elementare(cub, sferă, con etc.) au funcții suport foarte ușor de calculat, mai ales în spațiul local al corpului.
- **Suma Minkowski** a două mulțimi de puncte A și B este mulțimea

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

  și este puțin interesantă în cazul de față.
- **Diferența Minkowski** a două mulțimi de puncte A și B este mulțimea

$$A \ominus B = \{a - b \mid a \in A, b \in B\}$$

  și are o proprietate remarcabilă. Mulțimile A și B se află în coliziune(au cel puțin un punct în comun) dacă $A \ominus B$ conține originea spațiului geometric. în plus, distanța dintre acestea, în cazul în care nu se intersectează este:

$$dist(A, B) = \min\{\|a - b\| \mid a \in A, b \in B\} = \min\{\|c\| \mid c \in A \ominus B\}$$

- În geometrie, un **simplex** este o generalizare a noțiunii de triunghi în spații de dimensiune arbitrară. Simplex-urile întâlnite în cadrul algoritmilor de detecție a coliziunii sunt punctul, muchia, triunghiul și tetraedrul.
- Fie un triunghi T definit de vârfurile r_1, r_2 și r_3. Atunci, pentru orice punct r din interiorul triunghiului, există o serie de numere reale λ_1, λ_2, λ_3 unice, astfel încât $\lambda_1 + \lambda_2 + \lambda_3 = 1$ și $r = \lambda_1 r_1 + \lambda_2 r_2 + \lambda_3 r_3$. Cele trei numere λ_1, λ_2, λ_3 se numesc **coordonatele baricentrice** ale punctului r în raport cu triunghiul T.
- **Tensorul de inerție** este o matrice I care exprima măsura prin care un corp se opune modificării stării sale de repaus relativ sau de mișcare de rotație uniformă la acțiunea unui moment al forței, pe fiecare dintre cele trei axe ale sistemului său de coordonate locale.

3.2.2 Detecție coliziuni

Detecția coliziunilor se realizează de obicei în două etape: una preliminară(eng. *broad phase*) și una exactă(eng. *near phase*). Motivul este unul foarte simplu – algoritmii folosiți în a doua etapă sunt semnificativ mai intensivi computațional decât cei din prima.

Etapa preliminară

Pentru prima fază, fiecare corp din simulare are atașat un **volum încadrator** sub forma unei primitive(în cazul 3D, de regulă sferă sau paralelipiped) care să îl cuprindă în întregime. Dacă 2 corpuri sunt în coliziune(se intersectează), atunci este sigur că și volumele lor încadratoare se întrepătrund. Testele de intersecție pentru primitive sunt ușor de implementat și computat. Cele mai uzuale volume încadratoare sunt:

- **Sfera încadratoare**(eng. *bounding sphere*): este definită de o poziție și o rază. Testul de intersecție este banal: două sfere se intersectează dacă distanța dintre centrele lor este mai mică sau egală cu suma razelor. Este o soluție eficientă ca memorie și ca test de intersecție, dar este inexactă și conduce la multe verificări inutile în faza fină a detecției.

- **Axis-aligned bounding box**(AABB): este definit de o poziție și de lungimea paralelipipedului pe fiecare dintre cele 3 axe ale sistemului global de coordonate(uzual, se reține jumătatea lungimii fiecărei laturi). Testul de intersecție este din nou destul de banal - se verifică întrepătrunderea celor două AABB-uri pe fiecare dintre cele 3 axe. Este mai precis decât sfera încadratoare, dar dezavantajul este că trebuie recalculat de fiecare dată când corpul este rotit, astfel încât volumul să rămână minim.
- **Oriented bounding box**(OBB): este definit de o poziție, lungimile paralelipipedului pe fiecare dintre cele 3 axe ale sistemului de coordonate local obiectului și o orientare. Este asemănător unui AABB, dar în loc de axele sistemului global de coordonate, sunt folosite axele sistemului de coordonate locale obiectului. Astfel, atât timp cât obiectul nu este deformabil, el nu trebuie recalculat și este mai precis decât un AABB. Dezavantajul este că testul de intersecție devine mai complicat și se bazează pe aplicarea **Teoremei axei separatoare**(eng. *SAT - Separating axis theorem*), descrisă mai jos, pentru 15 axe posibile de separare.

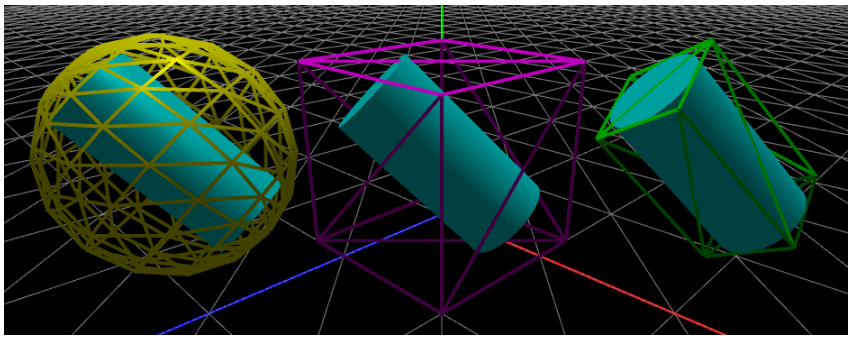


Figura 1: sferă încadratoare, AABB, OBB

Până acum am acoperit doar coliziunea în cazul unei perechi de obiecte, însă în cadrul unei simulări pot exista un număr foarte mare de obiecte. Abordarea $O(n^2)$, de a verifica fiecare pereche de obiecte din scenă este ineficientă.

O posibilă soluție este aranjarea obiectelor într-un **arbore de volume încadratoare**(eng. *bounding volume hierarchy*), care să fie actualizat pe parcursul simulării(odată la una sau mai multe etape ale acesteia). Frunzele arborelui vor fi chiar obiectele individuale ale scenei, iar restul nodurilor vor constitui volumul minim care încadrează toate nodurile copii. Astfel, dacă arborele este menținut echilibrat, timpul de verificare a perechilor de

obiecte devine logaritmic, întrucât nodurile copii nu trebuie verificate pentru coliziune dacă părinții lor nu se intersectează.

Un alt tip de arbori folosiți sunt cei care partiționează întreg spațiul scenei, pe baza unor plane alese astfel încât împărțirea obiectelor să fie cât mai uniformă. Exemple de astfel de arbori sunt **arborii BSP**(eng. *binary space-partitioning tree*) sau **octrees**(și variații ale acestora – **quadtrees**). O resursă foarte bună pentru mai multe detalii despre aceștia o reprezintă Cap. 6. *Bounding Volume Hierarchies* și Cap. 7. *Spatial Partitioning* din cartea[5] lui Christer Ericson.

Etapa exactă

A doua parte a detecției de coliziuni o consider ușor mai interesantă, deoarece este responsabilă de stabilirea **punctelor de contact** dintre corpurile aflate în coliziune. Un exemplu de caracterizare a unui astfel de punct poate fi:

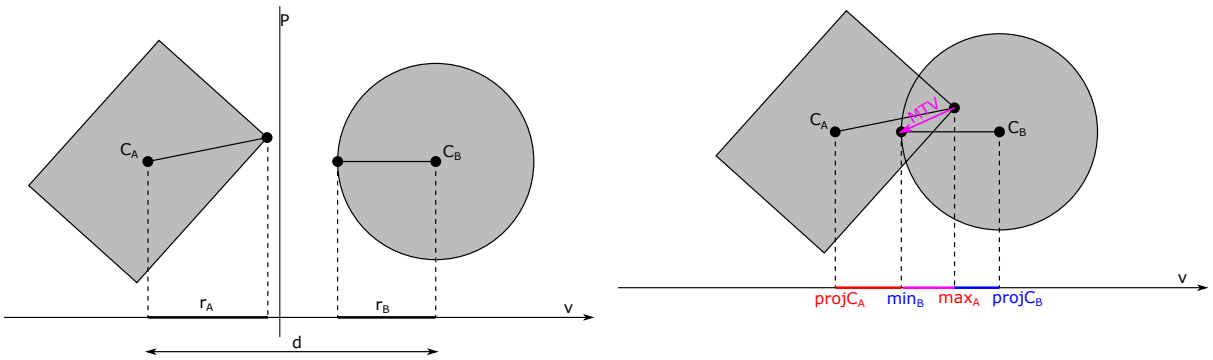
```
struct ContactPoint {
    vec3 positionA; // pozitia celui mai adanc punct de interpenetrare
    vec3 positionB; // in coordonatele locale ale fiecarui obiect
    vec3 normal; // normala contactului (directia de separare)
    float penetration; // distanta de interpenetrare
    Obj *objA; // pointer catre fiecare dintre obiecte, pentru a accesa
    Obj *objB; // matricele de modelare, functii suport etc.
}
```

În acest scop, au fost definiți mai mulți algoritmi care preiau o pereche de obiecte și stabilesc definitiv dacă acestea se intersectează, **punctul cel mai adânc de interpenetrare**, **normala**(sau direcția de separare) și **distanța de penetrare**(sau de separare) în sine.

Teorema axei separatoare derivă din **teorema hiperplanului separator**:

Teorema 1 (Teorema hiperplanului separator)**.** *Dacă A și B sunt două submulțimi disjuncte nevide ale lui $\mathbb{R}^n$, atunci există $v \in \mathbb{R}^n, v \neq 0, c \in \mathbb{R}$, astfel încât $v^T x \leq c, \forall x \in A$ și $v^T x \geq b, \forall x \in B$.*

Adaptată pentru cerințele noastre, ea ne spune că două corpuri se intersectează dacă nu există nicio axă pe care intervalele formate de proiecțiile punctelor celor două corpuri pe acea axă să se intersecteze. Astfel, două obiecte nu se intersectează(pe o axă) dacă suma razelor lor de proiecție este mai mică decât distanța dintre proiecțiile centrelor lor pe acea



(a) hiperplanul de separație P, axa separatoare v și razele de proiecție r_A și r_B ale două corpuri care nu se intersectează

(b) o axă arbitrară v și vectorul minim de translație MTV(eng. *minimum translation vector*) pentru această axă în cazul unei intersecții

Figura 2: Teorema axei separatoare

axă. În cazul poliedrelor convexe, intersecția poate fi de 3 tipuri: față-față, față-muchie, muchie-muchie (vârfurile pot fi considerate muchii degenerate) și este suficient să testăm doar următoarele posibile axe de separare:

- axele paralele cu normalele fețelor obiectului A
- axele paralele cu normalele fețelor obiectului B
- axele paralele cu vectorii rezultați în urma produsului vectorial al tuturor muchiilor lui A cu toate muchiile lui B

Normala de coliziune este chiar axa care a rezultat într-o penetrare minimă, distanța de separare este chiar lungimea vectorului minim de translație ($min_B - max_A$ sau $min_A - max_B$), iar pentru determinarea punctelor de contact, se vor calcula pentru ambele obiecte, punctele suport în direcția de separare.

Algoritmul Gilbert-Johnson-Keerthi(GJK) este o altă metodă de a determina cu precizie dacă două obiecte se intersectează. A fost propus inițial în 1988[7], ca metodă de a determina distanța euclidiană între două mulțimi convexe din $\mathbb{R}^n$, iar o implementare eficientă și robustă a fost propusă de Gino van den Bergen în 1998[13].

Algoritmul 1 Testul de intersecție Gilbert-Johnson-Keerthi

```
function GJKTestIntersection(shape_A, shape_B, init_dir)
    new_point ← Support(shape_A, init_dir) − Support(shape_B, −initial_dir)
    simplex ← {new_support_point}
    dir ← −new_point
    loop
        new_point ← Support(shape_A, dir) − Support(shape_B, −dir)
        if dot(new_point, dir) < 0 then
            return False
        end if
        simplex ← simplex ∪ new_point
        simplex, dir, contains_origin ← DoSimplex(simplex)
        if contains_origin then
            return True
        end if
    end loop
end function

function DoSimplex(simplex)
    1. determină simplex-ul cel mai apropiat de origine care se poate forma din cât mai puține din
punctele simplex-ului dat ca parametru
    2. direcția de căutare devine normala către origine a noului simplex
    3. în cazul tetraedru, întoarce True dacă simplex-ul conține originea
end function
```

Practic, la fiecare iterație, simplex-ul încearcă să se extindă și să cuprindă originea, adăugând mereu un punct suport de pe diferența Minkowski a celor două obiecte, aflat în direcția originii, până când aceasta este cuprinsă în simplex sau nu se mai apropie de acesta.

În cazul în care algoritmul GJK a stabilit că există o coliziune, pentru determinarea normalei, distanței de penetrare și a punctelor de contact, **algoritmul EPA**[14](eng. *expanding polytope algorithm*) este continuarea firească. Acesta preia simplex-ul rezultat în urma aplicării GJK și îl extinde iterativ cu puncte suport de pe frontiera diferenței Minkowski, până când distanța minimă dintre politopul rezultat și origine nu se mai modifică. Odată întâlnită această situație, coordonatele baricentrice ale proiecției originii pe triunghiul(în cazul 3D) sau dreapta(în cazul 2D) cele mai apropiate de origine pot fi folosite pentru determinarea punctelor de contact, iar distanța de la origine la triunghi(sau dreaptă) este chiar distanța de penetrare și normala contactului. Expansiunea politopului se face prin divizarea feței celei mai apropiate și crearea de noi triunghiuri sau laturi folosind punctul nou ales și punctele rămase.

Algoritmul 2 Expanding Polytope Algorithm și determinarea punctelor de contact

function EPACreateContact(*shape_A*, *shape_B*, *simplex*)
  polytope ← *simplex.triangles*
  loop
    closest_triangle ← argmin_t rDISTANCE$(tr, origin), tr \in polytope$
    distance ← DISTANCE(*closest_triangle*, *origin*)
    normal ← *closest_triangle.normal*
    new_point ← Support(*shape_A*, *normal*) − Support(*shape_B*, −*normal*)
    new_distance ← DISTANCE(*new_point*, *origin*)
    if $new_distance - distance < threshold$ **then**
      coords ← BARYCENTRIC(*origin*, *closest_triangle*)
      contact_points ← pentru fiecare obiect, se calculează punctul de contact în funcție de corespondențele fiecărui punct al triunghiului din mulțimea de puncte a obiectului respectiv
      return *contact_points*, *normal*, *distance*
    end if
    polytope ← *polytope* \ {*closest_triangle*}
    creează triunghiuri noi folosind *new_point* în spațiul lăsat descoperit
  end loop
end function

3.2.3 Rezolvare coliziuni

În esență, rezolvarea coliziunilor implică aplicarea unui **răspuns** asupra corpurilor aflate în contact, care să conducă la separarea acestora. În lumea reală, răspunsul vine sub forma forțelor elastice care se opun comprimării(oricât de mică ar fi aceasta) corpurilor aflate în contact, care determină o accelerație care reduce viteza de ciocnire până la valori negative, când corpurile se separă. În cadrul unui motor de fizică, acest fenomen este simulat cu ajutorul **impulsurilor** – modificări bruște a vitezelor obiectelor aflate în coliziune, astfel încât acestea să tindă spre separare. În general, sunt suficiente două impulsuri – cel liniar și cel unghiular, fiecare modificând viteza corespondentă.

Soluționarea unei coliziuni este de regulă realizată cu ajutorul conceptului mult mai general de **constrângeri fizice**, definite ca o serie de ecuații și inecuații care trebuie să fie satisfăcute. În cazul rezolvării unei coliziuni, constrângerea care trebuie satisfăcută de vitezele celor două corpuri este:

$$\dot{C}: \left(-\vec{V_A} - \vec{\omega_A} \times \vec{r_A} + \vec{V_B} + \vec{\omega_B} \times \vec{r_B}\right) \cdot \vec{n} + b \geq 0$$

unde:

- $\vec{V_A}, \vec{V_B}$ sunt vitezele liniare ale celor două corpuri
- $\vec{\omega_A}, \vec{\omega_B}$ sunt vitezele unghiulare ale celor două corpuri
- $\vec{r_A}, \vec{r_B}$ sunt definite ca $P_A - C_A$ și $P_B - C_B$, cu P_A, P_B punctele cele mai adânci de interpenetrare și C_A, C_B centrele de masă ale corpurilor

- $\vec{n}$ este normala contactului
- b este un termen de bias, care corespunde vitezei de separare a celor două corpuri și este influențat de coeficientul de restituire al ciocnirii:

$$b = C_R \left(-\vec{V_A} - \vec{\omega_A} \times \vec{r_A} + \vec{V_B} + \vec{\omega_B} \times \vec{r_B} \right) \cdot \vec{n}$$

Frecările sunt rezolvate sub forma unor **impulsuri tangențiale**, care vor modifica viteza corpurilor în două direcții perpendiculare pe normala de contact, în plus față de **impulsul normal**.

La calculul impulsurilor se ține cont și de masele celor două corpuri și de tensorii de inerție. O derivare a formulei poate fi urmărită în prezentarea lui Erin Catto[1].

O simulare va conține un număr mare de contacte, care se pot afecta unele pe altele(de ex. în cazul unei stive de obiecte), motiv pentru care rezolvarea acestora se face iterativ, până la convergență. Astfel, dacă rezolvarea unui contact va afecta un altul (este modificată distanța de penetrare, normala sau viteza de întâlnire), acest lucru se va reflecta în iterațiile succesive și sistemul poate găsi soluția corectă. În realitate, într-o simulare, acest lucru se întâmplă destul de rar, dar rezultatele obținute sunt satisfăcătoare, imperfecțiunile fiind neglijabile.

Realistic vorbind, o coliziune este deseori formată din mai multe puncte de contact, care împreună formează un **manifold** care trebuie rezolvat.

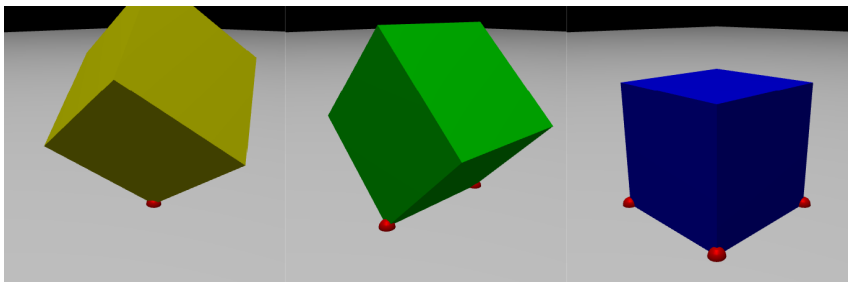


Figura 3: 3 cazuri de contact și manifold-urile lor

În acest scop, există 2 posibile abordări [4, Secțiunea 6.6.2 *Collision response for colliding contact*]:

1. **Metoda impulsurilor secvențiale** – impulsurile sunt calculate și aplicate iterativ în fiecare dintre punctele de contact.

2. **Metoda contactelor simultane** – Jacobian-ul folosit pentru calculul impulsurilor este determinat pe baza tuturor punctelor din manifold-ul de contact, iar impulsurile sunt aplicate o singură dată per manifold.

3.2.4 Integrare numerică

Într-un motor de fizică, etapa de integrare constă în actualizarea stării obiectelor, prin **integrarea în raport cu timpul** a mărimilor fizice derivate. Ne amintim de la orele de fizică din liceu că accelerația unui corp este dată de formula $a = \frac{F}{m}$, unde F este rezultanta forțelor care acționează asupra corpului și m este masa acestuia. Într-o simulare, accelerația este considerată o mărime fizică primară și este cea dintâi calculată în fiecare cadru. Accelerația este totodată definită și ca variația vitezei în timp, iar viteza este variația poziției în timp, ceea ce ne permite să calculăm atât viteza $\dot{p}'$, cât și poziția p', prin integrarea accelerației $\ddot{p}$, respectiv vitezei $\dot{p}$:

$$\dot{p}' = \dot{p} + \ddot{p}t$$

$$p' = p + \dot{p}t + \ddot{p}\frac{t^2}{2} \approx p + \dot{p}t$$

Deoarece într-o simulare avem de-a face cu momente discrete de timp(dictate de diferența de timp dintre două cadre – **deltaTime**, aceste valori trebuie aproximate, de unde rezultă nevoia folosirii unor metode de integrare numerică cu pas de timp discret. Se disting o serie de metode, mai mult sau mai puțin precise[6]:

- **Metoda Euler explicită** presupune determinarea, în ordine, mai întâi a noii poziții și apoi a noii viteze, dar are dezavantajul că pierde din precizie dacă există variații mari ale mărimilor de la un cadru la altul, cu eroare de ordinul $O(deltaTime)$

```
position = position + velocity * deltaTime;
velocity = velocity + acceleration * deltaTime;
```

- **Metoda Euler semi-implicită** presupune folosirea noii viteze la determinarea noii poziții și este considerabil mai precisă, cu eroare de ordinul $O(deltaTime^2)$

```
velocity = velocity + acceleration * deltaTime;
position = position + velocity * deltaTime;
```

- **Metoda Runge-Kutta 4** evaluează derivata stării în 4 puncte diferite din cadrul intervalului de derivare, folosind ca feedback rezultatele anterioare și este mult mai precisă, ceea ce conduce la o eroare de ordinul $O(deltaTime^4)$

3.2.5 Interfață grafică

TODO:

3.3 Alegeri pentru lucrarea de față

Dintre soluțiile menționate în subsecțiunile de mai sus, am fost nevoit să fac niște alegeri, pe care urmează să le prezint și să le motivez în continuare.

3.3.1 Detecție coliziuni

Deoarece scenele cu care am testat simulatorul sunt de dimensiuni relativ mici(sub 100 de obiecte), am decis că o ierarhie de volume încadratoare nu aduce o îmbunătățire semnificativă în raport cu overhead-ul pe care l-ar aduce etapei de implementare a soluției. La fel, o partiționare a spațiului scenei în arbori, folosind BSP mi s-a considerat nejustificată. Astfel, m-am utilizat doar de un **detector** $O(n^2)$, care testează intersecția dintre **OBB-urile** obiectelor folosind **Teorema axei separatoare**.

Pentru că obiectele din scenă sunt doar corpuri geometrice elementare pentru care nu stochez liste de vârfuri, muchii și fețe, ci doar descrierea geometrică a formei acestora, Teorema axei separatoare nu se pretează pentru determinarea punctelor de contact. În plus, corpurile rotunde(cilindru, con, sferă, capsulă) ar pune probleme în realizarea testului de separare, deoarece ar avea nevoie de un număr foarte mare de axe care să fie testate. În schimb, am ales să folosesc **algoritmul GJK**, care se poate folosi de descrierea geometrică a formei corpurilor pentru calculul facil al punctelor de suport necesare la determinarea simplex-ului final.

Și, în mod evident, algoritmul GJK se potrivește de minune cu **EPA**, având în comun o bună parte dintre metode și structuri de date.

3.3.2 Rezolvare coliziuni

Complexitatea pe care ar fi adus-o implementarea metodei rezolvării simultane a contactelor nu este justificată în cazul de față. Ar fi fost nevoie de un redesign al structurilor de date folosite adus de necesitatea operațiilor dintre matrice și vectori de dimensiuni mari, direct proporționale cu numărul de contacte din manifold – ar fi trebuit să îmi scriu propria implementare pentru structurile geometrice de date - vectori și matrice de dimensiuni mai mari decât 4x4. Am ales, astfel, să utilizez **metoda impulsurilor secvențiale**, care produce rezultate acceptabile pentru o simulare care nu se vrea a fi hiper-exactă.

De asemenea, am decis să nu generalizez contactele la constrângeri fizice și am ales să introduc în schimb alte optimizări, descrise în capitolele ulterioare.

3.3.3 Integrare numerică

În cazul unei simulări, metoda Euler explicită este inferioară din toate punctele de vedere celei **Euler implicită**, iar precizia altor metode de ordin superior ar fi combătute oricum de micile imperfecțiuni apărute în urma rezolvării coliziunilor. Am ales să păstrez lucrurile simple și să folosesc a doua metodă prezentată mai sus.

4 SOLUȚIA PROPUSĂ

Soluția propusă urmărește arhitectura firească a unei aplicații grafice **OpenGL**, cu următoarele componente:

1. **backend**-ul – este un framework care oferă funcționalitățile de bază necesare funcționării aplicației OpenGL
2. **logica** – este implementarea propriu-zisă a simulatorului de interacțiuni fizice
3. **frontend**-ul – reprezintă interfața cu utilizatorul

4.1 Backend-ul

Funcționalitățile pe care le oferă așa-numitul backend sunt:

- crearea, controlul și interfațarea cu un context OpenGL
- suport pentru încărcarea de meshe 3D
- suport pentru definirea și încărcarea de shadere OpenGL
- crearea și controlul unei ferestre de afișare
- oferirea unui model generic pentru scrierea de aplicații OpenGL:
  - control pentru fereastra de afișare
  - management-ul input-ului de la mouse și tastatură
  - implementarea unei camere first-person pentru vizualizarea scenei
  - o scenă de bază care poate fi moștenită și căreia i se vor adăuga funcționalitățile specifice aplicației
  - interfață pentru desenarea meshelolor 3D

Lista de mai sus nu este exhaustivă, complexitatea unui astfel de framework, chiar și minimal, este destul de mare. Cum implementarea acestuia nu a făcut parte din obiectivele proiectului, am ales să folosesc framework-ul realizat de Gabriel Ivănică pentru laboratoarele de EGC[8]. O descriere mai detaliată a acestuia poate fi găsită aici[9].

Astfel, etapele funcționării unei aplicații, facilitate de framework-ul descris mai sus sunt:

1. Se definesc proprietățile pentru fereastra de lucru.
2. Se inițializează API-ul OpenGL.
3. Se creează fereastra de lucru cu un context OpenGL.
4. Se atașează evenimentele de fereastră la motorul care se ocupă de tratarea lor.
5. Se creează și se inițializează noua scenă 3D.
6. Se pornește bucla principală a aplicației.

4.2 Logica aplicației

Este componenta centrală a oricărei aplicații grafice și funcționează în întregime în cadrul buclei principale de program.

4.2.1 Funcționare

Într-o iterație au loc următorii pași:

0. Backend-ul realizează un preambul în care își actualizează parametrii interni
    - Se pregătește contextul OpenGL pentru desenare – sunt curățate buffer-ele de culoare, dimensiunea ferestrei de desenare este reactualizată, dacă este cazul
    - Se estimează timpul de execuție pentru iterația actuală, pe baza duratei iterației precedente(deltaTime).
    - Sunt procesate evenimentele salvate anterior.
1. Este desenată scena în totalitate.
2. Se actualizează starea obiectelor din simulare, prin efectuarea pasului de integrare.
3. Se efectuează faza preliminară a detecției de coliziuni – se obține o listă de perechi de obiecte posibil aflate în contact.
4. Se efectuează faza exactă a detecției de coliziuni – lista de mai sus este prelucrată și sunt actualizate vechile contacte, în cazul perechilor de obiecte aflate în coliziune continuă, sau sunt generate altele noi.
5. Se încearcă, iterativ, rezolvarea tuturor contactelor din scenă.

6. Se face trecerea la următorul cadru – se revine la pasul 0.

- Pentru a nu bloca execuția simulării, toate evenimentele de input sau de control al ferestrei sunt salvate de backend într-un buffer, pentru a putea fi soluționate toate deodată în pasul 0.

4.2.2 Structură

În continuare voi descrie, pe rând, componentele individuale ale simulatorului.

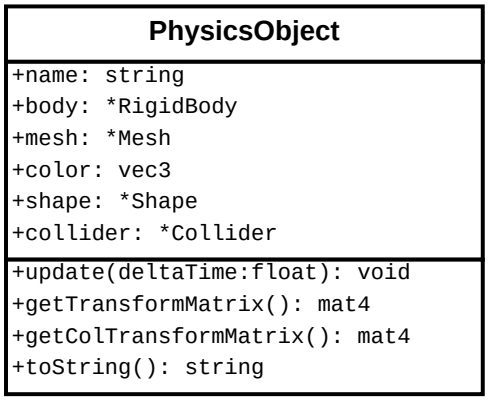


Figura 4: Descrierea unui obiect al scenei

Începând cu **obiectele scenei**, acestea sunt alocate dinamic la rularea aplicației și stochează informații necesare la desenare, dar oferă și o interfață pentru actualizarea corpului solid, prin metoda **update()**. Am ales să folosesc obiectul scenei ca pe o scurtătură către toate referințele necesare celorlalte componente(de ex. în cadrul algoritmului GJK, determinarea punctului de suport pe diferența Minkowski a 2 corpuri necesită apelul metodei *getSupportPtInLocalSpace()* din clasa Shape și poate fi accesată prin referința din PhysicsObject).

Starea cinematică a obiectului este încapsulată într-un obiect **RigidBody**(descris în întregime aici 9). Pasul de integrare presupune:

1. **determinarea accelerației** ca rezultat al forțelor care acționează asupra obiectului
2. determinarea noilor valori pentru **viteză**(atât liniară, cât și unghiulară), și pentru **poziție**, respectiv **orientare**, folosind metoda Euler semi-implicită

3. sunt simulate forțele de frecare cu aerul, prin înmulțirea vitezelor cu un **factor de amortizare**
4. sunt actualizate **matricele de modelare** și **tensorul de inerție** și se resetează acumulatorii de forțe și momente

Mai departe, controlul aplicației este preluat de **detectorul de coliziuni**, a cărui structură completă poate fi găsită aici 10.

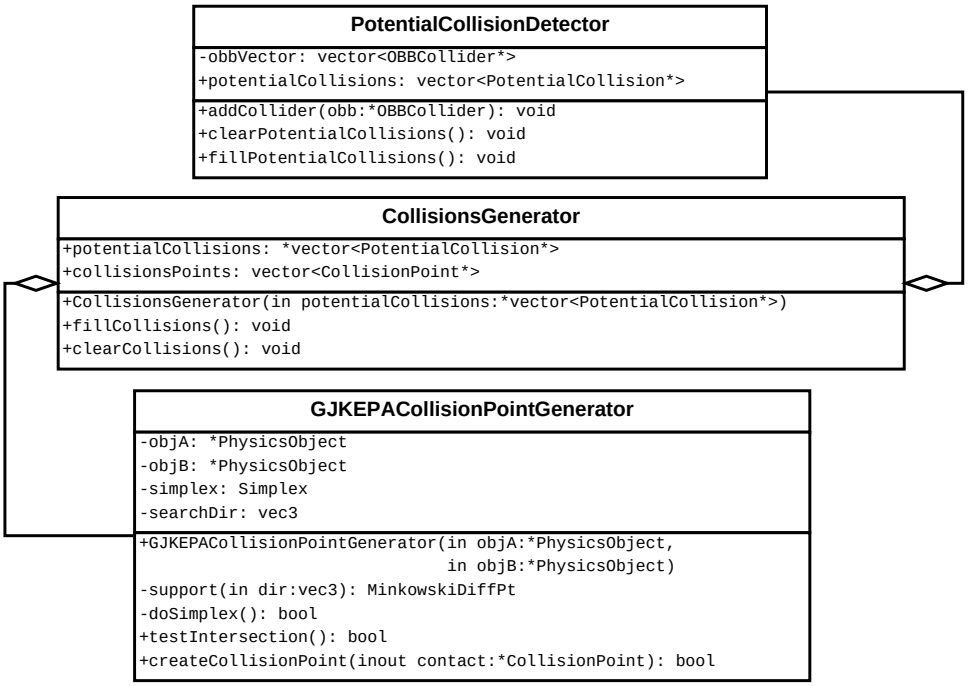


Figura 5: Sistemul de detecție de coliziuni

Am considerat necesară separarea procesului de detecție a coliziunilor în două subsisteme – **detectorul de posibile coliziuni**, corespondent fazei preliminare a algoritmului și **generatorul de puncte de coliziune**, care preia responsabilitatea pentru corpurile descoperite de primul. **generatorul de coliziuni** nu este decât un agregator al celor două subsisteme care oferă o interfață simplă aplicației.

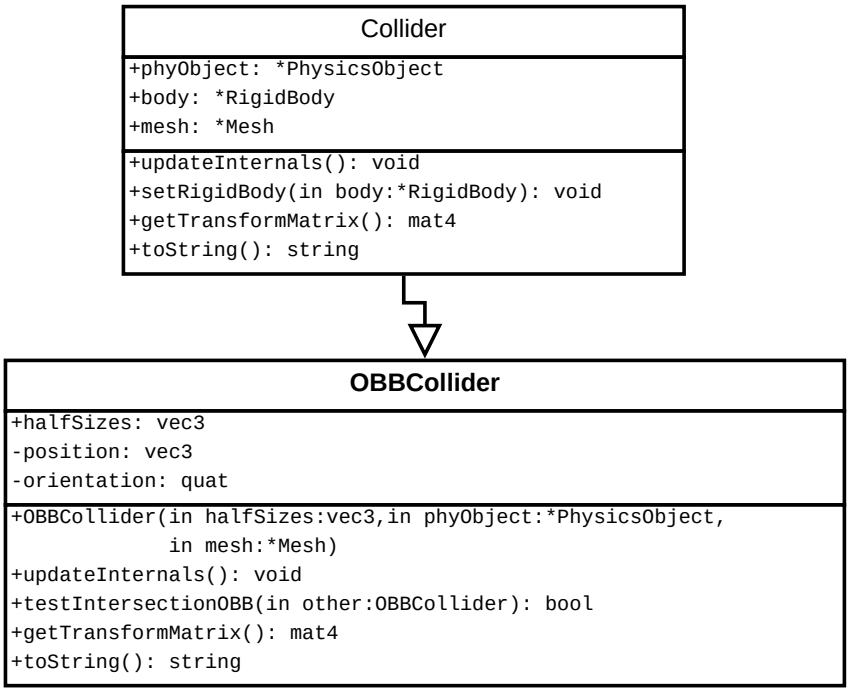


Figura 6: OBBCollider implementează clasa abstractă Collider

Detectorul de posibile coliziuni menține un **vector de collider-e OBB** pentru obiectele din scenă pentru care se dorește posibilă existența coliziunilor și construiește **vectorul de posibile coliziuni** folosind referințele către obiectele fizice din collider-e. Generatorul de puncte de coliziune preia rezultatul și efectuează, mai întâi testul de intersecție, și mai apoi calculează punctele de coliziune, care este rezultatul final al etapei de detecție a coliziunilor și care este transmis mai departe în pipeline **rezolvitorului de coliziuni**.

Deoarece **punctele de coliziune** reprezintă doar niște informații legate de coliziunea cea mai adâncă dintre 2 corpuri(câte un singur punct pe suprafețele celor două corpuri, normala și distanța de penetrare) și ar fi ineficient să se aloce dinamic noi contacte la fiecare cadru, am folosit o schemă de caching bazată pe ideea de **contacte persistente** și **manifold-uri de contact**:

- introduc aici noțiunea de **punct de contact**, care pe lângă informațiile de coliziune, mai include și alte câmpuri necesare algoritmului de caching sau rezolvării în sine

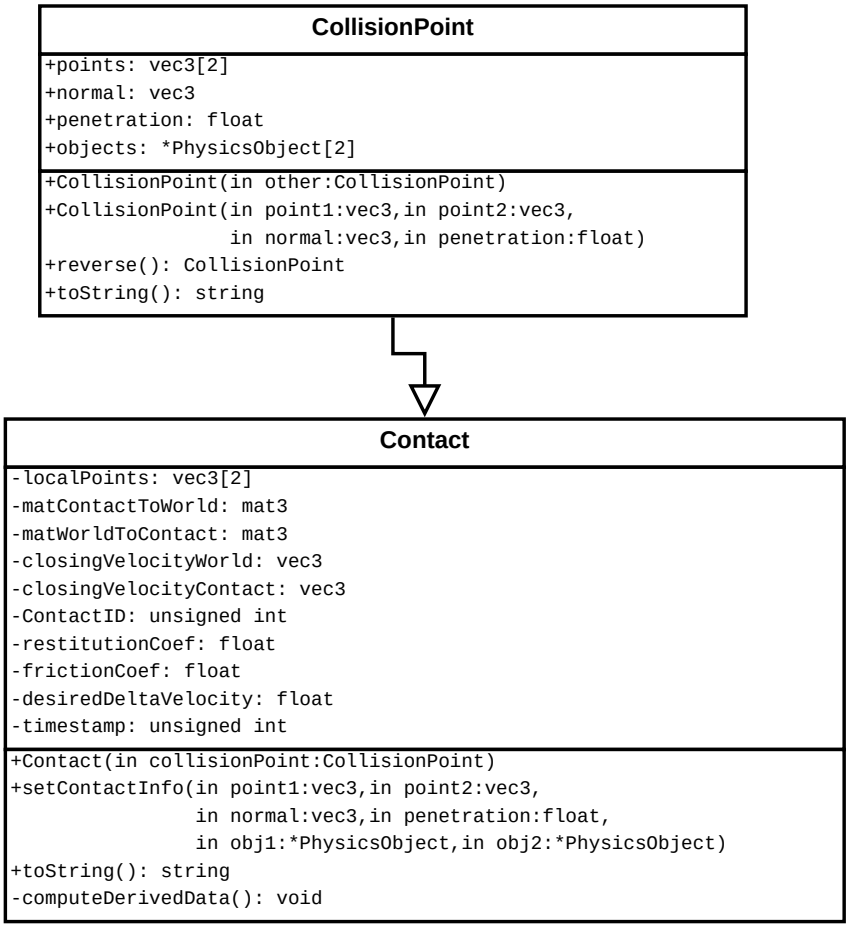


Figura 7: diferența subtilă dintre punctul de coliziune și punctul de contact (a se observa că Contact moștenește CollisionPoint)

- pentru fiecare pereche de obiecte pentru care există coliziune, punctele de contact sunt reținute într-un manifold format din maxim 4 astfel de puncte
- de fiecare dată când obțin un nou punct de coliziune între două obiecte, caut manifold-ul de contact asociat perechii de obiecte și îl actualizez:
  - dacă nu am găsit un manifold pentru perechea curentă, atunci creez unul nou și creez un nou punct de contact pe care îl adaug la manifold
  - dacă punctul de coliziune poate fi atribuit unuia dintre punctele de contact ale manifold-ului(se află la o distanță suficient de mică), atunci doar actualizez punctul de contact deja existent
  - dacă nu corespunde unui astfel de punct, atunci este creat un nou punct de contact și este adăugat manifold-ului, înlocuind, dacă există deja 4 puncte, punctul de contact care oferă cea mai puțină informație(cel cu penetrarea cea mai mică)

- manifold-urile care nu au fost actualizate în cadrul curent, sau cele formate din puncte de contact cu penetrare negativă, sunt pur și simplu eliminate din sistem

Cu asta se încheie pipeline-ul logic al simulatorului. Totul se reia de la început în noul cadru.

4.3 Interfața cu utilizatorul

TODO:

5 DETALII DE IMPLEMENTARE

Acest capitol conține câteva observații asupra procesului de dezvoltare și implementare a aplicației aferente proiectului.

Implementarea a fost realizată folosind limbajul **C++** și biblioteca **OpenGL**, în mediul de programare Microsoft **Visual Studio** Community Edition 2017, iar testarea s-a făcut pe laptop-ul personal(Intel i5 2.5GHz, 8GB RAM, Intel HD Graphics 4600).

Dezvoltarea a pornit de la framework-ul utilizat la laboratoarele de EGC, care a fost descris în capitolul precedent. Acesta oferă clasa abstractă SimpleScene, care poate fi moștenită și care oferă interfețe pentru rularea aplicației(metodele Init(), FrameStart(), Update(), FrameEnd()), randarea de meshe și pentru tratarea de input de la mouse și tastatură. Aplicațiile demo sunt pur și simplu implementări ale acestei SimpleScene. În plus, framework-ul oferă și implementarea unei camere first-person, care poate fi controlată din mouse și tastatură, pentru vizualizarea scenei.

Biblioteca folosită pentru tipuri și funcții matematice și geometrice este **GLM**

Primul obiectiv atins a fost **implementarea clasei RigidBody**9 și, implicit, a operației de integrare. Testarea s-a făcut cu un simplu cub asupra căruia acționează gravitația și asupra căruia se pot aplica forțe, momente și impulsuri cu ajutorul tastaturii. Singura mențiune specială este că am ales să stochez orientarea corpului într-un **quaternion**, în loc de o reprezentare matriceală sau cu unghiurile lui Euler. Motivul este foarte simplu, ocupă mai puțin spațiu în memorie, nu prezintă problema Gimbal Lock[10], iar suportul GLM pentru operații cu quaternioni este foarte bun. Exemplu de utilizare:

```
glm::quat orientation;
...
orientation = orientation + deltaTime * 0.5f * glm::quat(0.0f,
    angVelocity) * orientation;
orientation = glm::normalize(orientation);
```

A doua etapă de dezvoltare a constat în implementarea sistemului de **detecție a coliziunilor**. Complexitatea etapei mi-a pus mult mai multe bătăi de cap.

Pentru **faza preliminară**, am implementat direct collider-ele de tip OBB, cu un detector $O(n^2)$. După ce am citit despre teorema axei separatoare și eșuarea în a corecta erorile din implementarea mea a testului de intersecție, am ales să folosesc cea mai eficientă implementare posibilă, cea propusă de Ericson în [5, Secțiunea 4.4 *Oriented Bounding Boxes(OBBs)*]. Tot aici, am fost nevoit să implementez abstractizarea obiectelor din scenă, care să încapsuleze obiectele RigidBody și Collider.

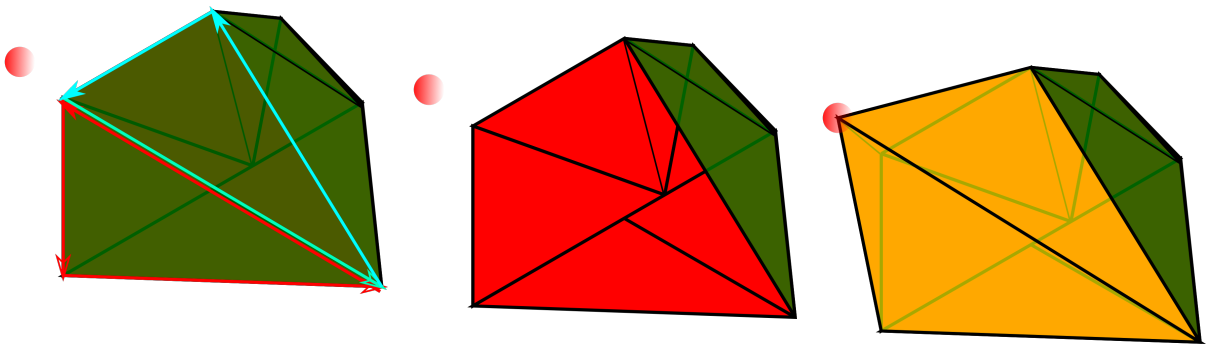
Faza de detecție exactă și generare a punctelor de coliziune este cea care mi s-a părut cea mai dificilă. A necesitat o foarte mare atenție la detalii cum ar fi ordinea vârfurilor care compun simplex-ul din algoritmul GJK sau triunghiurile care compun poliedrul din cadrul EPA. Pentru implementarea GJK, foarte utilă mi s-a părut explicația lui Casey Muratori[12], care a oferit o interpretare mult mai ușor de înțeles a algoritmului decât enunțarea lui inițială[7]. Remarcabil mi se pare că în cazul unui simplex tetraedru, am obținut până la 4 if-uri imbricate, pentru determinarea precisă a noului simplex și a noii direcții. Includ aici cazul triunghi, deoarece mi se pare că algoritmul în pseudocod nu spune prea multe cititorilor nefamiliarizați cu acesta.

```
void GJKEPA::GJKEPACollisionPointGenerator::doSimplex3() {
    /* simplex is a triangle ABC, A was just added */
    glm::vec3 vecAO = -simplex.a.v;
    glm::vec3 vecAB = simplex.b.v - simplex.a.v;
    glm::vec3 vecAC = simplex.c.v - simplex.a.v;
    glm::vec3 vecABC = glm::cross(vecAB, vecAC);

    if (glm::dot(glm::cross(vecABC, vecAC), vecAO) > 0) {
        if (glm::dot(vecAC, vecAO) > 0) {
            simplex.set(simplex.a, simplex.c);
            searchDir = glm::cross(glm::cross(vecAC, vecAO), vecAC);
        } else {
            if (glm::dot(vecAB, vecAO) > 0) {
                simplex.set(simplex.a, simplex.b);
                searchDir = glm::cross(glm::cross(vecAB, vecAO), vecAB);
            } else {
                simplex.set(simplex.a);
                searchDir = vecAO; }}
    } else {
        if (glm::dot(glm::cross(vecAB, vecABC), vecAO) > 0) {
            if (glm::dot(vecAB, vecAO) > 0) {
                simplex.set(simplex.a, simplex.b);
                searchDir = glm::cross(glm::cross(vecAB, vecAO), vecAB);
            } else {
                simplex.set(simplex.a);
                searchDir = vecAO; }
        } else {
            if (glm::dot(vecABC, vecAO) > 0) {
                searchDir = vecABC;
            } else {
                simplex.set(simplex.a, simplex.c, simplex.b);
                searchDir = -vecABC;
    }}}
}
```

Un alt hop a fost reprezentat de **implementarea EPA**. Problemele întâlnite au fost:

- felul în care ar trebui stocat poliedrul – am ales să îl reținca o listă de triunghiuri
- pasul de expansiune a poliedrului – adăugarea unui nou vârf presupune eliminarea triunghiurilor care ar fi acoperite de acesta
  - parcurg muchiile triunghiurilor care ar fi acoperite în sens trigonometric
  - construiesc o listă cu aceste muchii
  - în momentul în care am ajuns la o muchie deja existentă în listă, o elimin cu totul, întrucât aceasta ar fi între două triunghiuri care trebuie eliminate
  - la final, în lista de muchii, rămân doar muchiile de frontieră
  - cu fiecare din aceste muchii și noul punct, formez noi triunghiuri care vor fi adăugate la poliedru



(a) punctul roșu este noul punct, iar săgețile colorate determină sensul de parcurgere a triunghiurilor

(b) poliedrul după eliminarea celor două triunghiuri care ar fi acoperite de noul punct

(c) poliedrul după construirea noilor triunghiuri

Figura 8: Un pas de extindere al poliedrului în cadrul EPA

Bucata de cod care face exact acest lucru este:

```
/* adds an edge to the list of edges or removes it, if it is already there,
    but reversed */
static void addRemoveEdge(std::list<GJKEPA::Edge> &edges, const GJKEPA::
    MinkowskiDiffPt &a, const GJKEPA::MinkowskiDiffPt &b) {
    for (auto it = edges.begin(); it != edges.end(); it++) {
        if (it->a.v == b.v && it->b.v == a.v) {
            /* found reversed edge, just remove it */
            it = edges.erase(it);
            return; }}
    edges.push_back(GJKEPA::Edge(a, b));
}
```

```
[...]
for (auto it = triangles.begin(); it != triangles.end();) {
    if (glm::dot(it->vecABC, nextSup.v - it->a.v) > -EPA_EPSILON) {
        /* update the edge list in order to remove the triangles facing
           this point */
        addRemoveEdge(edges, it->a, it->c);
        addRemoveEdge(edges, it->c, it->b);
        addRemoveEdge(edges, it->b, it->a);
        it = triangles.erase(it);
        continue; }
    it++; }
/* re-create the triangles from the remaining edges */
for (Edge e : edges) {
    triangles.push_back(Triangle(nextSup, e.a, e.b));
}
```

Rezolvarea contactelor este componenta cea mai complexă a simulatorului din punct de vedere matematic și programatic. Procesul constă în doi pași:

- rezolvarea interpenetrării
- determinarea vitezei corecte de separare și aplicarea ei

Rezolvarea interpenetrării presupune modificarea poziției corpurilor astfel încât acestea să nu se mai intersecteze. Pentru a obține rezultate mai realiste, se aplică atât o mișcare liniară, cât și una unghiulară. Calculul acestora presupune determinarea inerției liniare și unghiulare pentru ambele corpuri, împărțirea acestora la inerția totală necesară mișcării unitare și înmulțirea cu distanța de penetrare, pentru a obține cantitățile de mișcare necesare pentru separare.

Calculul impulsului care să stabilească viteza corectă de separare se bazează din nou, pe componentele individuale – cea liniară și cea unghiulară. Se determină schimbarea în viteze cauzată de o unitate de impuls. Se determină diferența în viteză dorită $desiredVelocity = -closingVelocity(1 + coefRestitution)$. Apoi, folosind cele două rezultate, sunt calculate componentele impulsului care vor fi aplicate celor două corpuri.

Deoarece rezolvarea pe rând a contactelor dintr-un manifold poate duce la situații în care un alt contact decât cel curent să fie accentuat, folosesc mai multe iterații pentru aplicarea celor doi pași de mai sus, de fiecare dată încercând să rezolv cel mai adânc, respectiv cel mai rapid contact din manifold, astfel încât soluția să poată converge. Odată rezolvat un contact, schimbările aduse de acesta sunt aplicate tuturor celorlalte contacte din manifold.

Principala resursă folosită pentru înțelegerea acestui proces a fost cartea[11] lui Ian Mil-

lington, iar soluția mea urmează îndeaproape indicațiile și implementarea acestuia.

TODO: ceva și despre interfața grafică

6 EVALUARE

TODO:

7 CONCLUZII

TODO:

BIBLIOGRAFIE

BIBLIOGRAFIE

[1] Erin Catto. Modeling and solving constraints. In *Game Developers Conference*, page 81, 2009.

[2] Erin Catto. Box2D. `https://github.com/erincatto/Box2D`, 2014.

[3] Erwin Coumans. Bullet Physics. `https://github.com/bulletphysics/bullet3`, 2017.

[4] Dave H. Eberly. *Game Physics*. Elsevier Science Inc., New York, NY, USA, 2003.

[5] Christer Ericson. *Real-Time Collision Detection*. CRC Press, San Francisco, CA, 2004.

[6] Glenn Fiedler. Integration basics - how to integrate the equations of motion, 2004. ultima accesare: 18 August 2018.

[7] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, April 1988.

[8] Gabriel Ivănică. Framework-EGC. `https://github.com/UPB-Graphics/Framework-EGC`, 2016.

[9] Gabriel Ivănică. Laboratorul 01, 2016. ultima accesare: 18 August 2018.

[10] Valentin Koch. Rotations in 3d graphics and the gimbal lock, 2016. ultima accesare: 18 August 2018.

[11] Ian Millington. *Game Physics Engine Development*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[12] Casey Muratori. Implementing GJK - 2006. ultima accesare: 18 August 2018.

[13] Gino Van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, March 1999.

[14] Gino Van den Bergen. Proximity queries and penetration depth computation on 3d game objects. 2001.

ANEXE

A EXTRASE DE COD

...

B DIAGRAME DE CLASE

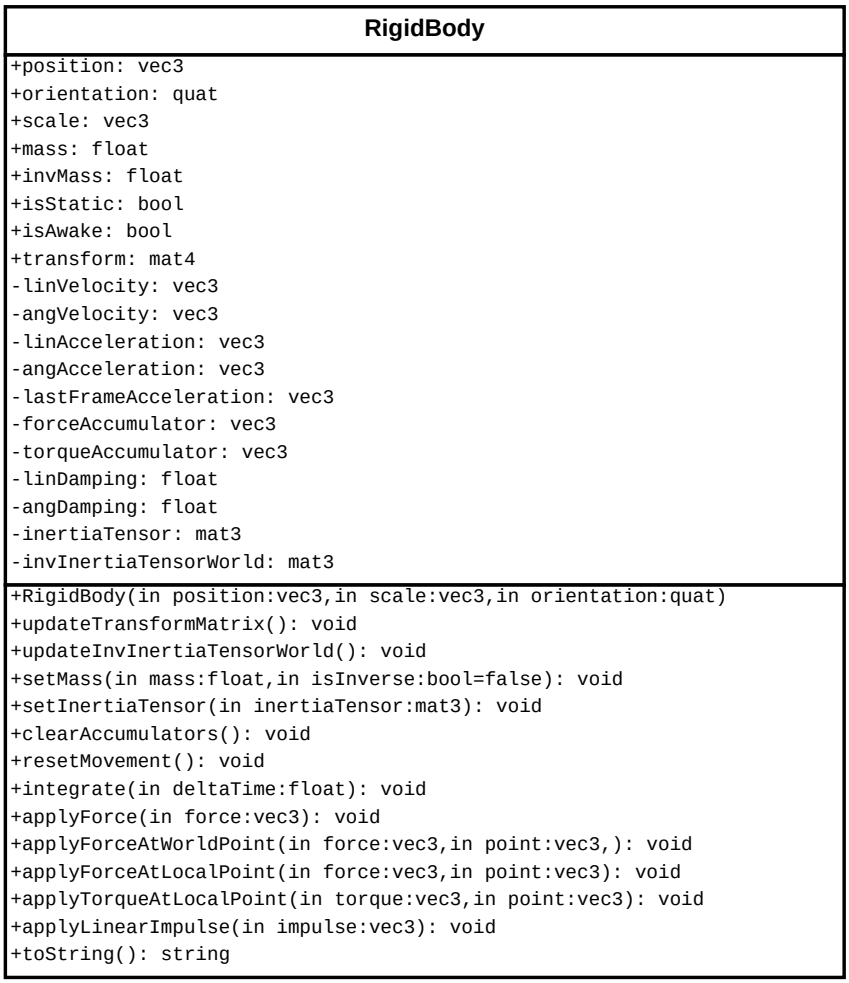


Figura 9: Definiția unui RigidBody

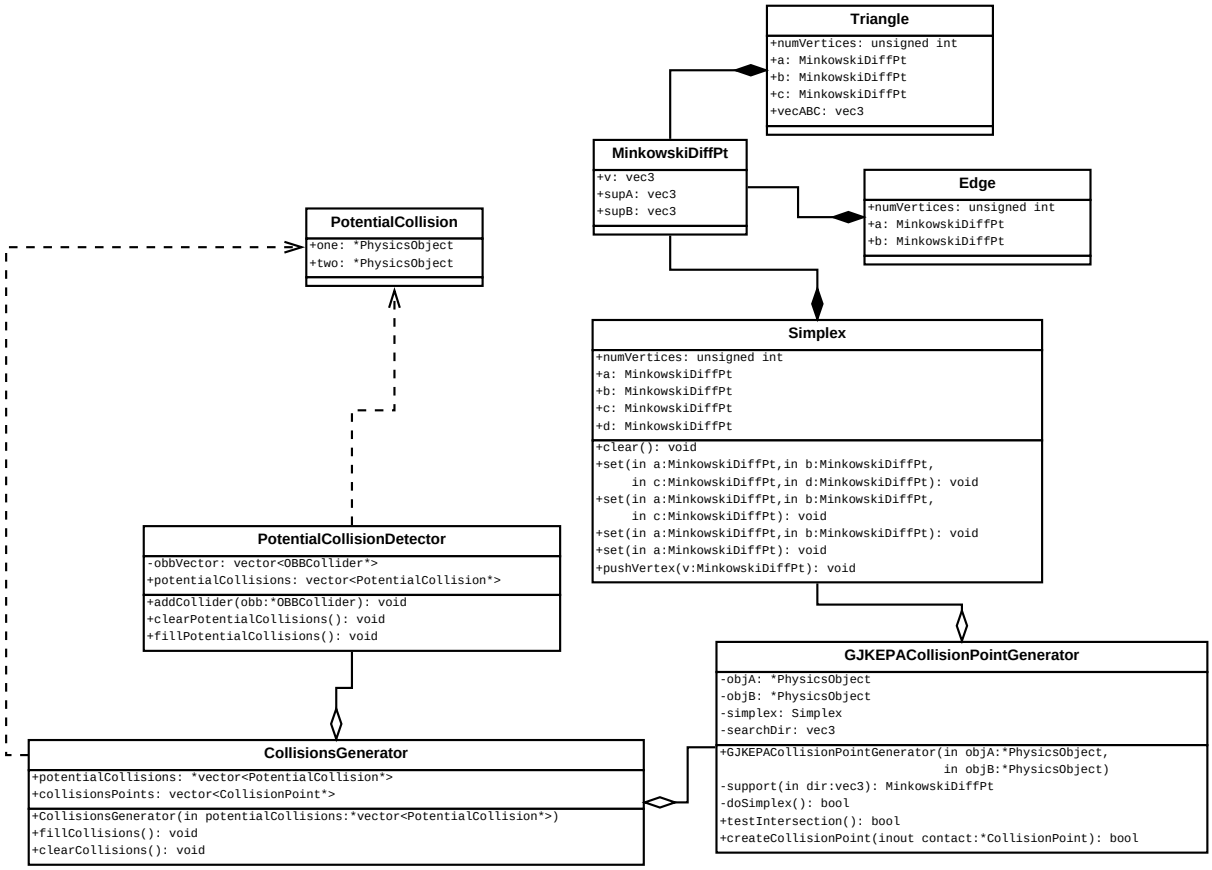


Figura 10: Subsistemul de detecție de coliziuni