UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Simularea interacțiunilor fizice

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

Simulation of physical interactions

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 de 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* application 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).

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 bune 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 menționate 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*[1], *bullet*[2]). 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

În urma realizării acestei lucrări, am dobândit cunoștințe semnificative în domeniul simulării interacțiunilor mecanice și al implementării de motoare de fizică robuste și efi-

ciente.

În plus, am obținut o aplicație grafică *OpenGL*, capabilă să ruleze o simulare de dimensiune rezonabilă (max. 64 de obiecte într-o scenă) a interacțiunilor mecanice dintre corpuri 3D, cuplată cu o interfață grafică pentru utilizator, prin care acesta poate încărca sau salva scene și controla parametrii simulării sau direct obiectele din scenă.

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 unul personal, acela 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 să se obțină 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 ales 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 (ex. *bullet*[2]) sau, de ce nu, să efectuez muncă în cercetare sau în industrie în acest domeniu, la un nivel ceva mai complex și actual.

3 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*[3], *Source*[4], *Unity*[5] 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**[1] 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 și folosește un arbore dinamic pentru faza preliminară a detecției coliziunilor. 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*[5] ca opțiune pentru motorul de fizică 2D.
- **Bullet**[2] 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 pentru mesh-uri 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*[6], *Blender Game Engine*[7] sau *Unity 3D*[5].

- Din sfera closed source, poate fi amintit **NVIDIA PhysX**[8], unul dintre cele mai populare motoare de fizică în industria jocurilor video - inclus în *Unreal Engine 3+*[3], *Unity 3D*[5] 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 care, chiar dacă nu are aplicabilitate în cazul de față, a condus la următorul rezultat:
- **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 exprimă 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 ale 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.

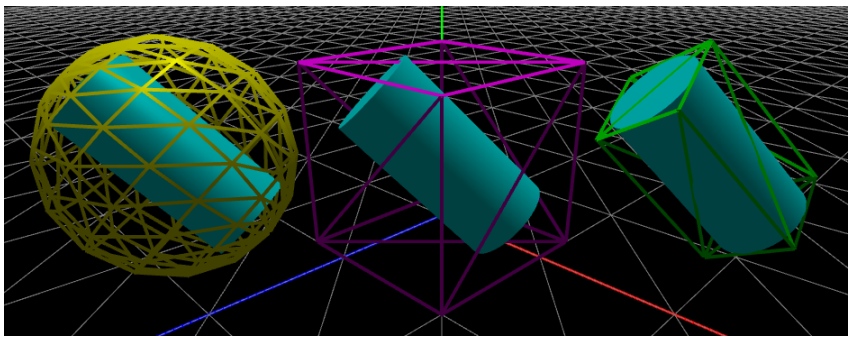


Figura 3.1: sferă încadratoare, AABB, OBB

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*)[9], 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*)[10] sau **octrees** (și variații ale acestora – **quadtrees**)[11].

Etapa exactă

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

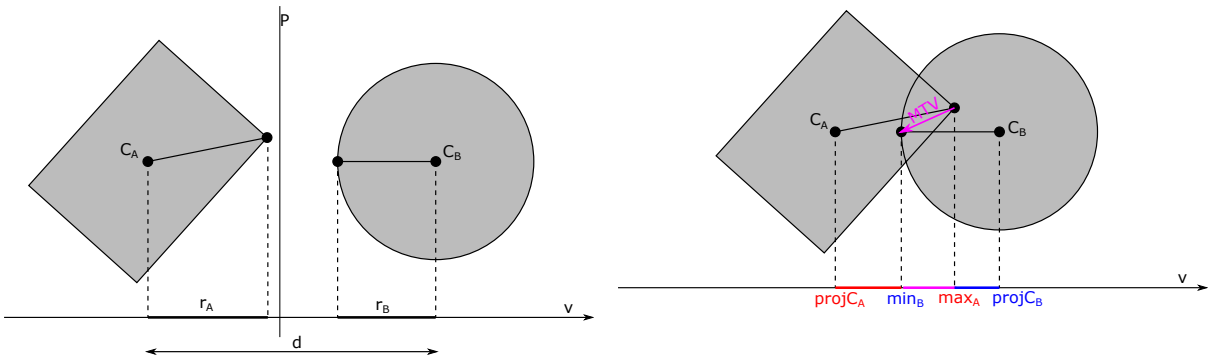
```
struct CollisionPoint {
    vec3 positionA;     // pozitia celui mai adanc punct de interpenetrare
    vec3 positionB;     // in coordonatele locale ale fiecarui obiect
    vec3 normal;        // normala contactului (directia de separare)
    float penetration; // distanta de interpenetrare
    Obj *objA;          // pointer catre fiecare dintre obiecte, pentru a
    Obj *objB;          // accesa 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**[12]:

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 3.2: 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[13], 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[14].

Algoritmul 1 Testul de intersecție Gilbert-Johnson-Keerthi

funcție GJKTestIntersecție($forma\_A$, $forma\_B$, $directie\_initiala$)
  $punct\_nou \leftarrow$ Suport($forma\_A, directie\_initiala$) − Suport($forma\_B, -directie\_initiala$)
  $simplex \leftarrow \{punct\_nou\}$
  $directie \leftarrow -punct\_nou$
  repetă
    $punct\_nou \leftarrow$ Suport($forma\_A, directie$) − Suport($forma\_B, -directie$)
    dacă produs_scalar($punct\_nou, directie$) < 0 **atunci**
      întoarce *Fals*
    sfârșit dacă
    $simplex \leftarrow simplex \cup punct\_nou$
    $simplex, directie, contine\_originea \leftarrow$ ActualizeazăSimplex($simplex$)
    dacă *contine_originea* **atunci**
      întoarce *Adevărat*
    sfârșit dacă
  sfârșit repetă
sfârșit funcție

funcție ActualizeazăSimplex($simplex$)
  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
sfârșit funcție

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**[15] (eng. *expanding polytope algorithm*) poate fi folosit pentru determinarea informațiilor de coliziune. 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) cea mai apropiată de origine pot fi folosite pentru determinarea punctelor de coliziune, iar distanța de la origine la triunghi (sau dreaptă) este chiar distanța de penetrare și normala coliziunii. 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 coliziune

funcție EPACreeazăPunctDeColiziune($forma\_A$, $forma\_B$, $simplex$)
  $politop \leftarrow simplex.lista\_triunghiuri$
  repetă
    $cel\_mai\_apropiat\_triunghi \leftarrow \text{argmin}_{tr}$ DISTANȚA$(tr, origine), tr \in politop$
    $distanta \leftarrow$ DISTANȚA$(cel_mai_apropiat_triunghi, origine)$
    $normala \leftarrow cel\_mai\_apropiat\_triunghi.normala$
    $punct\_nou \leftarrow$ SUPORT$(forma_A, normala) -$ SUPORT$(forma_B, -normala)$
    $distanta\_noua \leftarrow$ DISTANȚA$(punct_nou, origine)$
    dacă $distanta\_noua - distanta < prag$ **atunci**
      $coordonate \leftarrow$ CoordonateBaricentrice$(origine, cel_mai_apropiat_triunghi)$
      $puncte\_coliziune \leftarrow$ pentru fiecare obiect, se calculează punctul de coliziune în funcție de corespondențele fiecărui punct al triunghiului din mulțimea de puncte a obiectului respectiv
      întoarce $puncte\_coliziune, normala, distanta$
    sfârșit dacă
    $politop \leftarrow politop \setminus \{cel\_mai\_apropiat\_triunghi\}$
    creează triunghiuri noi folosind $punct\_nou$ în spațiul lăsat descoperit
  sfârșit repetă
sfârșit funcție

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, determinând 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:

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

Formula completă pentru determinarea impulsului normal $\vec{j_n}$ necesar rezolvării vitezei unui corp după coliziune este:

$$\vec{j_n} = \frac{(-1 + C_R) \cdot \left(\vec{V_B} - \vec{V_A} + \vec{\omega_B} \times \vec{r_B} - \vec{\omega_A} \times \vec{r_B} \right)}{\left(\frac{1}{m_A} + \frac{1}{m_B} \right) + \left((\vec{r_A} \times \vec{n}) \cdot \left(I_A^{-1} * (\vec{r_A} \times \vec{n}) \right) + (\vec{r_B} \times \vec{n}) \cdot \left(I_B^{-1} * (\vec{r_B} \times \vec{n}) \right) \right)} * \vec{n}$$

unde:

- m_A, respectiv m_B sunt masele celor două corpuri
- I_A, respectiv I_B sunt tensorii de inerție în coordonate din spațiul lume pentru cele două corpuri

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**.

$$\vec{j_{t_1}} = \frac{(-1 + C_R) \cdot \left(\vec{V_B} - \vec{V_A} + \vec{\omega_B} \times \vec{r_B} - \vec{\omega_A} \times \vec{r_B} \right)}{\left(\frac{1}{m_A} + \frac{1}{m_B} \right) + \left((\vec{r_A} \times \vec{t_1}) \cdot \left(I_A^{-1} * (\vec{r_A} \times \vec{t_1}) \right) + (\vec{r_B} \times \vec{t_1}) \cdot \left(I_B^{-1} * (\vec{r_B} \times \vec{t_1}) \right) \right)} * \vec{t_1}$$

$$\vec{j_{t_2}} = \frac{(-1 + C_R) \cdot \left(\vec{V_B} - \vec{V_A} + \vec{\omega_B} \times \vec{r_B} - \vec{\omega_A} \times \vec{r_B} \right)}{\left(\frac{1}{m_A} + \frac{1}{m_B} \right) + \left((\vec{r_A} \times \vec{t_2}) \cdot \left(I_A^{-1} * (\vec{r_A} \times \vec{t_2}) \right) + (\vec{r_B} \times \vec{t_2}) \cdot \left(I_B^{-1} * (\vec{r_B} \times \vec{t_2}) \right) \right)} * \vec{t_2}$$

O derivare a formulelor de mai sus poate fi urmărită în prezentarea lui *Erin Catto*[16].

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.

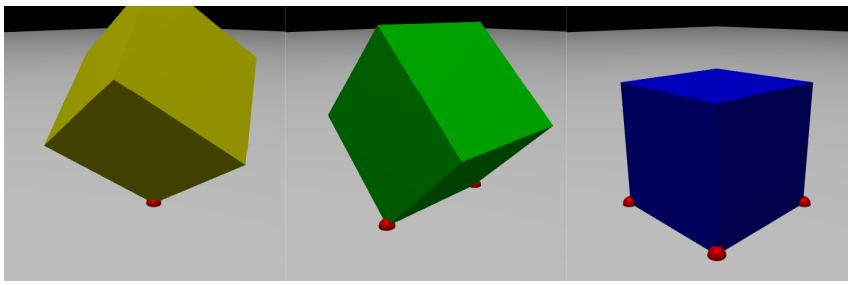


Figura 3.3: 3 cazuri de contact și manifold-urile lor

În acest scop, există 2 posibile abordări [17]:

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.

În plus ar trebui menționat faptul că impulsurile descrise mai sus sunt menite doar să oprească întrepătrunderea corpurilor, și nu să le și separe efectiv. Pentru acest lucru există mai multe alternative[18]:

1. **Stabilizarea Baumgarte** – presupune adăugarea unei părți a distanței de penetrare la termenul de bias din formula pentru determinarea impulsului
2. **Metoda pseudo-vitezelor** – constă în rezolvarea vitezelor corpurilor aflate în coliziune, recalcularea distanței de penetrare, a vitezelor și a maselor și mai apoi aplicarea unei corecții asupra pozițiilor celor două corpuri într-un pas ulterior.
3. **Metoda Gauss-Seidel neliniară** – este asemănătoare cu metoda pseudo-vitezelor, dar presupune recalcularea interpenetrării după fiecare punct de contact rezolvat

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. 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 [19]:

- **Metoda Euler explicită**[20] 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**[21] 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**[22] 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

Soluțiile pentru implementarea unei **interfețe grafice cu utilizatorul** (eng. *graphical user interface - GUI*) sunt întâlnite sub forma unor biblioteci care oferă API-uri programatorilor care doresc să le folosească în proiectele lor. Astfel de biblioteci există în număr destul de mare, iar facilitățile oferite de fiecare variază – de la colecții de widget-uri minimale până la întregi framework-uri pentru dezvoltare de aplicații complete, care oferă instrumente și funcționalități deja implementate care acoperă o multitudine de scenarii de utilizare. O posibilă clasificare a interfețelor grafice este cea bazată pe proprietatea de a reține scena desenată (GUI-ul în sine) de la un cadru la altul. Se disting, astfel:

1. **Interfețele grafice persistente** (eng. *retained mode GUIs*) sunt caracterizate de menținerea unui model intern(de obicei, o ierarhie de obiecte) al interfeței în memorie. Astfel, apelurile de API doar modifică starea acestui model, fără să conducă neapărat la desenarea elementelor. Biblioteca poate, prin urmare, să optimizeze randarea interfeței pe ecran, sau să mențină stări anterioare la care se poate reveni. Sunt soluții robuste, eficiente, rapide, dar necesită un timp mai îndelungat de familiarizare și de învățare a acestora. Exemple includ:
    - **Windows Presentation Foundation**[23] este un framework pentru dezvoltarea de aplicații client pentru sistemul de operare *Windows*. Este un subset al framework-ului *.NET* și utilizează un limbaj de tip Markup(i.e. *XAML*) pentru a oferi un model declarativ de programare. Asigură facilități de desenare (folosind *Direct3D*), șabloane, animații, documente, securitate etc.
    - **Qt**[24] este un alt framework complet pentru dezvoltare de aplicații cross-platform. Este aproape în întregime open source și este scris în C++. Oferă, printre altele, un mediu de dezvoltare complet (*QtCreator*), un limbaj declarativ (*QML*) pentru prototipare, suport pentru lucrul cu baze de date **SQL**, parsare de documente *XML* sau *JSON*, multithreading etc. Ar trebui menționată și documentația foarte bine pusă la punct și comunitatea imensă de utilizatori.
    - **wxWidgets**[25] constă într-un API ușor de folosit pentru scrierea de aplicații grafice pe multiple platforme. Folosește controlul și utilitățile native platformei pentru care se dezvoltă aplicația, este în întregime open source și scris în C++

standard, cu multiple binding-uri pentru alte limbaje de programare populare. Oferă multiple componente grafice – de la un simplu buton până la o fereastră pentru previzualizarea documentelor printate – și sisteme pentru aranjarea elementelor, evenimente, integrare cu motorul de randare HTML nativ, tratare de erori, multithreading etc.

2. De cealaltă parte, **interfețele grafice imediate** (eng. *immediate mode GUIs*) lasă aproape totul la îndemâna utilizatorului(tratarea evenimentelor, management-ul resurselor). Elementele sunt desenate imediat pe ecran și nu mai există o delimitare clară a logicii aplicației de elementele de interfață. Sunt considerabil mai simple, dar mai ușor de extins și de modificat și sunt utile mai ales pentru prototipare, dar și în cazurile în care interfața grafică nu reprezintă o parte centrală a aplicației (de ex. jocuri). Pot fi amintite:
   - **Dear ImGui**[26] este o bibliotecă rapidă, portabilă, și independentă de motorul de randare. A fost dezvoltată pentru C++, nu conține dependințe externe, este open source și se concentrează pe simplitate și productivitate. Se pretează mai ales integrării în game engine-uri, tool-uri pentru vizualizare sau pentru debugging. Conține o multitudine de widget-uri și elemente anexe, care pot fi desenate pe loc, oriunde în aplicație, pentru a obține o strânsă legătură cu logica acesteia.
   - **Nuklear**[27] este un toolkit minimal pentru dezvoltarea de GUI-uri, scris în ANSI C și open source. A fost proiectat să poată fi ușor de integrat în aplicații și constă dintr-un singur fișier header. Are o amprentă redusă asupra memoriei, este personalizabilă și se concentrează strict pe elementele de interfață.

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, utilizând *BSP*, mi s-a considerat nejustificată. Astfel, m-am folosit 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 utilizez **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.

Deoarece rezultatul algoritmului *GJK* poate fi utilizat ca intrare pentru **EPA**, am folosit acest algoritm în determinarea punctelor de coliziune.

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. Corectarea pozițiilor corpurilor o realizez printr-o metodă asemănătoare cu cea a **pseudo-vitezelor**.

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 implicite**, iar precizia altor metode de ordin superior ar fi combătută oricum de micile imperfecțiuni apărute în urma rezolvării coliziunilor. Am ales să păstrez lucrurile simple și să folosesc a doua metodă menționată mai sus.

3.3.4 Interfață grafică

Pentru realizarea interfeței grafice, lipsa de experiență în lucrul cu tool-uri consacrate ca *Qt* sau *WPF* m-au făcut să aleg să folosesc o soluție imediată, care să fie ușor de învățat și de utilizat. Documentația **Dear ImGui** mi s-a părut mai bine realizată și multitudinea de widget-uri și addon-uri deja existente au contribuit la alegerea făcută. Rapiditatea cu care am putut implementa o interfață grafică completă și funcțională m-a surprins plăcut și consider alegerea ca fiind una inspirată.

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 mesh-uri 3D
- suport pentru definirea și încărcarea de shader-e *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 mesh-urilor 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[28] oferit pentru laboratoarele aferente

cursului de *Elemente de Grafică pe Calculator*, din cadrul secției *CTI* a *Universității Politehnica București*. O descriere mai detaliată a acestuia poate fi găsită pe wiki-ul asociat laboratorului[29].

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.

Obiectele scenei

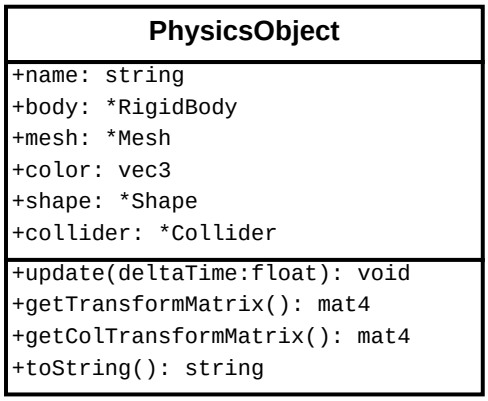


Figura 4.1: Descrierea unui obiect al scenei

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*).

Toate obiectele scenei sunt menținute într-un obiect de tip **ObjectSpawner** care se ocupă de instanțierea și actualizarea lor și oferă funcționalități de nivel înalt aplicației.

ObjectSpawner

```
+objects: list<PhysicsObject*>
+meshes: *unordered_map<string, Mesh*>
+pcd: *PotentialCollisionDetector
+spawnPosition: vec3
+spawnDirection: vec3
+offsetInFront: float
+applyStartingImpulse: bool
+selectedObject: *PhysicsObject
+nextObjectType: string
+randomizeNextObject: bool
+randomizeProperties: bool
```

```
+updateFromCamera(in cameraPosition:vec3,
                  in forward:vec3): void
+updateObjects(): void
+spawnNewObject(): void
+loadFromFile(in filename:string): void
+saveToFile(in filename:string): void
+spawnBoxDynamic(): *PhysicsObject
+spawnBoxStatic(in position:vec3,in scale:vec3,
                in mass:float,in orientation:quat): *PhysicsObject
+spawnLongBoxDynamic(): *PhysicsObject
+spawnLongBoxStatic(in position:vec3,in scale:vec3,
                    in mass:float,in orientation:quat): *PhysicsObject
+spawnWallStatic(in position:vec3,in scale:vec3,
                 in orientation:quat): *PhysicsObject
+spawnSphereDynamic(): *PhysicsObject
+spawnSphereStatic(in position:vec3,in scale:vec3,
                   in mass:float,in orientation:quat): *PhysicsObject
+spawnCylinderDynamic(): *PhysicsObject
+spawnCylinderStatic(in position:vec3,in scale:vec3,
                     in mass:float,in orientation:quat): *PhysicsObject
+spawnCapsuleDynamic(): *PhysicsObject
+spawnCapsuleStatic(in position:vec3,in scale:vec3,
                    in mass:float,in orientation:quat): *PhysicsObject
+toString(): string
```

Figura 4.2: Structura unui obiect ObjectSpawner

Corpurile solide

Starea cinematică a obiectului este încapsulată într-un obiect **RigidBody**(descris în întregime în [figura B.1]). 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

Detecția de coliziuni

Mai departe, controlul aplicației este preluat de **detectorul de coliziuni**, a cărui structură completă poate fi găsită în [figura B.2].

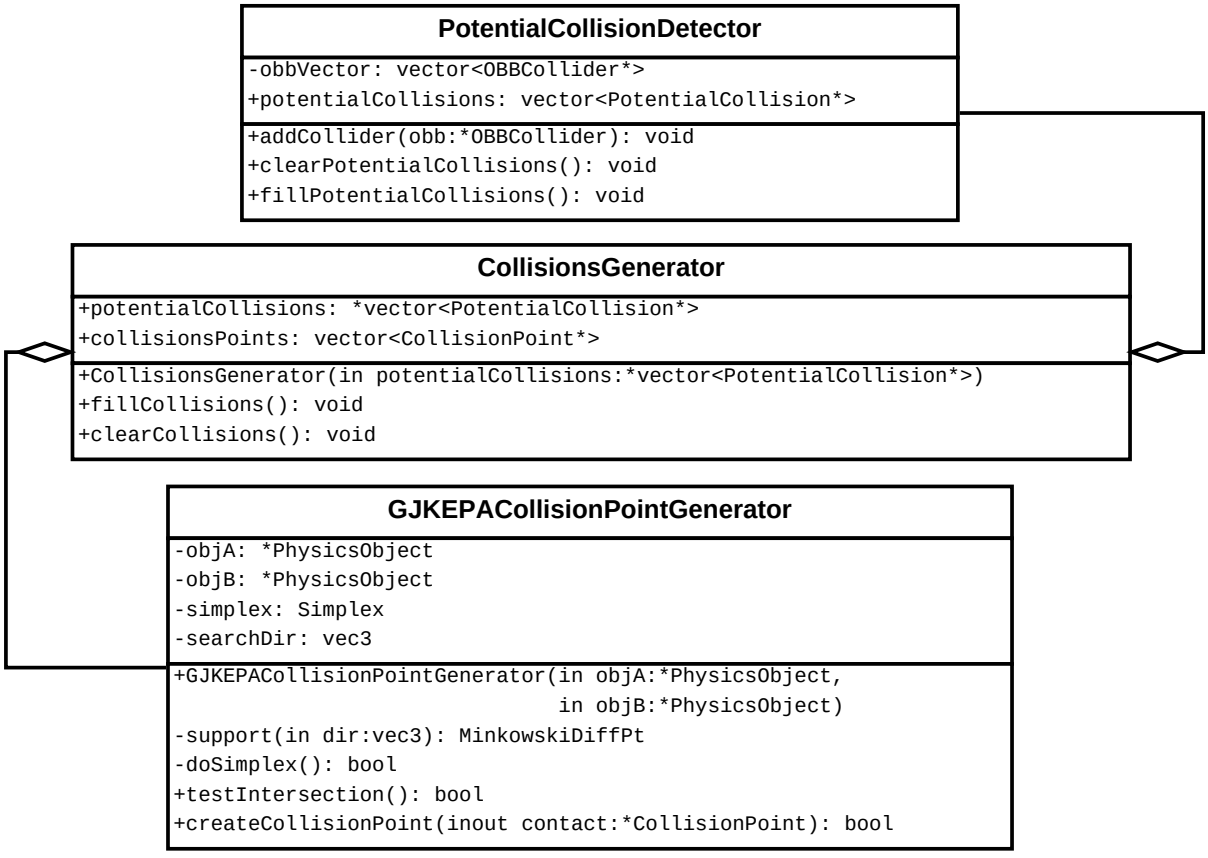


Figura 4.3: Subsistemul de detecție de coliziuni

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.

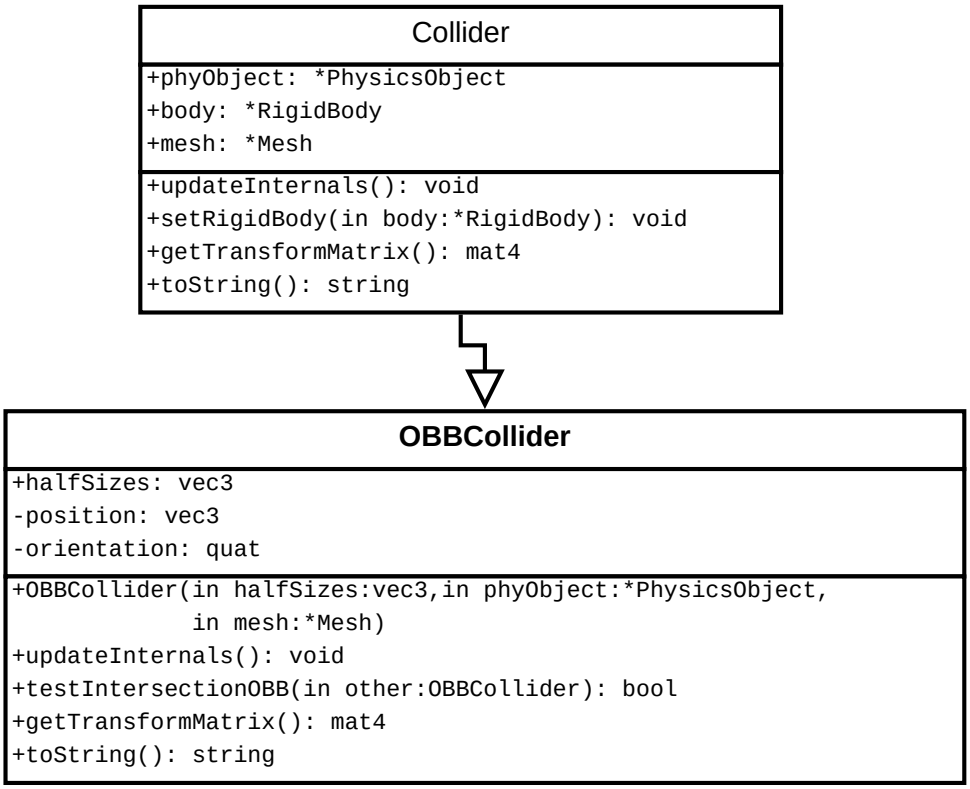


Figura 4.4: OBBCollider implementează clasa abstractă Collider

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**, a cărui structură completă poate fi observată în [figura B.3].

Rezolvarea coliziunilor

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

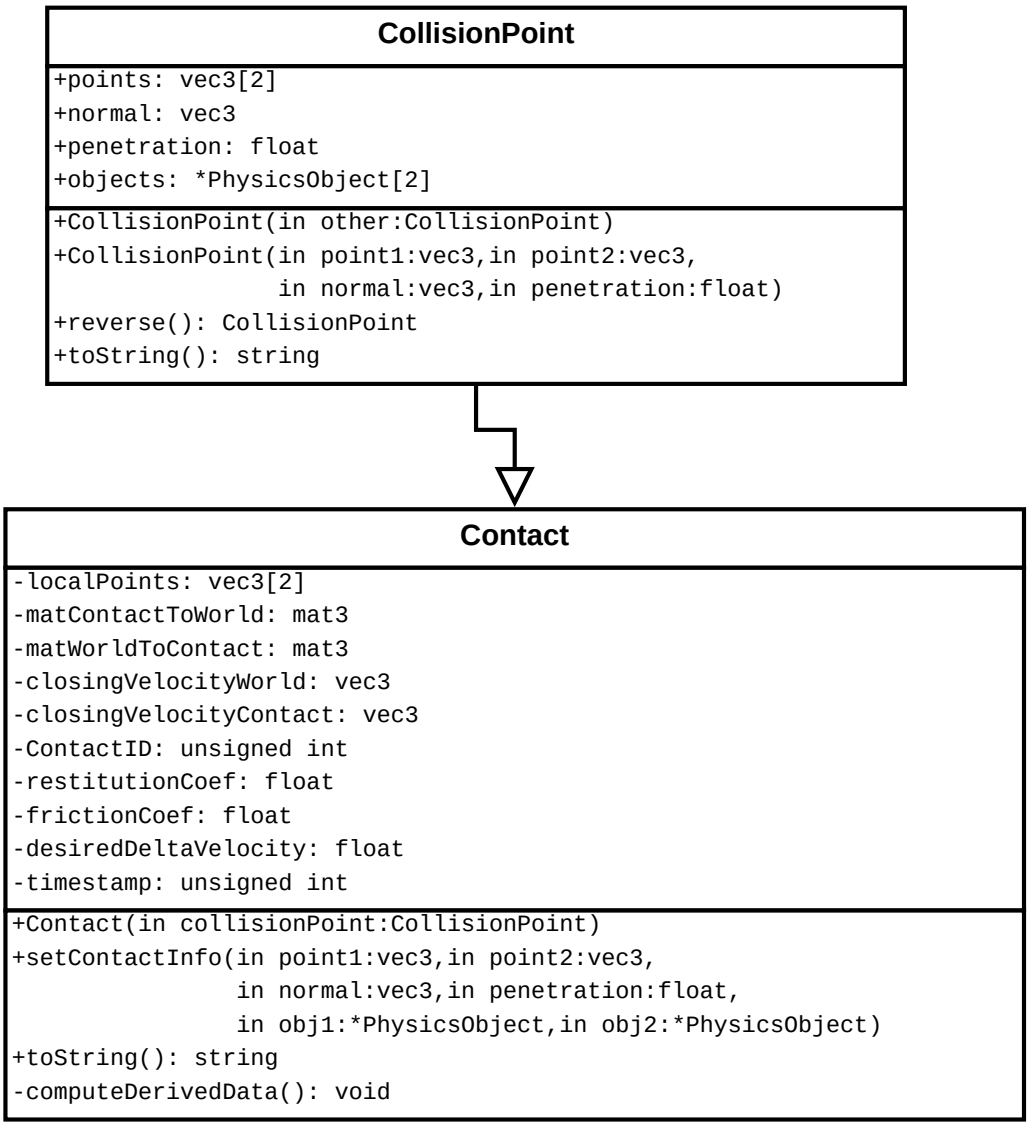


Figura 4.5: diferența dintre punctul de coliziune și punctul de contact
(a se observa că Contact moștenește CollisionPoint)

- 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

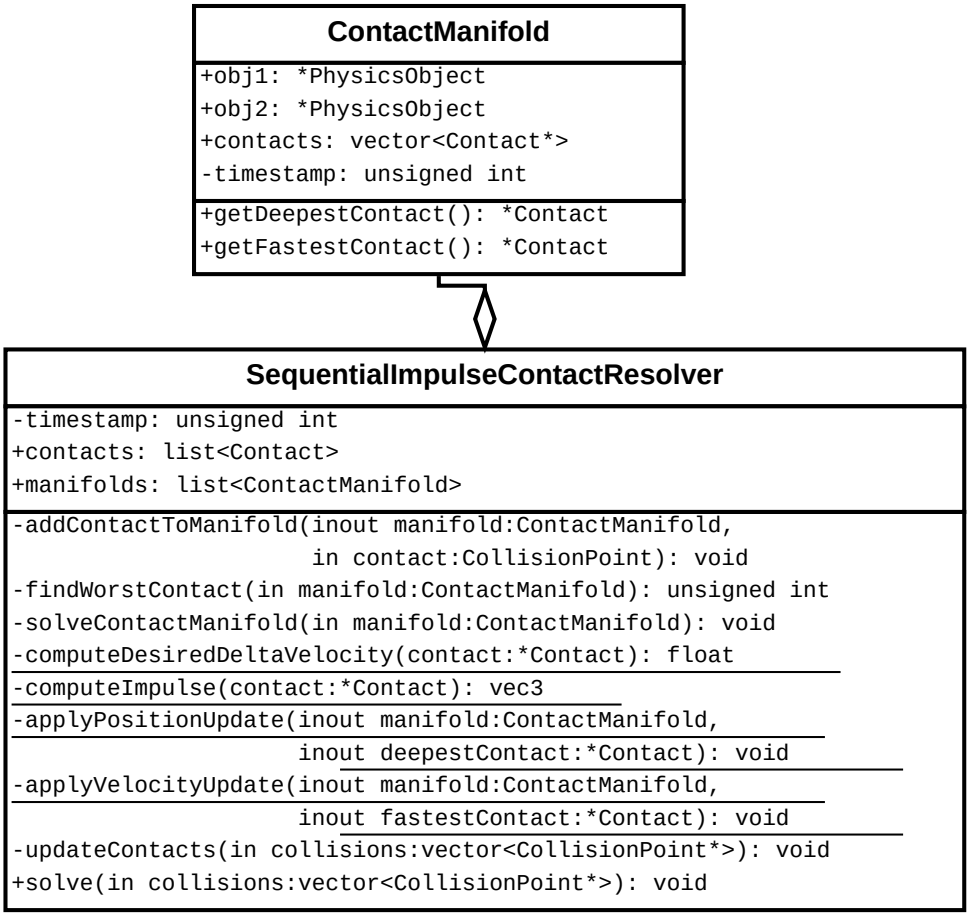


Figura 4.6: Subsistemul de rezolvare a coliziunilor

Rezolvarea completă a coliziunilor presupune rezolvarea tuturor manifold-urilor de contact. La nivelul unui singur manifold au loc următorii pași:

1. identificarea celui mai adânc contact (cu distanța de penetrare cea mai mare)
2. rezolvarea erorii de poziție a corpurilor pe baza informațiilor acestui contact
3. repetarea pașilor 1 și 2 cât timp iterațiile pentru rezolvarea pozițiilor nu sunt epuizate
4. identificarea celui mai rapid contact (cu viteza de separare (valoare negativă!) cea mai mică)
5. rezolvarea vitezelor corpurilor pe baza informațiilor acestui contact
6. repetarea pașilor 4 și 5 cât timp iterațiile pentru rezolvarea vitezelor nu sunt epuizate

4.3 Interfața cu utilizatorul

Posibilitatea utilizatorului de a interacționa cu simularea este oferită de o interfață minimală. Aceasta este bazată atât pe input-ul de la mouse și tastatură, cât și pe un GUI care să ofere control mai fin asupra parametrilor.

Astfel, utilizatorul poate selecta câte un obiect de pe ecran dând click pe acesta și îi poate controla poziția și orientarea folosind tastatura. În plus, controlul camerei oferite de backend se realizează tot cu ajutorul tastaturii, atunci când butonul dreapta al mouse-ului este ținut apăsat. Totodată, utilizatorul poate introduce oricând dorește obiecte noi în scenă, atât static, la inițializare, cât și dinamic, în timpul rulării aplicației.

Interfața grafică oferă posibilitatea de a controla în timp real parametrii simulării (elementele desenate, gravitația, numărul de iterații în cadrul diferiților algoritmi, precizie etc.). Utilizatorul primește feedback în timp real, valorile fiind în permanență afișate. În plus față de setările ce țin de simulare, mai pot fi alterate proprietățile (masă, poziție, orientare, coeficienți de frecare / elasticitate etc.) obiectului selectat cu mouse-ul.

În ultimul rând, pentru a putea testa și crea diferite simulări într-un mod cât mai facil, există posibilitatea de a încărca scene (formate din parametrii de simulare și lista cu obiectele din scenă) definite în fișiere *.json* sau de a salva starea actuală a simulării într-o nouă astfel de scenă.

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 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 mesh-uri 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 utilizată pentru tipuri și funcții matematice și geometrice este **GLM**[30].

5.1 Reprezentarea corpurilor solide

Primul obiectiv atins a fost **implementarea clasei RigidBody**[figura B.1] și, implicit, a operației de integrare. Reprezentarea aleasă este una standard. Accelerația liniară și cea unghiulară sunt mărimi primare, determinate din forțele sau momentele care sunt aplicate corpului. Viteza (atât liniară, cât și unghiulară) este determinată pe baza acestor accelerații, iar poziția și orientarea corpului sunt calculate cu ajutorul vitezelor și sunt mărimile finale, care codifică starea corpurilor.

O mențiune specială este că am ales să stochez orientarea corpului într-un **quaternion**, în loc de o reprezentare matriceală sau folosind *unghiurile lui Euler*. Motivul este foarte simplu, ocupă mai puțin spațiu în memorie, nu prezintă problema *Gimbal Lock*[31], iar

suportul *GLM* pentru operații cu quaternioni este foarte bun, incluzând atât adunări, înmulțiri, rotații, cât și funcții pentru obținerea unei matrice de rotație dintr-un quaternion sau viceversa.

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

Extras de cod 5.1: actualizarea orientării în pasul de integrare

5.2 Detecția coliziunilor

A doua etapă de dezvoltare a constat în implementarea sistemului de **detecție a coliziunilor**. Complexitatea etapei a fost semnificativ mai mare, și am întâmpinat mai multe dificultăți.

5.2.1 Etapa preliminară

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 implementarea optimă propusă de *Christer Ericson*[32]. Tot aici, am fost nevoit să implementez abstractizarea obiectelor din scenă, care să încapsuleze obiectele *RigidBody* și *Collider*.

5.2.2 Etapa exactă

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*.

GJK

Pentru **implementarea GJK**, foarte utilă mi s-a părut explicația lui *Casey Muratori*[33], care a oferit o interpretare mult mai ușor de înțeles a algoritmului decât enunțarea lui inițială[13]. Remarcabil mi se pare că în cazul unui simplex tetraedru, am obținut până la 4 teste condiționale imbricate, pentru determinarea precisă a noului simplex și a noii direcții. Includ aici cazul muchie, deoarece mi se pare că algoritmul în pseudocod nu spune prea multe cititorilor nefamiliarizați cu acesta.

```
void GJKEPA::GJKEPACollisionPointGenerator::doSimplex2()
{
    /* simplex is an edge \vec{AB}, A was just added */
    glm::vec3 vecAO = -simplex.a.v;
    glm::vec3 vecAB = simplex.b.v - simplex.a.v;
    /* origin is in the direction of \vec{AB}
    search direction is perpendicular to \vec{AB} and coplanar with \vec{AO
        } */
    if (glm::dot(vecAB, vecAO) > 0) {
        searchDir = glm::normalize(glm::cross(glm::cross(vecAB, vecAO),
            vecAB));
    }
    else {
        simplex.set(simplex.a);
        searchDir = glm::normalize(vecAO);
    }
}
```

Extras de cod 5.2: cazul muchie în algoritmul GJK

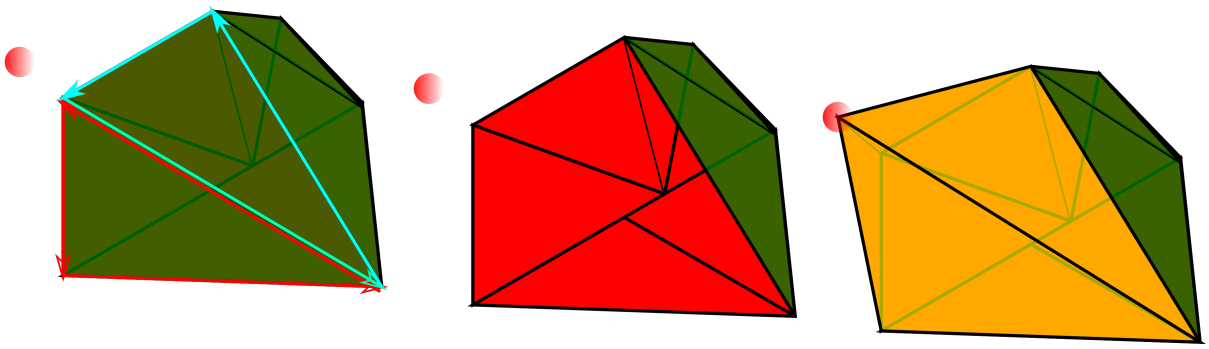
Ofer și codul pentru tratarea cazului triunghi în [extrasul de cod A.1].

EPA

Un alt impediment 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 5.1: Un pas de extindere al poliedrului în cadrul EPA

Secvența 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));
}
```

Extras de cod 5.3: extinderea politopului în cadrul EPA

5.3 Rezolvarea coliziunilor

Rezolvarea coliziunilor este componenta cea mai complexă a simulatorului din punct de vedere matematic și programatic. Procesul a fost descris mai pe larg în capitolele precedente, dar reiterez faptul că am ales o metodă de rezolvare a interpenetrării bazată pe metoda pseudo-vitezelor. În plus, informațiile de coliziune sunt încapsulate în structuri numite *contacte*, care pot supraviețui de la un cadru la altul și sunt grupate în manifold-uri diferite pentru fiecare pereche de obiecte aflate în coliziune. Astfel, rezolvarea unui manifold de contact se realizează în felul următor:

Sunt calculate pozițiile relative, coeficienții de frecare și elasticitate, viteza corpurilor dorită după rezolvare.

```
void SequentialImpulseContactResolver::solveContactManifold(ContactManifold
    &manifold, float deltaTime)
{
    unsigned int penetrationIterations = 0;
    unsigned int velocityIterations = 0;

    for (auto & contact : manifold.contacts) {
        for (uint8_t i = 0; i < 2; i++) {
            if (contact->objects[i]->collider->body != nullptr)
                contact->objects[i]->collider->body->isAwake = true;
        }

        contact->computeDerivedData(deltaTime);
        contact->desiredDeltaVelocity = computeDesiredDeltaVelocity(contact
            );
    }
[cont.]
```

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.

```
[cont.]
    /* fix interpenetration */
    for (penetrationIterations = 0; penetrationIterations < PhysicsSettings
        ::get().collisionResolution.penMaxIterations; penetrationIterations
        ++) {
        /* get the contact with deepest penetration */
        Contact *deepestContact = manifold.getDeepestContact();

        if (deepestContact == nullptr || deepestContact->penetration <
            PhysicsSettings::get().epsilons.penEpsilon) {
            penetrationIterations++;
```

```
            break;
        }

        /* compute and apply position update */
        applyPositionUpdate(manifold, deepestContact);
    }
[cont.]
```

Se rezolvă vitezele corpurilor prin metoda impulsurilor secvențiale. Calculul impulsului care să stabilească viteza corectă de după coliziune se bazează din nou, pe componentele individuale – cea liniară și cea unghiulară.

```
[cont.]
    /* fix velocities */
    for (velocityIterations = 0; velocityIterations < PhysicsSettings::get
        ().collisionResolution.velMaxIterations; velocityIterations++) {
        /* get the contact with the lowest desired delta velocity (aka the
            fastest one) */
        Contact *fastestContact = manifold.getFastestContact();

        if (fastestContact == nullptr || fastestContact->
            desiredDeltaVelocity >= PhysicsSettings::get().epsilons.
            velEpsilon) {
            velocityIterations++;
            break;
        }

        /* compute and apply velocity update */
        applyVelocityUpdate(manifold, fastestContact);
    }
}
```

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[34] lui *Ian Millington*, iar soluția mea urmează îndeaproape indicațiile și implementarea acestuia.

5.4 Controlul obiectelor

Controlul obiectelor din scenă este realizat cu ajutorul unui obiect **ObjectSpawner**[figura 4.2], care menține o listă cu toate obiectele și care se ocupă de instanțierea și de actualizarea lor. Folosirea acestui obiect care să abstractizeze operațiile de bază (instanțiere, actualizare, încărcare/salvare scene) asigură un cod cât mai lizibil în aplicația principală și

posibilitatea de a folosi aceste apeluri chiar și în cadrul interfeței grafice. Pentru simplitate, poziția acestuia este determinată pe baza poziției camerei și un deplasament față de aceasta. Exemple de folosire:

```
ObjectSpawner *spawner = new ObjectSpawner();
[...]
// instantiaza un nou obiect la runtime, cu parametri aleatori(sau nu),
    stabiliti de spawner
spawner->spawnNewObject();
// instantiaza un nou perete cu parametri alesi de utilizator
spawner->spawnWallStatic(position, scale, orientation);
// incarca o noua scena din fisier
spawner->loadFromFile("test.json");
// elimina obiectele prea indepartate si apeleaza update() pentru fiecare
    obiect
spawner->updateObjects();
```

5.5 Interfața grafică

Folosirea **Dear ImGui** nu a ridicat nicio problemă majoră. A trebuit să redirectez evenimentele de tastatură și scroll către cele definite în exemplul oficial de folosire a bibliotecii cu *GLFW*, dar integrarea a fost facilă în rest. Definirea UI-ului am realizat-o într-un fișier separat, iar funcția pentru desenarea acestuia este apelată la sfârșitul fiecărui cadru. Am preluat în plus un addon pentru interfațarea cu sistemul de fișiere la alegerea unui fișier din care să se încarce o nouă scenă și să îmi definesc un nou widget pentru afișarea personalizată a celor 3 câmpuri dintr-un *vec3*. Fiind vorba de un GUI imediat, elementele de logică sunt strâns legate de cele ce țin de desenare. Exemplu din *ui.cpp*:

```
PhysicsSettings *settings = PhysicsSettings::get()
[...]
if (ImGui::TreeNode("rigid bodies default values"))
{
    ImGui::PushItemWidth(128);
    ImGui::DragFloat("mass", &settings->rigidBodies.defaultMass, 1.0f, 0.0f
        , 10000.0f, "%.1f");

    ImGui::DragFloat("friction coefficient", &settings->rigidBodies.
        defaultFrictionCoef, 0.005f, 0.0f, 1.0f);
    ImGui::DragFloat("restitution coefficient", &settings->rigidBodies.
        defaultRestitutionCoef, 0.005f, 0.0f, 1.0f);

    ImGui::PopItemWidth();
    ImGui::TreePop();
}
```

Extras de cod 5.4: widget-urile pentru stabilirea parametrilor impliciti ai corpurilor solide

5.6 Parametrii simulării

Pentru a nu hardcoda valori literale direct în cod, pentru a avea acces facil la ele, și pentru a le putea modifica la runtime, toate setările aplicației – parametri de simulare, constante, erori admisibile, parametri impliciți pentru obiecte, configurația spawner-ului sau elementele desenate – sunt reținute într-un obiect **PhysicsSettings**, care este un singleton accesibil de oriunde este nevoie în aplicație. Cea mai mare parte dintre acestea pot fi definite și în cadrul scenelor încărcabile în format *.json*.

```
{
        "settings" : {
                "gravity" : {
                        "x" : 0.0,
                        "y" : -20.0,
                        "z" : 0.0
                },
                "timeScale" : 1.0,
                "rendering" : {
                        "renderColliders" : false
                }
        },
        "objects" : [
                {
                        "type" : "box",
                        "name" : "cube0",
                        "position" : {
                                "x" : 12.6,
                                "y" : 4.0,
                                "z" : 0.0
                        },
                        "frictionCoef" : 0.6,
                        "restitutionCoef" : 0.1,
                        "mass" : -1.0
                }
        ]
}
```

Extras de cod 5.5: Exemplu de scenă în format .json

Un exemplu mai complet poate fi găsit în [extrasul de cod A.2].

6 EVALUARE

Pentru evaluarea aplicației obținute, voi încerca să fac o analiză din perspectiva **obiectivelor propuse** inițial, dar și din punct de vedere al **performanței** acesteia.

6.1 Atingerea obiectivelor

La sfârșitul dezvoltării aplicației, am reușit cu succes să ating toate obiectivele pe care mi le-am propus. Am obținut un sistem pentru simularea interacțiunilor mecanice (ciocniri) între corpuri solide într-un spațiu 3D. Răspunsul la aceste interacțiuni este credibil pentru utilizatorul obișnuit, iar interfața pusă la dispoziție permite un control fin asupra scenei.

Includ o listă completă de facilități oferite și suportate, care să evidențieze cele spuse mai sus:

- reprezentarea corpurilor 3D în scenă se realizează cu ajutorul obiectelor de tip *RigidBody*[figura B.1], în care:
  - masa, factorii de scalare, coeficienții de frecare/elasticitate sunt mărimi constante
  - forțele și momentele forței care acționează asupra corpurilor sunt mărimi brute care sunt folosite la ajustarea mărimilor secundare
  - vitezele și accelerațiile liniare și unghiulare sunt mărimi secundare, care sunt utilizate la actualizarea mărimilor finale
  - poziția și orientarea sunt mărimi finale, care determină starea absolută a corpurilor într-un moment de timp t
- corpurile au atribuite forme elementare și volume încadratoare, astfel încât să se poată detecta coliziunile dintre acestea în două etape:
  - în etapa preliminară, se realizează teste de intersecție mai puțin precise, pe baza volumelor încadratoare

  - în etapa exactă, se identifică toate coliziunile reale și sunt determinate informațiile care să ajute la rezolvarea acestora
- toate coliziunile identificate sunt soluționate printr-un răspuns aplicat corpurilor implicate, care are rolul de a:
  - asigura un set de viteze liniare și unghiulare post-coliziune realiste, care să respecte, pe cât posibil, legile fizicii
  - separa corpurile imediat, în cazul întrepătrunderilor vizibile
- toate obiectele și alte elemente anexe(puncte de coliziune, normale, volume încadratoare etc.) sunt desenate pe ecran, într-o fereastră *OpenGL*
- este oferită o interfață cu utilizatorul care să îi permită acestuia să:
  - încarce sau să salveze scene în format *.json* [figura C.2]
  - instanțieze noi obiecte
  - controleze un obiect individual selectat de acesta(atât starea, cât și parametrii constanți) [figura C.1]
  - altereze setările simulării [figura C.4]
  - poată naviga în scenă
  - aleagă exact ce elemente dorește să fie desenate [figura C.3]
- corpuri 3D suportate: cub, paralelipiped dreptunghic, sferă, cilindru, capsulă și variații obținute prin scalare ale acestora
- sisteme de operare suportate: *Windows*
- biblioteci adiționale: doar cele incluse în folder-ul */libs*

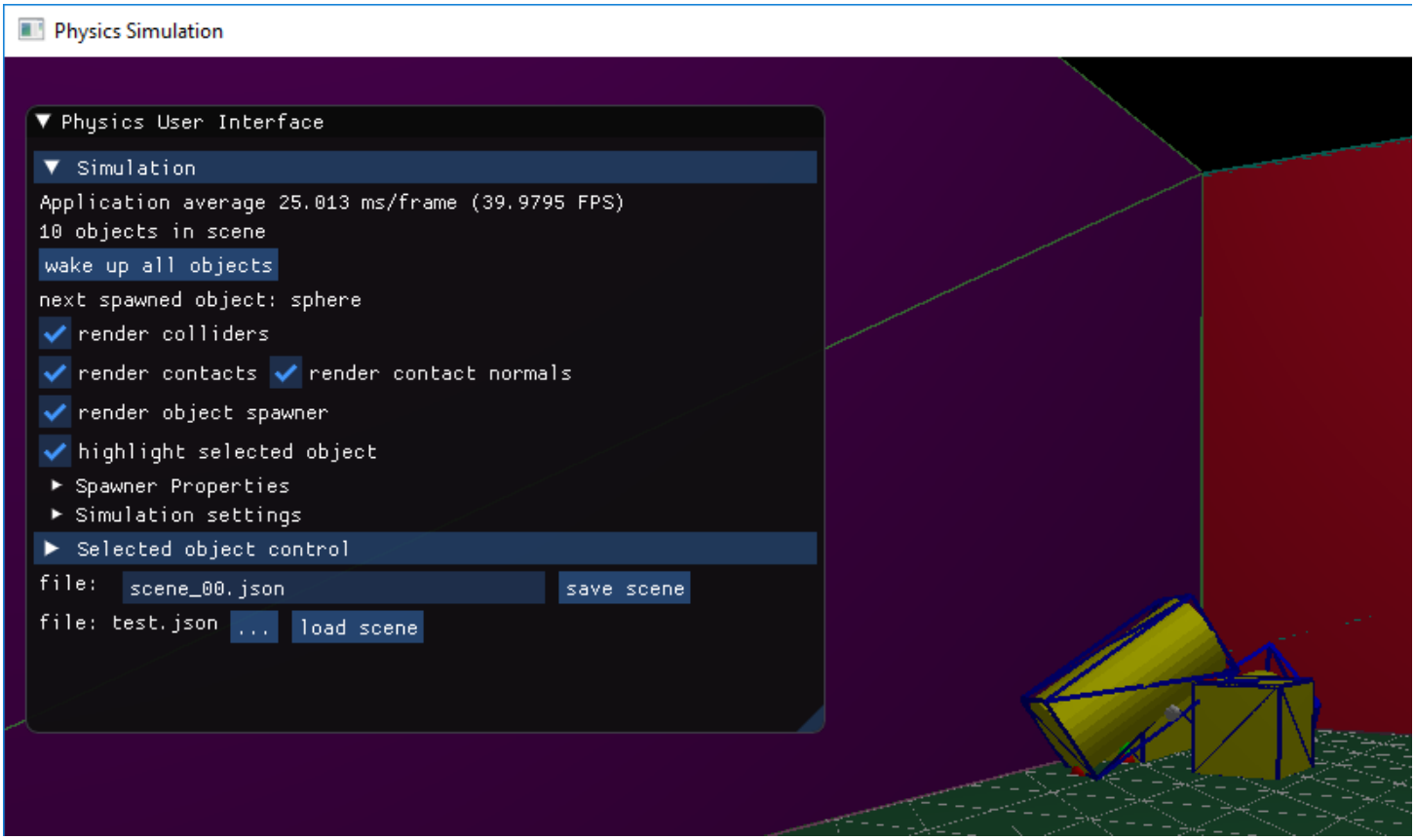


Figura 6.1: Interfața grafică oferită

6.2 Evaluare performanțe

Pentru evaluarea performanțelor aplicației, propun 3 scenarii de test diferite. Parametrii simulării și cei impliciți ai corpurilor sunt cei din [extrasul de cod A.2], dacă nu este specificat altfel. Toate testele au fost efectuate pe laptop-ul personal (*Intel i5* 2.5GHz, 8GB RAM, *Intel HD Graphics 4600*), utilizând mediul de dezvoltare *Visual Studio Community Edition 2017*.

6.2.1 Stive de obiecte

Pentru acest test, am considerat un număr variabil de cuburi unitare stivuite unul peste celălalt deasupra unui perete static care ține loc de podea și am măsurat timpul până când oricare două cuburi alăturate nu mai sunt în contact.

Creșterea pragului de mișcare pentru trecerea obiectelor în starea de sleep poate duce la

Nr. cuburi	**Timp până la separare (secunde)**	**Observații**
1	∞	cubul este complet stabil
2	40	vibrații foarte mici, dar ansamblul este stabil
3	30	vibrații foarte mici, dar ansamblul este stabil
4	30	vibrații foarte mici, dar ansamblul este stabil
5	25	cuburile alunecă în mod vizibil
6	15	vibrații observabile, cuburile alunecă în mod vizibil
7	12	vibrații observabile, cuburile alunecă în mod vizibil
8	8	vibrații observabile, cuburile alunecă în mod vizibil
16	4	stiva este complet instabilă

Tabela 6.1: Testul pentru stive de obiecte

stabilizarea stivelor de 2, 3, sau 4 obiecte. Pentru uz general, consider rezultatele obținute mulțumitoare.

6.2.2 Număr de obiecte în scenă

Pentru acest test, am considerat un număr variabil de cuburi instanțiate deasupra solului și lăsate să cadă. Cuburile sunt instanțiate astfel încât acestea să se afle în contact sau nu. La impactul cu solul, se va aplica un răspuns pentru coliziune. Am măsurat numărul de cadre efectuate pe secundă.

Așa cum era de așteptat, simularea devine instabilă când în scenă se află un număr mare de obiecte. În plus, rezolvarea coliziunilor are o influență semnificativă asupra vitezei aplicației.

Pentru a doua parte a acestui test, am determinat timpul petrecut de aplicație în fiecare dintre subsistemele majore ale acesteia pentru o scenă cu 32 de cuburi aflate în contact cu solul.

Nr. cuburi	**Nr. cadre pe secundă (în cădere)**	**Nr. cadre pe secundă (la sol)**	**Observații**
1	60.0	59.9	funcționare fluidă
2	60.0	59.9	funcționare fluidă
4	60.0	59.8	funcționare fluidă
8	59.9	59.6	funcționare fluidă
16	59.6	58.9	funcționare fluidă
32	59.2	57.7	funcționare acceptabilă
64	50.0	30.4	o cădere mare, dar simularea este încă stabilă
128	18.5	12.3	simularea este instabilă
256	6.3	0.0	simularea este complet instabilă

Tabela 6.2: Testul pentru număr de obiecte în scenă

Etapa	**timp (ms)**
desenare obiecte	4.06
integrare	0.22
detecție posibile coliziuni	1.33
stabilire puncte de contact	3.54
rezolvare contacte	0.76
desenare UI	2.66

Tabela 6.3: Timpul petrecut în fiecare dintre subsistemele simulatorului într-un cadru

Rezultatele nu sunt extrem de surprinzătoare. Etapa cea mai costisitoare, excluzând pe cele de desenare, este generarea punctelor de contact. Complexitatea algoritmilor folosiți este potrivită doar pentru teste de dimensiuni mici. Pentru măsurare am folosit profiler-ul oferit de *Visual Studio 2017*, care oferă detalii exacte asupra căror apeluri de funcții au ocupat cei mai mulți cicli de procesor.

6.2.3 Plan înclinat

Acest test nu are fundamente obiective, dar verifică funcționarea corectă a frecării în cadrul rezolvării contactelor. Toate tipurile de obiecte sunt instanțiate deasupra unui perete care joacă rolul unui plan înclinat de 30 deg. Sunt considerate 3 situații: coeficientul de frecare atât al corpurilor, cât și al planului înclinat ia pe rând fiecare dintre valorile 0.0, 0.5, respectiv 1.0.

În cazul sferei, ar trebui să se poată observa cu ușurință ca aceasta nu se rotește deloc în cazul fără frecare și se va roti puțin sau mai mult în celelalte două cazuri. Pentru vizualizare, folosesc *OBB*-ul asociat sferei, iar ca metrică, rețin norma maximă a vitezei unghiulare atinsă de sferă.

Coeficient de frecare	**Viteza unghiulară maximă atinsă**
0.0	0.02
0.5	1.04
1.0	1.32

Tabela 6.4: Testul pentru verificarea tratării corecte a frecărilor

Se observă diferența clară între cazul fără frecare și celelalte două. Pentru o vizualizare și mai bună, pot fi încărcate scenele din /demos/slope/.

6.2.4 Utilizare resurse

Câteva detalii despre resursele utilizate de aplicație:

- aplicația rulează pe un singur thread
- *Visual Studio* raportează o amprentă în memorie de 45.5 MB; cea mai mare parte este ocupată de biblioteci și alocări realizate de backend
- dimensiunea heap-ului se menține în jurul valorii de 5900 KB (de exemplu, încărcarea unei scene cu 32 de obiecte va produce 265 de alocări totalizând 34.80 KB)

Per total, aplicația nu a atins nivelul de performanță pe care mi l-aș fi dorit, optimizările

cele mai esențiale care ar trebui efectuate fiind cele ce țin de alocarea și dealocarea inteligentă a memoriei la runtime, dar și optimizări ale algoritmilor folosiți în cadrul simulatorului.

7 CONCLUZII

În urma realizării acestui proiect, am atins toate obiectivele pe care mi le-am propus, ceea ce mă face să îl consider un succes.

Pe plan teoretic, am reușit să aprofundez suficient de multe cunoștințe astfel încât pe viitor să pot extinde și îmbunătăți aplicația obținută sau să contribui la alte proiecte asemănătoare. Am dobândit o înțelegere foarte bună a arhitecturii unui motor de fizică și a câtorva dintre algoritmii și metodele folosite des în industrie. Mi-am exersat abilitățile de a programa în C++, mi-am reîmprospătat cunoștințele de *OpenGL* și m-am familiarizat cu biblioteci noi care mi-ar putea fi utile și în viitor.

Practic, am obținut o aplicație grafică *OpenGL*, capabilă să ruleze o simulare de dimensiune rezonabilă (max. 64 de obiecte) a interacțiunilor mecanice dintre corpuri 3D pe sistemul de operare *Windows*. În plus, am implementat cu succes o interfață grafică pentru utilizator, prin care acesta poate încărca sau salva scene, controla parametrii simulării și direct obiectele din scenă.

În realizarea aplicației:

- am implementat o reprezentare a corpurilor solide robustă și minimală
- am implementat o serie de corpuri 3D (cub, paralelipiped dreptunghic, sferă, cilindru, capsulă) care să aibă asociate câte un obiect *RigidBody* și care să participe în simulare
- am implementat un integrator numeric bazat pe *metoda Euler implicită*
- am implementat un sistem de detecție preliminară a coliziunilor, cu ajutorul volumelor încadratoare de tip *OBB*
- am implementat algoritmii *GJK* și *EPA* pentru detecția și calculul punctelor de coliziune
- am implementat un rezolvitor de coliziuni bazat pe *metoda impulsurilor secvențiale*
- am folosit *Dear ImGui* pentru realizarea interfeței grafice

- am testat și am oferit spre folosire o serie de demo-uri care să ilustreze capabilitățile simulatorului

Limitările aplicației sunt:

- numărul mic de forme care pot fi simulate – nu există posibilitatea de a încărca mesh-uri poligonale complexe
- performanța mai puțin decât ideală
- realismul simulării este mulțumitor, dar nu excelent
- lipsa suportului pentru alte tipuri de constrângeri mecanice (cum ar fi articulațiile)

Acestea sunt dictate de:

- erori numerice, datorate lucrului cu numere în virgulă mobilă
- utilizarea unor parametri neoptimi în cadrul simulării
- bug-uri și calitatea implementării
- folosirea unui algoritm ineficient pentru detecția preliminară a coliziunilor
- utilizarea ineficientă a memoriei

Acestea fiind spuse, mențin posibilitatea ca pe viitor să continui lucrul la aplicație. Cele mai importante adăugiri pe care le-aș putea face sunt:

- implementarea unei ierarhii de de volume încadratoare pentru a reduce complexitatea algoritmului $O(n^2)$ folosit în prezent în faza preliminară a detecției de coliziuni
- alternativ, implementarea unui arbore al scenei de tip *BSP* sau octree
- oferirea unui suport pentru încărcarea de mesh-uri convexe și adaptarea algoritmilor deja implementați la acest caz general
- implementarea și altor volume încadratoare pentru a putea compara performanțele acestora
- rafinarea parametrilor simulării
- implementarea și altor algoritmi, cum ar fi metoda contactelor simultane
- optimizarea lucrului cu heap-ul, prin implementarea unui alocator propriu
- repararea bug-urilor existente

BIBLIOGRAFIE

[1] E. Catto, "Box2D." https://github.com/erincatto/Box2D, 2014. ultima accesare: 18 August 2018.

[2] E. Coumans, "Bullet Physics." https://github.com/bulletphysics/bullet3, 2017. ultima accesare: 18 August 2018.

[3] Epic Games, "Unreal Engine." https://docs.unrealengine.com/en-us/. ultima accesare: 18 August 2018.

[4] Valve Corporation, "Source." https://developer.valvesoftware.com/wiki/SDK_Docs. ultima accesare: 18 August 2018.

[5] Unity Technologies, "Unity." https://docs.unity3d.com/Manual/index.html. Ultima accesare: 18 August 2018.

[6] J. Linietsky, "Godot Engine." https://github.com/godotengine/godot, 2018. Ultima accesare: 18 August 2018.

[7] B. Wallace, "Blender Game Engine." https://docs.blender.org/manual/en/dev/game_engine/index.html, 2017. Ultima accesare: 18 August 2018.

[8] NVIDIA Corporation, "PhysX." https://developer.nvidia.com/physx-sdk, 2017. Ultima accesare: 18 August 2018.

[9] C. Ericson, "Bounding Volume Hierarchies," in *Real-Time Collision Detection* [35], ch. 6, pp. 235–284.

[10] C. Ericson, "BSP Tree Hierarchies," in *Real-Time Collision Detection* [35], ch. 8, pp. 349–382.

[11] C. Ericson, "Spatial Partitioning," in *Real-Time Collision Detection* [35], ch. 7, pp. 285–348.

[12] A. A. Ahmadi, "Lecture 5: Separating hyperplane theorems, the farkas lemma, and strong duality of linear programming.." http://www.princeton.edu/~amirali/Public/Teaching/ORF523/ORF523_Lec5.pdf. ultima accesare: 18 August 2018.

[13] 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*, vol. 4, pp. 193–203, April 1988.

[14] G. van den Bergen, "A fast and robust gjk implementation for collision detection of convex objects," *J. Graph. Tools*, vol. 4, pp. 7–25, Mar. 1999.

[15] G. van den Bergen, "Proximity queries and penetration depth computation on 3d game objects," 2001.

[16] E. Catto, "Modeling and solving constraints," in *Game Developers Conference*, pp. 36–41, 2009.

[17] D. H. Eberly, "Collision response for colliding contact," in *Game Physics*, ch. 6.6, pp. 475–497, New York, NY, USA: Elsevier Science Inc., 2003.

[18] E. Catto, "Position Correction Notes." https://github.com/erincatto/Box2D/blob/master/Box2D/Dynamics/b2Island.cpp. ultima accesare: 18 August 2018.

[19] G. Fiedler, "Integration basics - how to integrate the equations of motion." https://gafferongames.com/post/integration_basics/, 2004. ultima accesare: 18 August 2018.

[20] T. Sauer, "Euler's method," [36], ch. 6.1, pp. 283–287.

[21] T. Sauer, "Backward Euler Method," [36], ch. 6.6, pp. 333–335.

[22] T. Sauer, "Runge-Kutta Methods And Applications," [36], ch. 6.4, pp. 316–317.

[23] Microsoft, "Windows Presentation Foundation." https://docs.microsoft.com/en-us/dotnet/framework/wpf/. ultima accesare: 18 August 2018.

[24] Qt Project, "Qt." https://www.qt.io/what-is-qt/. ultima accesare: 18 August 2018.

[25] Julian Smart, "wxWidgets." http://wxwidgets.org/about/. ultima accesare: 18 August 2018.

[26] Omar Cornut, "Dear ImGui." https://github.com/ocornut/imgui. ultima accesare: 18 August 2018.

[27] Micha Mettke, "Nuklear." https://github.com/vurtun/nuklear. ultima accesare: 18 August 2018.

[28] Echipa UPB-Graphics, "Framework-EGC." https://github.com/UPB-Graphics/Framework-EGC, 2016. ultima accesare: 18 August 2018.

[29] Echipa UPB-Graphics, "Framework laborator." https://ocw.cs.pub.ro/courses/egc/laboratoare/01, 2016. ultima accesare: 18 August 2018.

[30] C. Riccio, "OpenGL Mathematics." https://github.com/g-truc/glm, 2018. ultima accesare: 18 August 2018.

[31] V. Koch, "Rotations in 3d graphics and the gimbal lock." http://www.okanagan.ieee.ca/wp-content/uploads/2016/02/GimbalLockPresentationJan2016.pdf, 2016. ultima accesare: 18 August 2018.

[32] C. Ericson, "Oriented Bounding Boxes (OBBs)," in *Real-Time Collision Detection* [35], ch. 4.4, pp. 101–106.

[33] C. Muratori, "Implementing GJK - 2006." https://www.youtube.com/watch?v=Qupqu1xe7Io. ultima accesare: 18 August 2018.

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

[35] C. Ericson, *Real-Time Collision Detection*. San Francisco, CA: CRC Press, 2004.

[36] T. Sauer, *Numerical Analysis*. USA: Addison-Wesley Publishing Company, 2nd ed., 2011.

A EXTRASE DE COD

```
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;
            }
        }
    }
}
```

Extras de cod A.1: Cazul triunghi în algoritmul GJK

```
{
        "settings" : {
                "damping" : {
                        "linear" : 0.9,
                        "angular" : 0.9
                },
                "forceMultipliers" : {
                        "torque" : 1.0,
                        "force" : 1.0,
                        "impulse" : 1.0
                },
                "gjkepa" : {
                        "gjkMaxIters" : 100,
                        "epaMaxIters" : 50
                },
                "collisionResolution" : {
                        "penMaxIterations" : 5,
                        "velMaxIterations" : 5,
                        "minVelocityForRestitution" : 0.25,
                        "angularMovementLimitFactor" : 0.2,
                        "persistentContactDistanceThreshold" : 0.001,
                        "coefInterpAlpha" : 0.5
                },
                "rigidBodies" : {
                        "defaultMass" : 32.0,
                        "defaultRestitutionCoef" : 0.1,
                        "defaultFrictionCoef" : 0.6,
                        "sleepMotionThreshold" : 0.1
                },
                "gravity" : {
                        "x" : 0.0,
                        "y" : -20.0,
                        "z" : 0.0
                },
                "timeScale" : 1.0,
                "rendering" : {
                        "renderColliders" : true,
                        "renderContacts" : true,
                        "renderContactNormals" : true,
                        "renderSpawner" : true,
                        "renderSelection" : true
                }
        },
        "objects" : [
                {
                        "type" : "wall",
                        "name" : "floor",
                        "position" : {
                                "x" : 0.0,
                                "y" : -0.5,
                                "z" : 0.0
                        },
                        "scale" : {
                                "x" : 2.0,
                                "y" : 2.0,
                                "z" : 1.0
                        },
                        "orientation" : {
                                "pitch" : 90.0,
                                "yaw" : 0.0,
                                "roll" : 0.0
                        },
                        "frictionCoef" : 0.6,
                        "restitutionCoef" : 0.1,
                        "mass" : -1.0
                }
        ]
}
```

Extras de cod A.2: toate setarile disponibile pentru o scena in format .json

B DIAGRAME DE CLASE

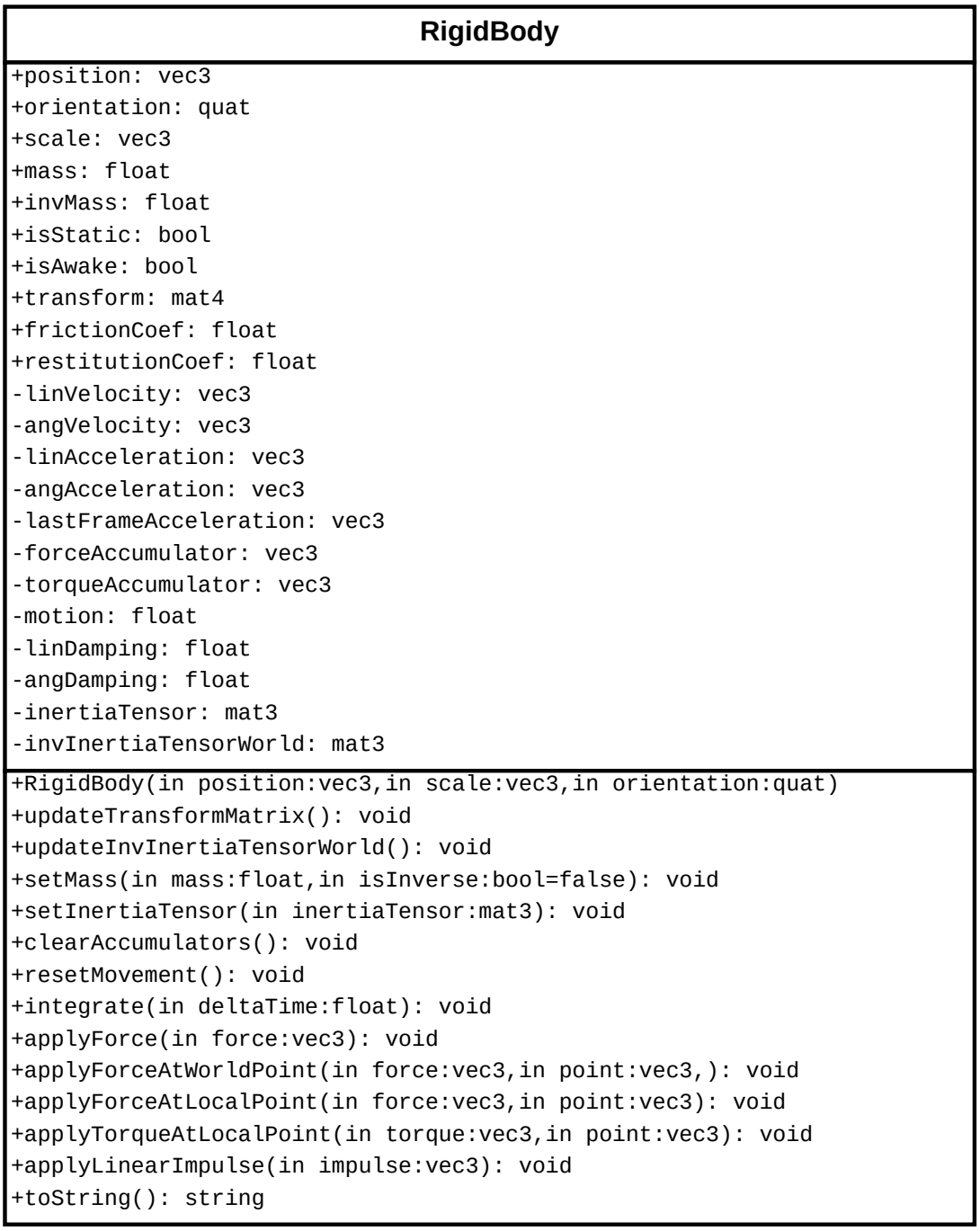


Figura B.1: Definiția unui RigidBody

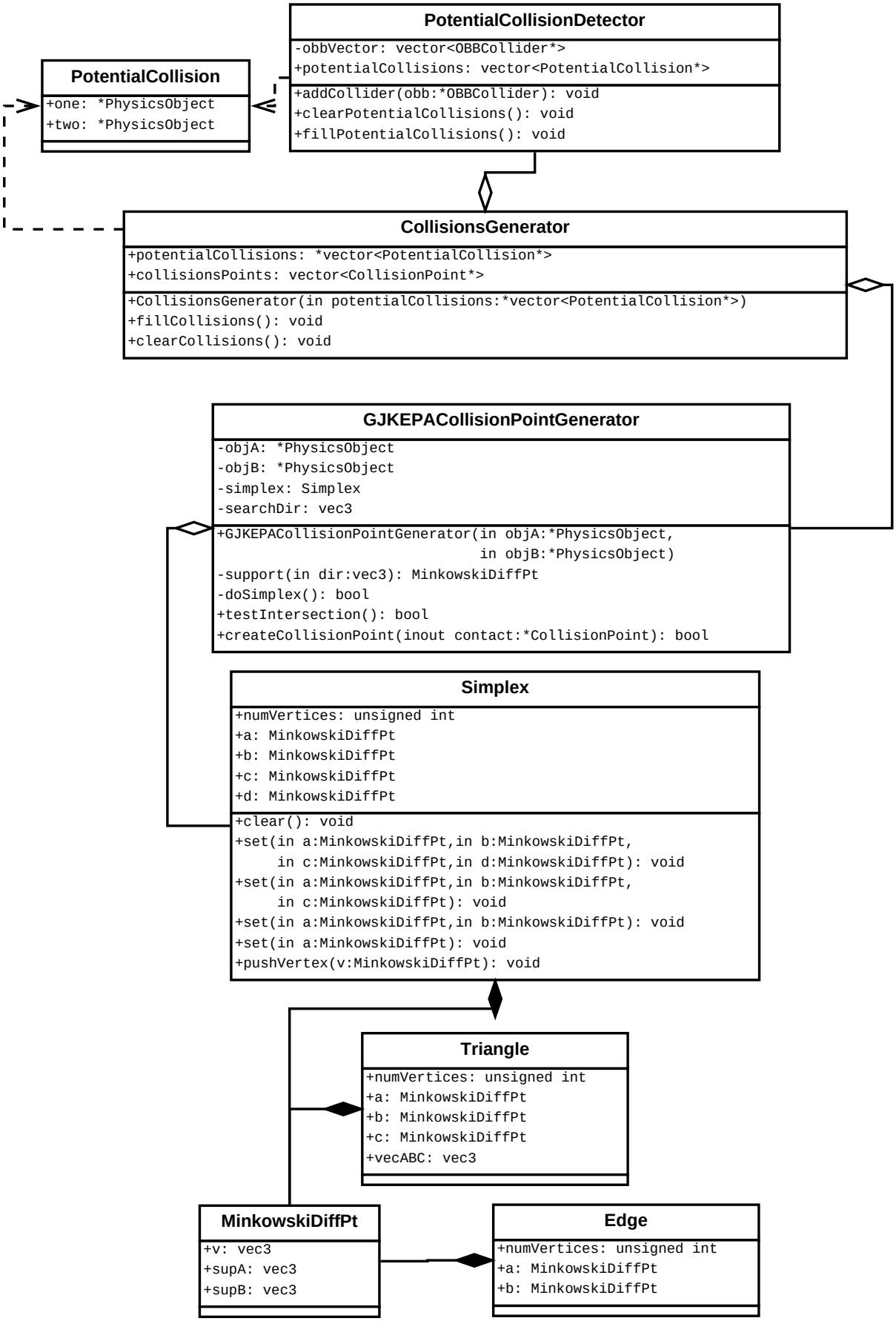


Figura B.2: Subsistemul de detecție de coliziuni

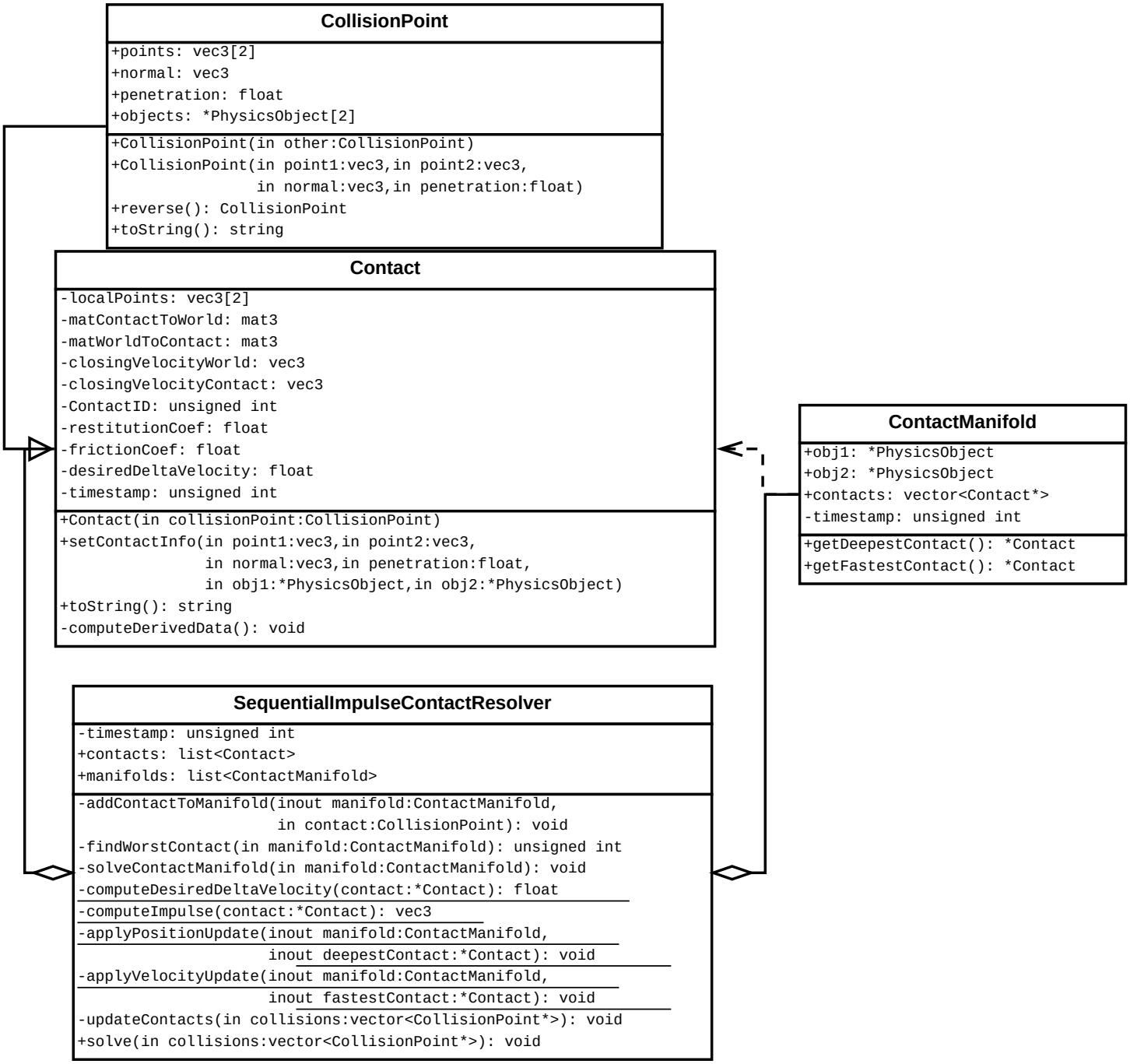


Figura B.3: Subsistemul de rezolvare a coliziunilor

C CAPTURI DE ECRAN

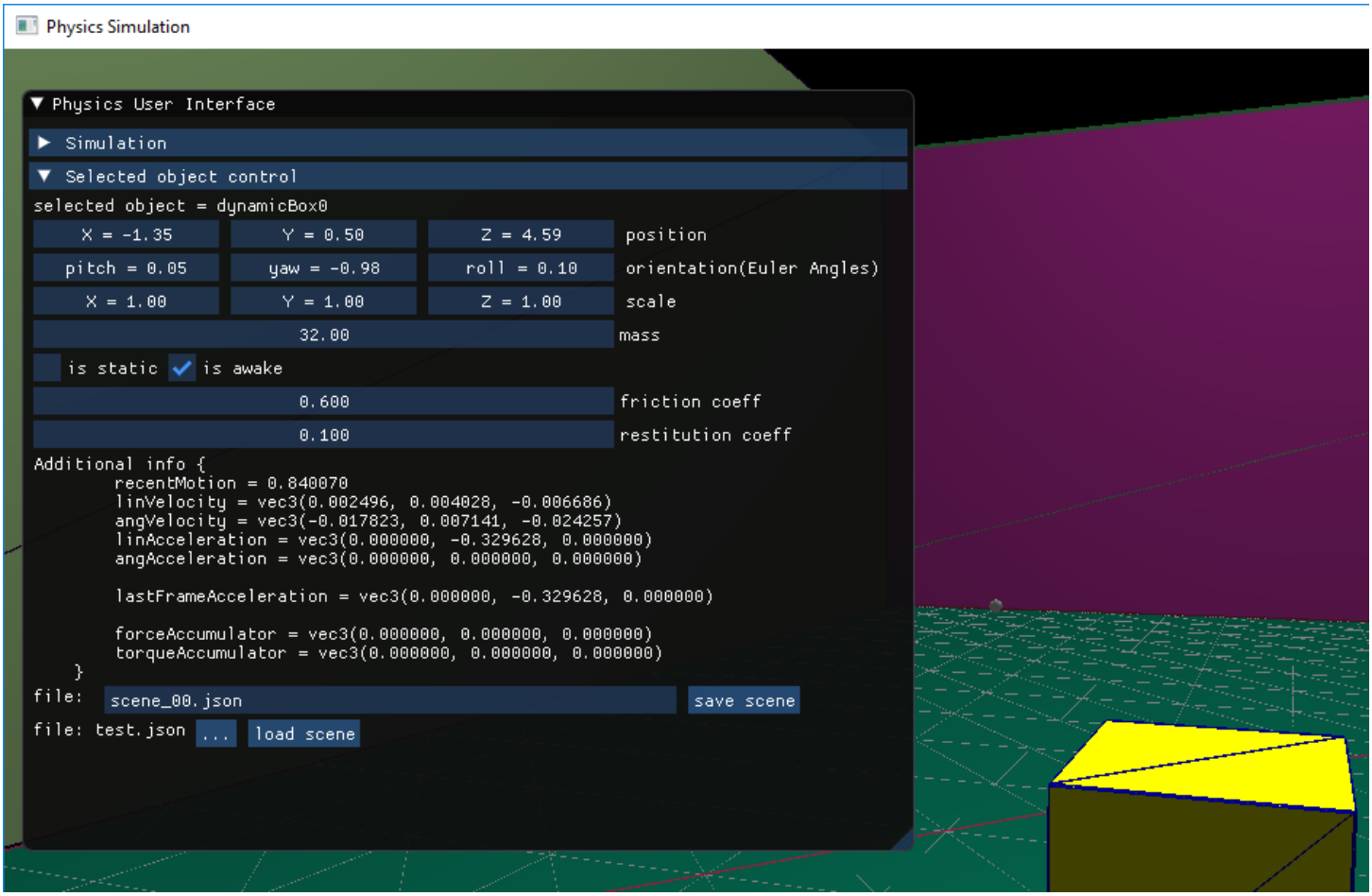


Figura C.1: Controlul obiectului selectat

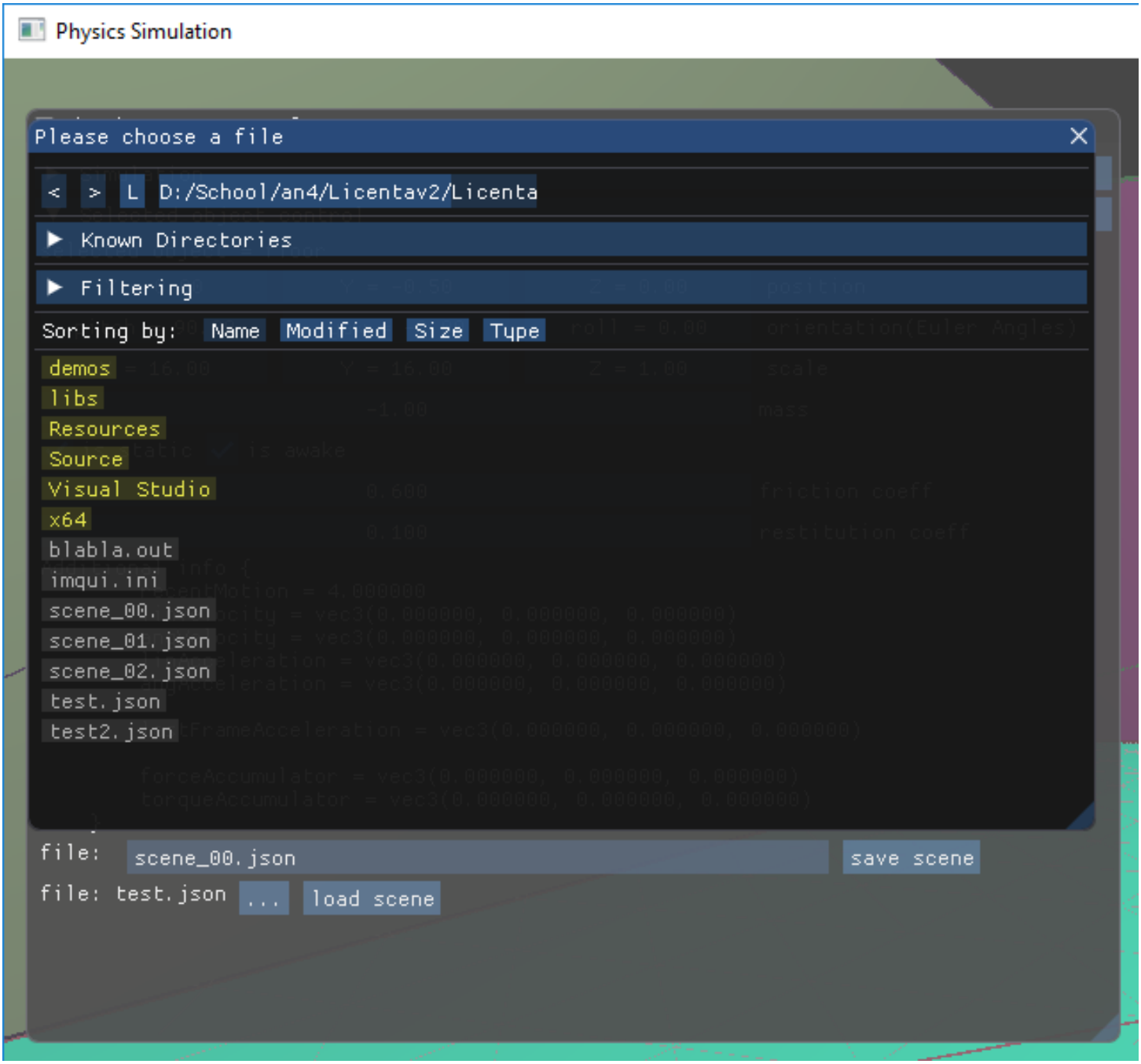


Figura C.2: Încărcarea de scene la runtime

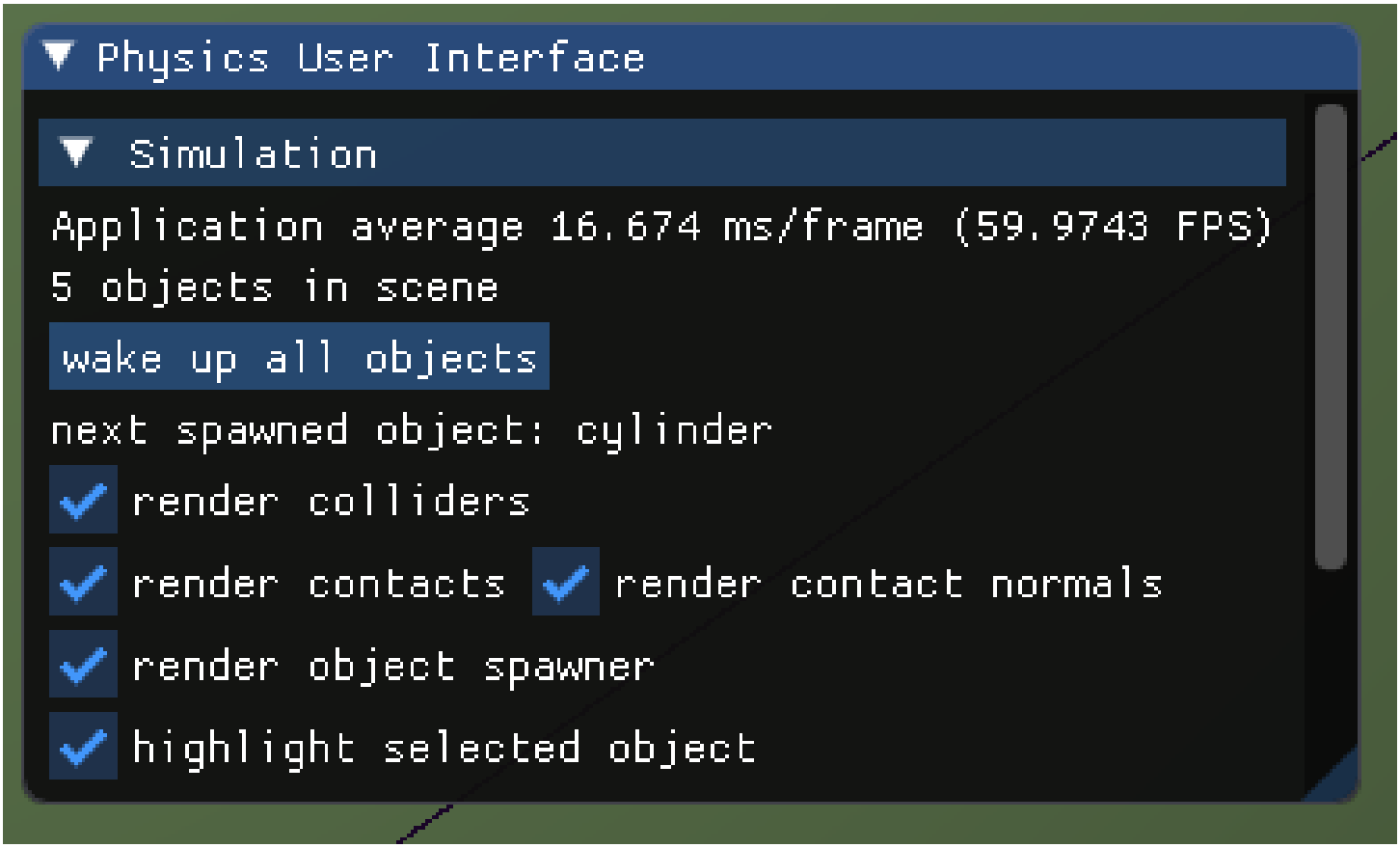


Figura C.3: Controlul elementelor desenate

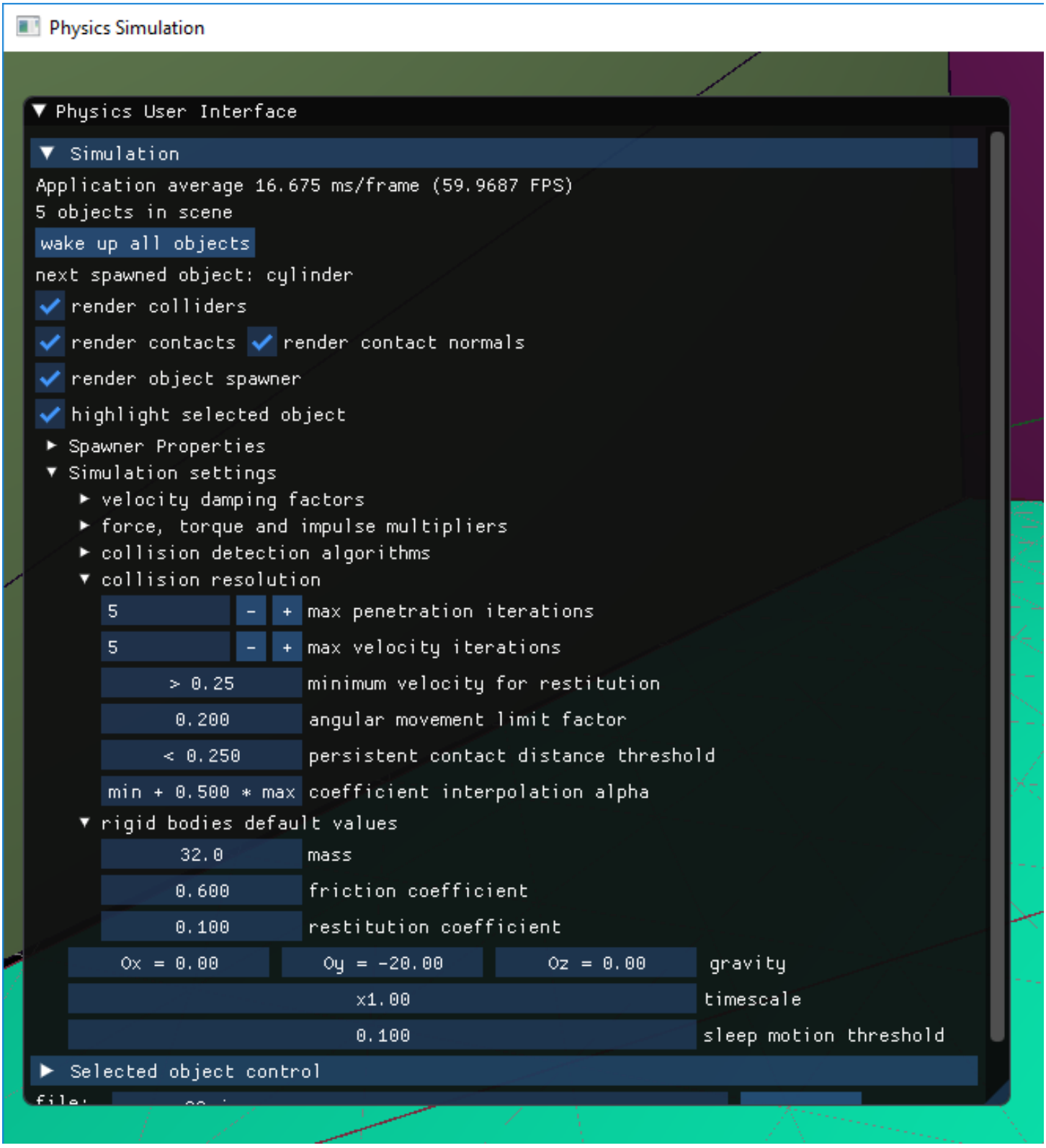


Figura C.4: Controlul parametrilor de simulare