UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

MILANO (MI)

Algoritmi nella simulazione della fisica nella computer grafica e BulletPhysics

Matricola
Lorenzo ***del Prete 885465***

Anno accademico
2023-2024

Settembre 2024
Elaborato per l'esame di Informatica Grafica

Indice

1 Introduzione alla simulazione della fisica **3**
1.1 I vantaggi di un Physics Engine 3
1.2 Tipi di engine e pipeline 4

2 I corpi rigidi **6**

3 Collision Detection **7**
3.1 Broad Phase: algoritmo Sweep and Prune 7
3.2 Narrow Phase: algoritmo Separating Axis Test 10

4 Resolution Phase **12**
4.1 Risoluzione di un vincolo di contatto lungo la normale di contatto 12

5 Metodo di Eulero semi-implicito **13**

6 La libreria BulletPhysics **14**
6.1 La pipeline dei corpi rigidi 14
6.2 Implementazione di Bullet 14
6.3 Bullet e la collision detection 15
6.4 I vincoli in Bullet 15

7 Implementazione di BulletPhysics sul codice creato a lezione **17**
7.1 Codice 1: caduta del grave 22
7.2 Codice 2: collisione con un altro corpo 22
7.3 Codice 3: pendolo semplice 22
7.4 Codice 4: pendolo smorzato 24
7.5 Codice 5: double pendulum 24
7.6 BulletPhysics e i SoftBodies 26

Elenco delle figure

1 Esempio dello sviluppo di una nuova tecnica di simulazione fisica in una produzione multimediale in *Tangled* (2010) di Walt Disney Animation Studios, in cui è stato utile per creare simulazioni verosimili del movimento dei lunghi capelli della protagonista. . . . 3
2 Esempio di una pipeline di simulazione fisica . . . 5
3 Orientazione di un corpo rigido con un asse ed un angolo . . . 6
4 La Broad Phase individua le coppie che collidono (le box rosse) e la narrow phase calcola i punti di contatto . . . 7
5 Bounding Box di una geometria complessa . . . 8
6 Esempi di AABB su alcune geometrie . . . 8
7 Esempio della creazione di una BVH. . . . 9
8 Rappresentazione grafica dell'algoritmo sweep and prune . . . 10
9 Separating Axis Test . . . 11
10 Esempio di come il metodo di Eulero permetta di risolvere numericamente la traiettoria di un pendolo in presenza di attrito e senza approssimazione di piccoli angoli ($\sin(\theta) \approx \theta$). . . . 13
11 Pipeline dei rigid bodies di BulletPhysics. . . . 14
12 Tabella degli algoritmi di collisione narrow in base alle combinazioni di shapes che collidono . . . 15
13 Vincolo punto-punto . . . 15
14 Vincolo cerniera . . . 16
15 Vincolo a scorrimento . . . 16
16 Schema del pendolo semplice. In Bullet si fissa l'anchor nell'intersezione dei due assi. Il corpo del pendolo è la mesh utilizzata. . . . 23
17 Legge oraria di un pendolo smorzato . . . 24
18 L'idea è quella di prendere una mesh e realizzarne una mesh col formalismo dei *voxel*, in questo modo si può tener conto del volume dell'oggetto e risolvere tutti i constraint tra le particelle del corpo . . . 26
19 Esempio di simulazione di un tessuto tramite la Demo di OpenGL. Essendo un soft body in 2D si nota la composizione in triangoli su tutta la superficie della mesh. Nel caso di un rigidbody un plane sarebbe stato stato creato con un array di 6 vertici. . . . 27

1 Introduzione alla simulazione della fisica

Nell'ambito della computer grafica la simulazione della fisica si ritrova in diversi ambiti. Un esempio è il ray-tracing, che fa uso di soluzioni fisiche al fine di simulare l'impatto dei raggi di luce sulla scena.
Tuttavia l'utilizzo è vasto e trova le proprie applicazioni anche nell'ambito delle produzioni multimediali e videoludiche per la simulazione di, ad esempio, corpi, particellari o tessuti.

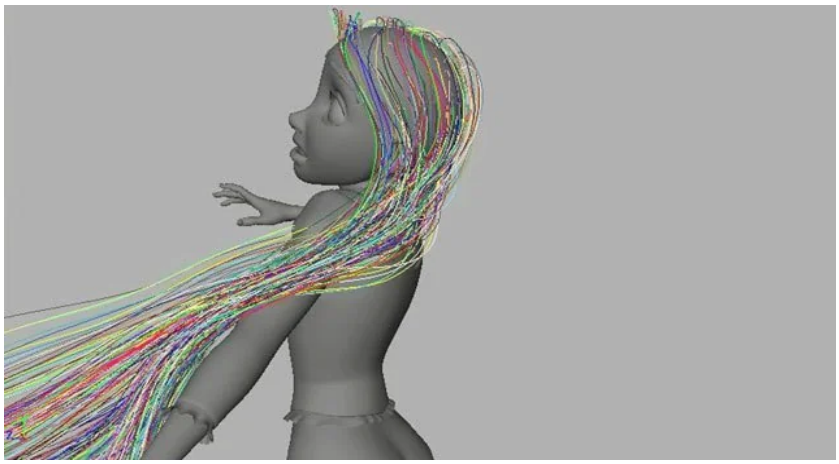

Figura 1: Esempio dello sviluppo di una nuova tecnica di simulazione fisica in una produzione multimediale in Tangled (2010) di Walt Disney Animation Studios, in cui è stato utile per creare simulazioni verosimili del movimento dei lunghi capelli della protagonista.[1]

[1] https://www.youtube.com/watch?v=-VoXmleeaaw

Per quanto complesso nell'implementazione, si tratta di un processo essenziale garantisce il realismo e l'interattività delle scene realizzate in 3D.

1.1 I vantaggi di un Physics Engine

Un Physics Engine è una centralizzazione di codice per la simulazione generica di determinati comportamenti fisici, applicabili in uno spettro più ampio di situazioni. Lo scopo pratico di un physics engine è quello, ad esempio, di simulare la traiettoria di un proiettile, classificato da alcuni parametri, a prescindere dal fatto che quest'ultimo sia un proiettile o una freccia.
Per quanto all'interno di un progetto semplice si possa utilizzare l'integrazione di alcuni meccanismi di simulazione di fisica funzionali a quell'unico progetto, come avveniva nei primi videogiochi ad integrare la fisica, con il crescere della complessità del progetto e della produzione si è iniziato a creare dei *motori fisici*[2] che potessero essere riutilizzabili e che simulassero una fisica generale, utile per più o meno tutte le produzioni.
All'interno di una produzione questo tipo di approccio permette di risparmiare tempo e risorse, reinvestibili nella simulazione di fenomeni fisici più complessi e specifici dell'opera.
Tuttavia perdita di specificità implica anche andare in contro a problemi di ottimizzazione.[3].

[2] Tra i più noti Open Source, *BulletPhysics*, *PhysX* di NVIDIA, *Tokamak*. Per quanto riguarda i motori closed source alcuni esempi sono *Havok* e il *Decima Engine*, utilizzato in *Death Stranding* di *Kojima Productions*.

[3] *Game Physics Engine Development* di Ian Millington, edito da Morgan Kaufmann Publishers

1.2 Tipi di engine e pipeline

Nella fase preliminare della creazione di un engine è necessario definirne lo scopo, motivo per cui in generale è possibile distinguere due classi di engine:

- *High-Precision Engine*, utilizzabili per dei calcoli più precisi e costosi a livello di risorse computazionali;
- *Real-Time Engine*, che fa uso di calcoli semplificati che permettono un compromesso tra accuratezza e pesantezza di calcolo.

L'elaborato tratterà principalmente la seconda tipologia, in quanto si baserà su BulletPhysics, un physics engine open source per la realizzazione di fisica real-time. Non considerando al momento il legame con il framework che permette di realizzare la scena 3D [1], un physics engine sviluppa la propria simulazione in una pipeline che parte dall'avere N corpi (o *bodies*) in un mondo fisico e si prefissa di ottenere la traiettoria dei corpi in seguito a delle interazioni. Pertanto la generica pipeline di un motore fisico è la seguente:

1. **Rappresentazione degli oggetti** ovvero i bodies sui quali si agisce come vengono rappresentati? Solitamente la distinzione avviene tra:
   - *Rigid body*, un corpo che non si deforma sotto l'azione delle forze, ovvero che mantiene costanti le distanze tra i punti del corpo;
   - *Soft body*, un corpo rappresentato da una mesh complessa di punti e che può dunque simulare deformazioni e forze interne tra i punti.
2. **Collision Detection** lo step della pipeline nel quale si calcola quali oggetti del dynamic world collidono a partire dai valori di partenza. Viene effettuata in due fasi:
   - *Broad Phase* una fase in cui vengono calcolate in modo più approssimato le coppie di oggetti[2] (*collision pairs*) che sicuramente non collideranno.
   - *Narrow Phase* la fase in cui venogono effettuati calcoli più accurati tra le coppie sopravvissute alla Broad Phase. Qui si prendono in considerazione le forme degli oggetti e si calcolano i punti di contatto tra gli oggetti che collidono.
3. **Resolution Phase** una volta considerata la collisione, si calcola le nuove direzioni e le nuove velocità degli oggetti. Per fare ciò si utilizzano due approcci diversi
   - *Impulse Based* in cui l'interazione viene considerata come un impulso istantaneo al momento dell'impatto, ovvero come un evento isolato.
     Largamente utilizzato per collisioni singole, in quanto facilmente risolvibili e isolabili.
   - *Constraint Based* si impongono dei vincoli (*constraint*) tra gli oggetti e l'interazione viene considerata come la risoluzione dei vincoli, dunque come interazione tra forze che vengono applicate in modo continuativo.
     utilizzato per la simulazione di interazioni più complesse come quella del pendolo che è costantemente sottoposto al vincolo del filo, tuttavia rende la risoluzione più complicata e pesante.

[1] OpenGL nel nostro caso

[2] Il termine *body* ed *object* sono interscambiabili

4. **Integrazione temporale** una volta calcolate le nuove direzioni e velocità degli oggetti serve calcolarne l'evoluzione temporale nel tempo allo scopo di creare la traiettoria, aggiornando frame by frame gli stati. Per fare ciò si utilizzano algoritmi di risoluzione di equazioni differenziali tra cui il metodo di *Eulero* o i metodi di *Runge-Kutta*.

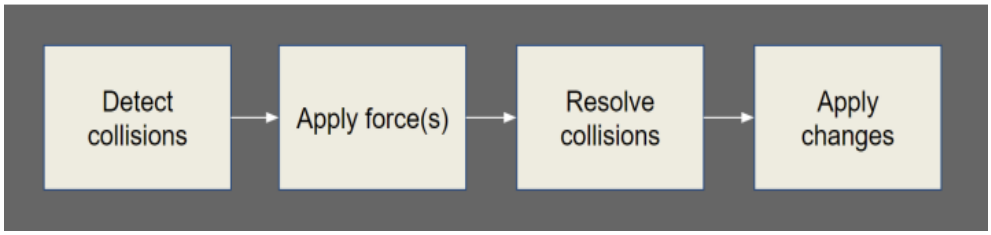

Figura 2: Esempio di una pipeline di simulazione fisica

2 I corpi rigidi

Il formalismo del rigid body è comodo per definire un oggetto su cui è possibile calcolare delle collisioni e che sia in grado, oltre che di traslare, anche di effettuare delle rotazioni rigide. Fisicamente parlando un corpo rigido è un corpo nel quale le distanze tra i punti rimangono fisse nelle interazioni, ovvero un corpo *indeformabile*. L'indeformabilità del corpo permette di escludere tutta una serie di fenomeni con un costo computazionale elevato, ad esempio l'effetto delle forze esterne nella relazione tra i punti del corpo.

Per definire un rigid body serve fissare un origine del corpo. Per effettuare i calcoli sul rigid body viene comodo utilizzare come origine il *centro di massa* dell'oggetto, definito come

$$\vec{x_{com}} = \frac{\sum_i^N m_i \cdot \vec{x_i}}{\sum_i^N m_i} \tag{1}$$

dove m_i è la massa del punto i-esimo, N è il numero di punti con cui voglio approssimare il corpo rigido e $\vec{x_i}$ è la posizione del punto i-esimo. Si tratta di una media pesata dei punti rispetto alla loro massa. La comodità nell'utilizzare il centro di massa come origine è che per ogni corpo rigido il suo moto può essere scomposto nel moto traslazionale del punto in questione e del moto di rotazione attorno al punto. In questo modo ogni corpo rigido ha un numero di *gradi di libertà* pari a 3 traslazionali + 3 rotazionali.

Per l'orientazione del corpo si usa la rappresentazione *axis-angle*, ovvero con la definizione di un asse rappresentato da un vettore e un angolo rispetto all'asse.[3]

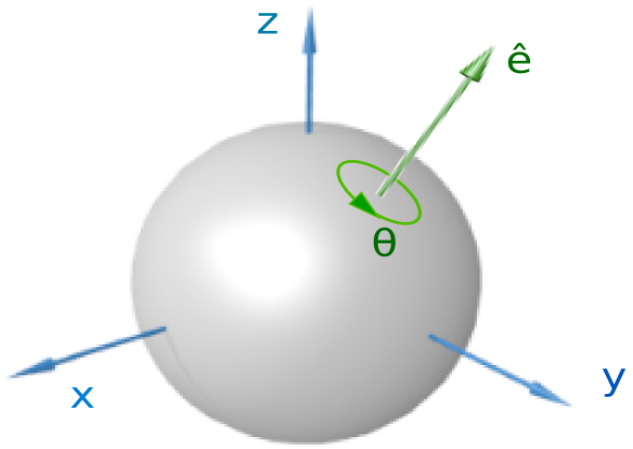

Figura 3: Orientazione di un corpo rigido con un asse ed un angolo

Oltre il centro di massa che considera il corpo rigido come un corpo puntuale con tutta la massa dei punti e con una velocità del centro di massa, per trattare le rotazioni serve anche definire la velocità angolare del corpo e il momento d'inerzia.

[3] https://en.wikipedia.org/wiki/Euler%27s_rotation_theorem

3 Collision Detection

Una volta configurate i parametri iniziali dei corpi ovvero le loro posizioni e velocità iniziali, bisogna effettuare un check sulle collisioni. Si tratta di un calcolo che se fatto senza una giusta computazione diviene dispendioso dal punto di vista computazionale, infatti volendo stimare senza un algoritmo efficiente con un Brute Force le *collision pairs*, ovvero le coppie di oggetti in collisione, si avrebbe una complessità che cresce come n^2. Volendo depositare le coppie in una lista Collision Pairs a partire da una lista Body dei corpi nel mondo si otterrebbe

```
for (ogni corpo in Body) do
    for (ogni altro corpo in Body) do
        if (collidono) then
            aggiungi i bodies alla lista Collision Pairs
```

come è chiaro, in questo modo l'algoritmo diviene inefficiente con l'aumentare dei corpi. Allo scopo di rendere efficiente il calcolo, il check delle collisioni avviene in due step differenti.

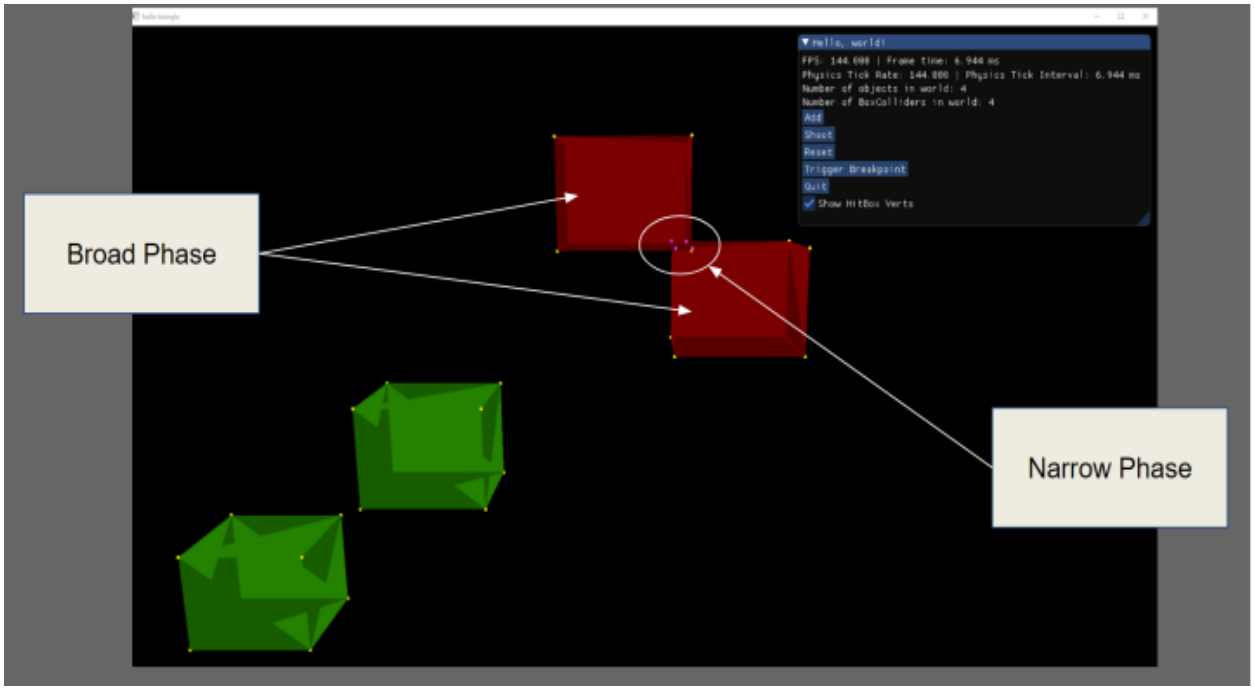

Figura 4: La Broad Phase individua le coppie che collidono (le box rosse) e la narrow phase calcola i punti di contatto

3.1 Broad Phase: algoritmo Sweep and Prune

La Broad Phase è una prima fase in cui si fanno dei check preliminari e approssimati per escludere le coppie di corpi che non collideranno in base alla loro distanza relativa.
Per poter applicare un algoritmo di *sweep and prune* serve un passaggio preliminare, ovvero

- Definire il *bounding volume*, ovvero un volume geometrico che contiene interamente l'oggetto in questione in una forma cubica o sferica, in modo tale da approssimarne il volume e testarne l'overlap con altri oggetti più facilmente.

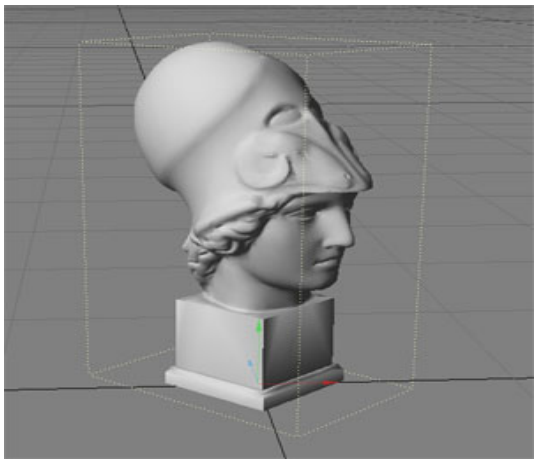

Figura 5: Bounding Box di una geometria complessa

Solitamente si utilizzano delle forme box per contenere l'oggetto, dette AABB o *Axis-aligned minimum bounding box*[4], ovvero una bounding box soggetta al vincolo di avere i lati paralleli agli assi cartesiani di riferimento.

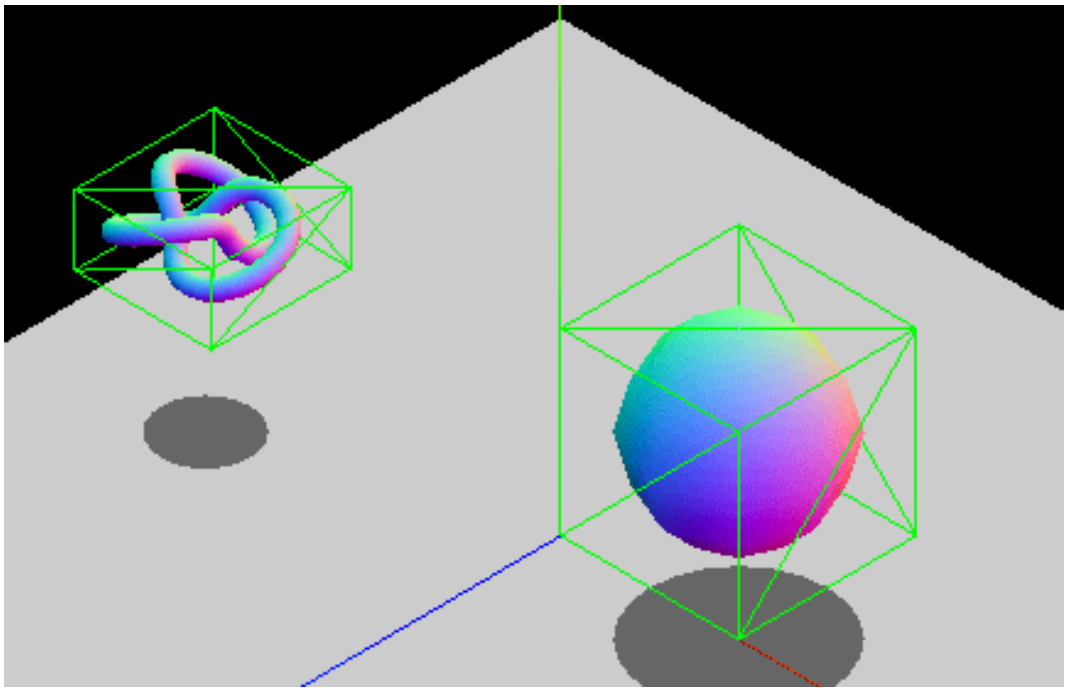

Figura 6: Esempi di AABB su alcune geometrie

Una volta definite le AABB degli oggetti, in cui ogni AABB è caratterizzata dunque da tre paia di valori, ovvero $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$, si procede con effettuare dei check sui singoli assi, ovvero la fase di *sweeping*. Iniziando dall'asse x, se due AABB sono in overlap, ovvero se $x_{min,A} \leq x_{max,B}$ e $x_{max,A} \geq x_{min,B}$, allora si effettua un check sugli altri due assi. Se le condizioni non sono soddisfatte su ogni asse allora la coppia non viene presa in considerazione, ovvero avviene il *prune*.
In definitiva definita una lista di AABBs e volendo creare una lista di collision pairs, l'algoritmo di sweep and prune consiste in

[4]In ambito videoludico sono spesso chiamate *hitboxes*

```
for (AABB_A nella lista AABBs) do
    for (AABB_B nella lista ABBS) do
        if (x_{min,A} < x_{max,B} e x_{max,A} > x_{min,B}) then
            aggiungi i due corpi nella lista Collision Pairs

for (ogni Collision Pair ) do
    if (y_{min,A} < y_{max,B} e y_{max,A} > y_{min,B}) then
        mantieni i due corpi nella lista Collision Pairs
    else
        rimuovili

for (ogni Collision Pair ) do
    if (z_{min,A} < z_{max,B} e z_{max,A} > z_{min,B}) then
        mantieni i due corpi nella lista Collision Pairs
    else
        rimuovili
```

Si tratta di un rimaneggiamento del brute force per renderlo più efficiente. Un metodo più efficiente sarebbe quello di creare una BVH *bounding volume hierarchy*, ovvero utilizzare una struttura ad albero, dividendo ad esempio il mondo in 8 ottanti e creando un albero ottale.

BVH e OpenGL il metodo BVH, per quanto più efficiente, non è di facile implementazione con OpenGL, in quanto GLSL *OpenGL Shading Language* non ammette l'uso di puntatori e di operazioni ricorsive[5].

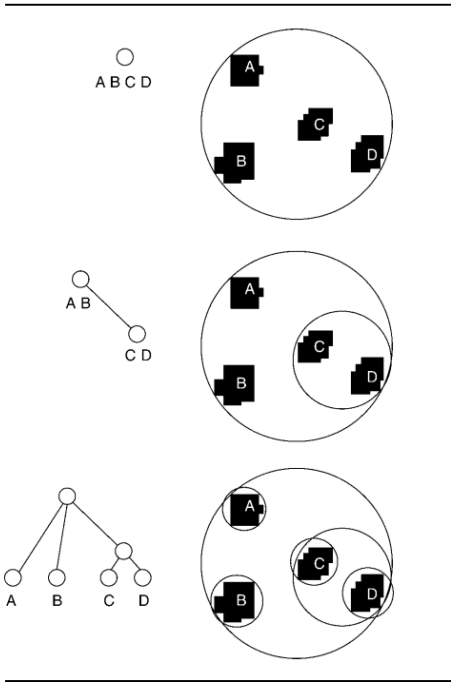

Figura 7: Esempio della creazione di una BVH.

[5] *Physics Engine on the GPU with OpenGL Compute Shaders,*
https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=3807context=theses

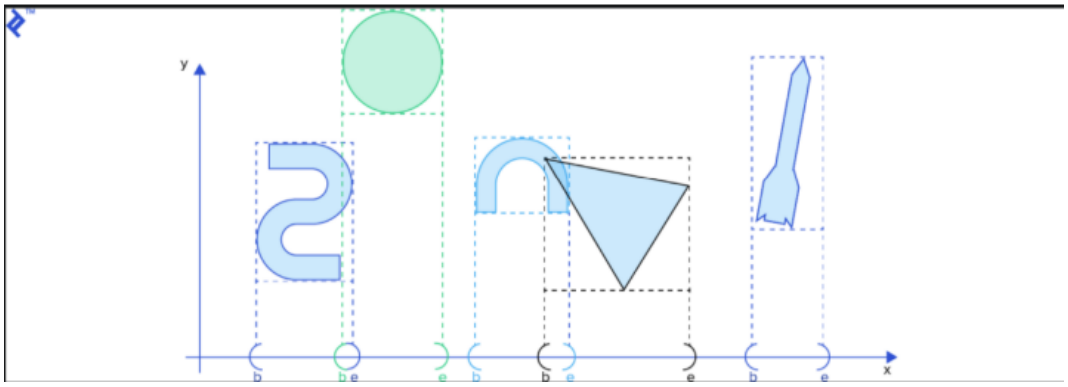

Figura 8: Rappresentazione grafica dell'algoritmo sweep and prune

3.2 Narrow Phase: algoritmo Separating Axis Test

La Narrow Phase è più complicata da implementare, in quanto deve tenere conto delle geometrie specifiche dei corpi.
Il modo più semplice per implementarlo è considerare dei cubi, sebbene nei physics engine venga implementato per ogni tipo di geometria convessa.

Nel caso dei cubi possono collidere solo le facce e gli edge. Per implementare una narrow phase è necessario creare una geometria di collisione che sia stabile, pertanto se la collisione riguarda una faccia, serve computare 4 punti di contatto; se a collidere è un edge, basta 1 solo punto di contatto. Prima di implementare l'algoritmo Separating Axis Test (SAT) dunque bisogna capire se sono edge o facce ad interagire. Dopodiché si effettua una query sulla faccia o l'edge in questione. Supponiamo di star trattando delle facce.
Trovate le facce in questione si procede a definire una faccia di *reference* e una *incidente*. Il passo successivo è effettuare un clipping della faccia incidente rispetto alle direzioni delle facce adiacenti della reference box. Per farlo si può utilizzare l'algoritmo di *Sutherman-Hodgeman*[6].
Dal clipping si mantengono solo i punti di intersezione e i punti interni al piano.
Si ottengono così una collezione di punti di contatto con cui creare una geometria di contatto.

In definitiva il SAT parte dal risultato che se esiste un asse che separa due box, allora non collidono. Se non viene trovato un asse separatore, allora i due box collidono.

Un algoritmo più raffinato è quello di di *Gilbert-Johnson-Keerthi* GJK.

[6]Un maggiore approfondimento in
https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=3807context=theses

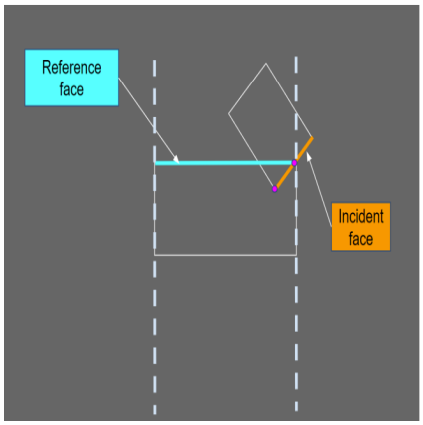

Figura 9: Separating Axis Test

4 Resolution Phase

Una volta trattata la collisione e ottenuti i punti di contatto, bisogna calcolare le direzioni e le velocità degli oggetti che hanno colliso, ovvero risolvere la collisione. Una collisione si considera risolta nel momento in cui non c'è più sovrapposizione tra i corpi e la forza di interazione nella collisione è stata simulata. Alla fine della fase di risoluzione si vuole sapere dove il corpo sta andando e come sta ruotando.

4.1 Risoluzione di un vincolo di contatto lungo la normale di contatto

Definito un constraint, ovvero un vincolo, C come $f(\vec{x_B},\vec{x_A})$ nel nostro caso, ovvero come funzione delle posizioni dei due corpi, $C : (\vec{x_B} - \vec{x_A}) \cdot \vec{n} \geq 0$. Ciò significa richiedere che le posizione relative dei due corpi debbano essere positive o al più nulle. C è detto *vincolo di contatto* e si dice che un vincolo è soddisfatto quando C = 0. Se il vincolo è violato, lo si soddisfa applicando un cambiamento nella velocità[7]. ΔV. Per farlo si decompone il moto nel moto del centro di massa e nel moto relativo tra i due corpi.

$$(\vec{CoM_B} - \vec{CoM_A} + \vec{r_B} - \vec{r_A}) \cdot \hat{n} \geq 0 \tag{2}$$

CoM sono i vettori relativi al centro di massa, mentre gli r sono i vettori delle posizioni relative dei punti di contatto rispetto al centro di massa del corpo. Derivando il vincolo si ottiene

$$(-\vec{V_A} - \vec{\omega_A} \times \vec{r_A} + \vec{V_B} + \vec{\omega_B} \times \vec{r_B}) \cdot \hat{n} \geq 0 \tag{3}$$

L'equazione può essere ricondotta a una forma matriciale $JV + b = 0$, con V matrice delle velocità. In questa forma si ha che $J(V + \Delta V) + b = 0$. Utilizzando il metodo matematico del moltiplicatore di Lagrange si può risolvere rispetto ad un parametro λ. Si tratta di un procedimento che richiede la risoluzione di un sistema di equazioni. Utilizzare un approssimazione ad impulsi, ovvero andando ad isolare il fenomeno fisico nell'interazione della forza in istanti di tempo, significa comunque effettuare diverse iterazioni. Pertanto utilizzare impulsi significa andare a lavorare linearmente sulle velocità, piuttosto che risolvere direttamente il vincolo.

L'ottimizzazione della soluzione con OpenGL Normalmente la pipeline del motore di fisica avverrebbe nella CPU. Utilizzare OpenGL permette di comunicare con la GPU e dunque di parallelizzare su un numero maggiore di threads. Se ciò è vero per le fasi di broad e narrow nella collision detection, ciò non può avvenire per la fase di risoluzione, dove gli impulsi sono sequenziali e dunque non parallelizzabili.

[7] si potrebbe effettuare una traslazione Δx ma renderebbe il movimento poco fluido

5 Metodo di Eulero semi-implicito

Ricavare le traiettorie nel tempo degli oggetti in seguito ad una collisione o rispetto ad un vincolo significa risolvere una o più equazioni differenziali. Vi sono diversi metodi per risolvere in modo iterativo delle equazioni differenziali andando a discretizzare le derivate. BulletPhysics fa uso di un metodo chiamato *Metodo di Eulero Semi-implicito*, che rispetto al semplice metodo di Eulero offre una maggiore stabilità numerica.
Si parte dall'assunto che la traiettoria dei punti devono seguire la legge di Newton $F = ma$ e che trovare la traiettoria significa integrare l'equazione di Newton del corpo.
Definito un passo temporale Δt si ottiene che

$$v = v_{precedente} + a \cdot \Delta t \tag{4}$$

$$x = x_{precedente} + v \cdot \Delta t \tag{5}$$

tale metodo aggiorna prima la velocità utilizzando la forza, dopodiché aggiorna la posizione utilizzando la velocità appena calcolata.

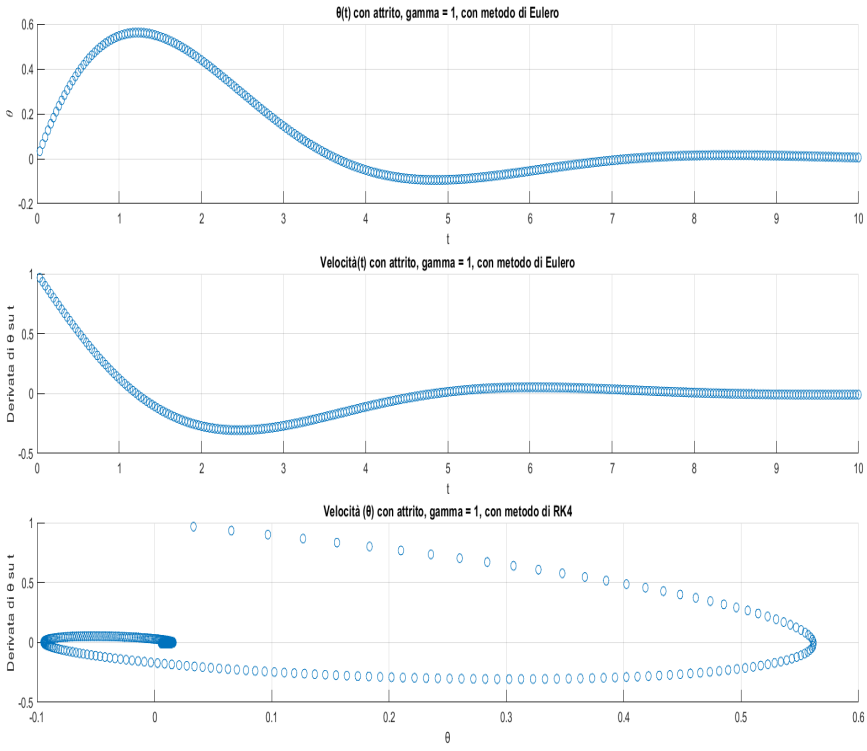

Figura 10: Esempio di come il metodo di Eulero permetta di risolvere numericamente la traiettoria di un pendolo in presenza di attrito e senza approssimazione di piccoli angoli ($\sin(\theta) \approx \theta$).

6 La libreria BulletPhysics

BulletPhysics è un motore fisico open source che funziona nella simulazione sia di corpi rigidi che di corpi morbidi. E' ampiamente supportato dalla maggior parte dei programmi di modellazione 3D tramite plug-in come Maya, Blender o Houdini.

Si tratta di una libreria ampiamente modulabile e ad alto linguaggio.

6.1 La pipeline dei corpi rigidi

Bullet utilizza una pipeline per la simulazione dei corpi rigidi che parte dall'applicare la gravità fino ad integrare le posizioni, in uno schema che riflette gli step trattati in precedenza.
L'intera pipeline e le sue strutture fanno parte di un *dynamic world*, implementabile tramite *btDiscreteDynamicsWorld*. L'aggettivo Discrete è sinonimo del fatto che la pipeline viene effettuata in uno *stepSimulation*.

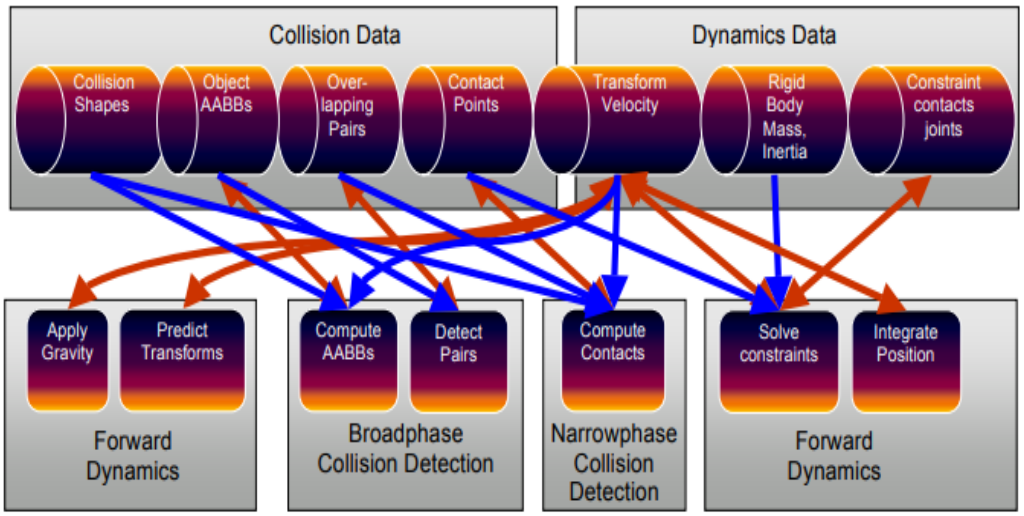

Figura 11: Pipeline dei rigid bodies di BulletPhysics.

6.2 Implementazione di Bullet

L'implementazione di Bullet in un codice avviene in tali step

- Creazione di un dynamics worlds tramite *btDiscreteDynamicsWorld*, che provvede ad un interfaccia di alto livello che permette di maneggiare gli oggetti fisici e i vincoli, implementando anche l'aggiornamento degli oggetti step-by-step;
- Creazione di un rigid body *btRigidBody* e l'inserimento del DynamicsWorld, definendone la massa, la *CollisionShape* e proprietà del materiale (attrito, coefficiente di restituzione[8]).
- Aggiornamento della simulazione ad ogni frame con la funzione *stepSimulation*

[8]Il coefficiente di restituzione è definito come il rapporto tra la differenza di velocità di due entità prima e dopo una collisione

6.3 Bullet e la collision detection

Bullet permette di decidere con quale metodo effettuare le varie fasi della collision detection senza doverle implementare manualmente. Permette di utilizzare un BVH basato su una struttura ad albero di AABB, un algoritmo di sweep and prune o il brute force.
Per la Narrow phase utilizza degli algoritmi differenti in base alla natura della collisione e alle shapes dei due oggetti che collidono, tra cui l'algoritmo GJK e il SAT.

	box	sphere	convex,cylinder cone,capsule	compound	triangle mesh
box	boxbox	spherebox	gjk	compound	concaveconvex
sphere	spherebox	spheresphere	gjk	compound	concaveconvex
convex, cylinder, cone, capsule	gjk	gjk	gjk or SAT	compound	concaveconvex
compound	compound	compound	compound	compound	compound
triangle mesh	concaveconvex	concaveconvex	concaveconvex	compound	gimpact

Figura 12: Tabella degli algoritmi di collisione narrow in base alle combinazioni di shapes che collidono

6.4 I vincoli in Bullet

Bullet permette la risoluzione di diversi tipi di vincolo, tra cui

- **Vincolo punto-punto**, che limita la traslazione di due corpi rigidi attaccati in una direzione (catena di corpi rigidi)

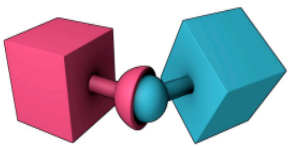

Figura 13: Vincolo punto-punto

- **Vincolo cerniera**, che limita due gradi di libertà rotazionali del corpo, permettendogli di ruotare solamente lungo l'asse della cerniera (porte, finestre...)

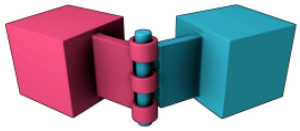

Figura 14: Vincolo cerniera

- **Vincolo a scorrimento**, permette la rotazione lungo un asse e la traslazione lungo lo stesso asse.

Figura 15: Vincolo a scorrimento

7 Implementazione di BulletPhysics sul codice creato a lezione

Partendo dal codice numero 14 delle lezioni del corso, si è implementata la fisica tramite BulletPhysics sulla mesh cubica creata con OpenGL, applicandovi la pipeline di fisica e la si è declinata in alcune casistiche fisiche.
All'interno della cartella zippata vi saranno dei video dimostrativi di ogni applicazione.

L'implementazione della pipeline e dei corpi rigidi BulletPhysics effettua le operazioni di simulazione sugli oggetti del dynamic world i quali non vengono a priori renderizzati dallo stesso Bullet, per quanto esistano delle librerie interne che permettono un render dal dynamic world tramite OpenGL. Volendo tuttavia utilizzare la mesh costruita con OpenGL, l'idea di base è stata quella di impacchettare le trasformazioni fisiche di ogni step nella matrice di trasformazione ed effettuare una sincronizzazione con la matrice di trasformazione sulla mesh, in questo modo, per quanto in verità ciò implichi effettuare due volte le trasformazioni, una volta sul dynamic world e una volta sulla mesh, permette agilmente di visualizzare sulla mesh la simulazione.
Le operazioni di Bullet vengono integrate tramite un unica libreria che gestisce le dipendenze con le differenti librerie BT.

```
#include "btBulletDynamicsCommon.h"
```

Dopodiché si è creata una struct chiamata Physics che centralizzasse la configurazione del DynamicsWorld.

```
    struct Physics {
  btDiscreteDynamicsWorld* dynamicsWorld; //andiamo a creare il mondo
   ↪  dinamico
  //per la realizzazione della collision detection phase
  btBroadphaseInterface* broadphase;
  btDefaultCollisionConfiguration* collisionConfiguration;
  btCollisionDispatcher* dispatcher;
  //per risolvere l'urto
  btSequentialImpulseConstraintSolver* solver;

  //costruttore
  Physics() : dynamicsWorld(nullptr), broadphase(nullptr),
   ↪  collisionConfiguration(nullptr), dispatcher(nullptr),
   ↪  solver(nullptr) {}

  //distruttore
  ~Physics() {
        delete dynamicsWorld;
        delete solver;
        delete dispatcher;
        delete collisionConfiguration;
        delete broadphase;
  }
}
```

A questo punto si è definita una funzione che inizializzasse la struttura fisica, ovvero *init_physics()*

```
    void init_physics(){
  //inizializziamo i campi della struct Physics
  physics.broadphase = new btDbvtBroadphase(); //il metodo Dbvt configura
   ↪  la broadphase usando AABB con HBV (hierarchy bounding volume)
  physics.collisionConfiguration  = new
   ↪  btDefaultCollisionConfiguration();
  physics.dispatcher = new
   ↪  btCollisionDispatcher(physics.collisionConfiguration); //gestione
   ↪  delle collisioni a partire da CollisionConfiguration che gestisce
   ↪  l'allocazione della memoria nella collisione
  physics.solver = new btSequentialImpulseConstraintSolver; //il
   ↪  risolutore della collisione fa uso dell'approccio ad impulsi
   ↪  sequenziali
  physics.dynamicsWorld = new btDiscreteDynamicsWorld(physics.dispatcher,
   ↪  physics.broadphase,
    physics.solver, physics.collisionConfiguration); //creazione del
     ↪  mondo dinamico



  physics.dynamicsWorld->setGravity(btVector3(0,global.g, 0));
   ↪  //impostiamo la gravità del mondo con un vettore tridimensionale
   ↪  con ogni valore del vettore l'accelerazione nella direzione
   ↪  corrispondente

}
```

Configurato il DynamicsWorlds e le varie fasi della pipeline, si può iniziare a creare i corpi rigidi

```
    //Creazione di una funzione per creare un corpo rigido
btRigidBody* createRB(float massa, const btVector3& pos,
 ↪  btCollisionShape* forma) {
  btTransform startTransform;
  // settiamo il corpo all'origine del mondo
  startTransform.setIdentity();
  startTransform.setOrigin(pos);

  btVector3 Inertia(0,0,0);
  if (massa != 0.f) //la condizione di massa uguale a zero corrisponde ad
   ↪  un body detto "static", ad esempio il pavimento della scena o un
   ↪  muro.
  {
    forma->calculateLocalInertia(massa, Inertia);
  }

  btDefaultMotionState* motionState = new
   ↪  btDefaultMotionState(startTransform); // lo stato del moto iniziale
   ↪  è quello di startTransform
  btRigidBody::btRigidBodyConstructionInfo Info(massa, motionState,
   ↪  forma, Inertia); //il motionState servirà a sincronizzare l'oggetto
   ↪  fisico con quello grafico (la mesh)
  btRigidBody* body = new btRigidBody(Info); //crea il corpo rigido a
   ↪  partire dalle informazioni

  physics.dynamicsWorld->addRigidBody(body); //aggiunge il nuovo corpo
   ↪  rigido al mondo dinamico

  return body;
}
```

Il primo corpo rigido da creare è proprio il body che corrisponderà alla mesh

```
btRigidBody* cubeRB = nullptr; //memorizziamo un puntatore al cubo che
 ↪  verrà creato

    void create_physics_cube() {
  btCollisionShape* cubeShape = new btBoxShape(btVector3(1,1,1));

  float massaCube = 1.0;


  cubeRB = createRB(1.0f, btVector3(0,0,0), cubeShape); //leghiamo il
   ↪  puntatore al corpo rigido
}
```

Si definiscono le funzioni che aggiornano la fisica, ovvero che compiono lo *stepSimulation*

e che aggiornano la visualizzazione della scena, in quanto si vuole creare un'animazione dell'evoluzione temporale del corpo. Si definisce anche la funzione che si occupa di effettuare il sync tra corpo rigido (cubeRB) e mesh.

```
    / Funzione per aggiornare il mondo fisico e farlo avanzare di uno
     ↪ StepSimulation
void update_phys(){
  physics.dynamicsWorld->stepSimulation(global.deltat, 10); //facciamo 10
   ↪ substeps
}

// Funzione che aggiorna continuamente la scena per creare l'animazione
void update(int value){
  update_phys();

  glutPostRedisplay();

  glutTimerFunc(16, update, 0);
}

// Funzione per sincronizzare il corpo fisico con il modello
void sync_btgl(){
  btTransform trans;
  cubeRB->getMotionState()->getWorldTransform(trans); //grazie al
   ↪ motionstate riusciamo a recuperare la trasformazione come matrice
   ↪ 4x4

  float matrix[16];
  trans.getOpenGLMatrix(matrix); //formattiamo la matrice per openGL

  //creiamo una matrice di GLM con la matrice appena creata.
  glm::mat4 trans_modello = glm::mat4(
            matrix[0],  matrix[1],  matrix[2],  matrix[3],
            matrix[4],  matrix[5],  matrix[6],  matrix[7],
            matrix[8],  matrix[9],  matrix[10], matrix[11],
            matrix[12], matrix[13], matrix[14], matrix[15]
        );

  global.shaders.set_model_transform(trans_modello); //applichiamo la
   ↪ matrice al corpo

}
```

In definitiva si chiama *init_physics* nella funzione init, la funzione *create_physics_cube* (e le eventuali funzioni di creazione di corpi rigidi) in create_scene, e le funzioni *update_phys()*, *sync_btgl()* nella funzione MyRenderScene(). Si rimuovono i corpi rigidi dal DynamicsWorld e i campi della struct Physics in MyClose per la gestione della memoria.

7.1 Codice 1: caduta del grave

Il primo codice simula l'azione della gravità sul cubo. Si tratta del codice descritto sopra, senza ulteriori aggiunte. Poiché non si è definito uno static plane, il corpo cade fino ad uscire dalla regione di clipping.

7.2 Codice 2: collisione con un altro corpo

Il secondo codice invece utilizza un corpo rigido *statico*, ovvero virtualmente "di massa infinita", che possa simulare un muro o un pavimento.
In questo caso si crea il corpo statico nella funzione create_physics_static(), si utilizza una box posta in (0, -5, 0)

```
    void create_physics_static(){
    btCollisionShape* staticShape = new btBoxShape(btVector3(1,1,1));

  float massaStatic = 0.0f; //affinché sia statico deve essere di massa
   ↪  nulla


  staticRB = createRB(massaStatic, btVector3(0,-5,0), staticShape);
   ↪  //leghiamo il puntatore al corpo statico

}
```

Per rendere la collisione realistica si assegna ai corpi un *coefficiente di restituzione*. Tale assegnazione avviene nella funzione per creare il corpo rigido

```
in createRB
    body->setRestitution(0.9f);
```

7.3 Codice 3: pendolo semplice

Il terzo codice si occupa di simulare un pendolo. La logica è di creare un vincolo tra il corpo e un punto fisso di *anchor*. Il tipo di vincolo è point-to-point. Nella funzione per la creazione del pendolo si definisce il vincolo e lo si lega al cubo. Si inserisce il vincolo nel DynamicsWorld. Nell'init si inizializza il pendolo insieme al cubo.

```
    void create_physics_pendulum(){
// Definiamo la posizione dell'ancoraggio del pendolo
btVector3 anchorPoint(5,5,0);

btVector3 anchorInCube = cubeRB->getCenterOfMassTransform().inverse() *
 ↪  anchorPoint;

  // Definiamo il vincolo punto-punto tra il cubo e l'ancoraggio
  btPoint2PointConstraint* pendulumConstraint = new
   ↪  btPoint2PointConstraint(
    *cubeRB,
    anchorInCube
  );

  //Aggiungiamo il vincolo al mondo fisico
  physics.dynamicsWorld->addConstraint(pendulumConstraint, true);

}
```

Si può lavorare sulle ampiezze e l'oscillazione del pendolo andando a modificare "g", ovvero il valore dell'accelerazione di gravità e la lunghezza L del pendolo, tenendo conto che

$$T = 2\pi\sqrt{\frac{L}{g}} \tag{6}$$

con T il periodo dell'oscillazione.
La risoluzione del pendolo consiste nell'integrazione dell'equazione del moto

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}sin(\theta) = 0 \tag{7}$$

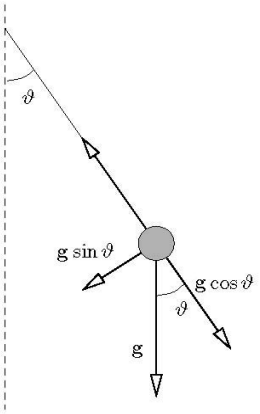

Figura 16: Schema del pendolo semplice. In Bullet si fissa l'anchor nell'intersezione dei due assi. Il corpo del pendolo è la mesh utilizzata.

7.4 Codice 4: pendolo smorzato

Il quarto codice articola il codice precedente andando ad aggiungere uno smorzamento, ovvero realizzando una condizione fisica più realistica nella simulazione in cui l'aria presenta un attrito viscoso che smorza le oscillazioni del pendolo.
Tramite BulletPhysics l'implementazione dell'attrito viscoso avviene tramite il metodo *setDamping* sul corpo rigido del Cubo. Il Damping è sia nel momento lineare che in quello angolare.
L'equazione differenziale di un pendolo smorzato non è di facile risoluzione analitica, in quanto presenta una dipendenza dalla derivata prima rispetto al tempo dell'angolo, ma tramite i metodi di integrazione utilizzati da Bullet l'implementazione è immediata.

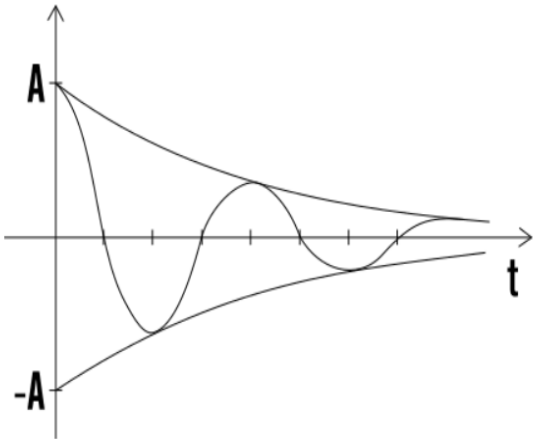

Figura 17: Legge oraria di un pendolo smorzato

7.5 Codice 5: double pendulum

Il quinto codice consiste nella simulazione di un sistema caotico, composto da un doppio pendolo. L'implementazione del doppio pendolo avviene tramite due vincoli Point-To-Point. Entrambi i pendoli in questo caso sono rappresentati da un corpo rigido, dove il primo pendolo è quello legato al cubo, mentre il secondo è quello legato al primo pendolo. Quest'ultimo deve essere necessariamente uno static affinché tutto il pendolo non venga trascinato in basso dalla gravità.
Tale sistema fisico non è risolvibile analiticamente ed è, per l'appunto, caotico. Il risultato è che, sebbene il codice non sia particolarmente raffinato, la simulazione è verosimile, nonostante si noti in alcuni momenti che la traiettoria è approssimata e poco plausibile nella vita reale. Osservando la traiettoria caotica di questo tipo d'oggetto si nota la somiglianza della simulazione[9].

[9] https://www.youtube.com/watch?v=pEjZd-AvPco

```
void create_physics_pendulum()
  btCollisionShape* pendulum1Shape = new btBoxShape(btVector3(0.5, 2.0,
   ↪  0.5));
  float massaPendulum1 = 1.0f;
  btVector3 posPendulum1(5, 5, 0);
  pendulum1RB = createRB(massaPendulum1, posPendulum1, pendulum1Shape);

  btCollisionShape* pendulum2Shape = new btBoxShape(btVector3(0.5, 2.0,
   ↪  0.5));
  float massaPendulum2 = 0.0f;
  btVector3 posPendulum2(7, 7, 0);
  pendulum2RB = createRB(massaPendulum2, posPendulum2, pendulum2Shape);
```

```
    void create_physics_constraints(){
  btVector3 anchorInCube(5,5,0);
  btVector3 anchorInPendulum1(0,0,0);

  btPoint2PointConstraint* pendulum1Constraint = new
   ↪  btPoint2PointConstraint(
        *cubeRB,
        *pendulum1RB,
        anchorInCube,
        anchorInPendulum1
    );
    physics.dynamicsWorld->addConstraint(pendulum1Constraint, true);

  //CREIAMO IL VINCOLO TRA IL PRIMO E IL SECONDO PENDOLO
  btVector3 anchorInPendulum1End(1,-2,0);
  btVector3 anchorInPendulum2(0,0,0);

  btPoint2PointConstraint* pendulum2Constraint = new
   ↪  btPoint2PointConstraint(
        *pendulum1RB,
        *pendulum2RB,
        anchorInPendulum1End,
        anchorInPendulum2
    );
    physics.dynamicsWorld->addConstraint(pendulum2Constraint, true);

}
```

7.6 BulletPhysics e i SoftBodies

I soft bodies sono la simulazione di corpi in cui si decide di escludere il vincolo di rigidità, ovvero delle distanze fisse tra i punti, per poter simulare la deformabilità dei corpi. L'idea è quella di simulare i corpi come nuvole di punti collegati tra loro con delle molle senza massa, in questo metodo si tiene conto delle forze interne che attraggono e respingono i punti adiacenti[10]. Data una mesh superficiale è necessario creare un algoritmo che crei un modello nel formalismo dei *voxel*, che permettono di tener conto del volume dell'oggetto.[11]

- In una dimensione, un soft body è una catena di nodi, dunque utile per la simulazione di corde, catene e simili.
- In due dimensioni si parla di *patch*, utili per la simulazione di tessuti.
- In tre dimensioni si opera su tetraedri. In questo caso non è più sufficiente utilizzare una mesh superficiale ma il volume del corpo deve essere composto da tetraedri[12].

BulletPhysics mette a disposizione diverse librerie ad alto livello per la trattazione, solitamente complicata, dei soft body. Inoltre quest'ultimi richiedono un elevato numero di calcoli per essere simulati e dunque vengono utilizzati principalmente nel rendering offline.

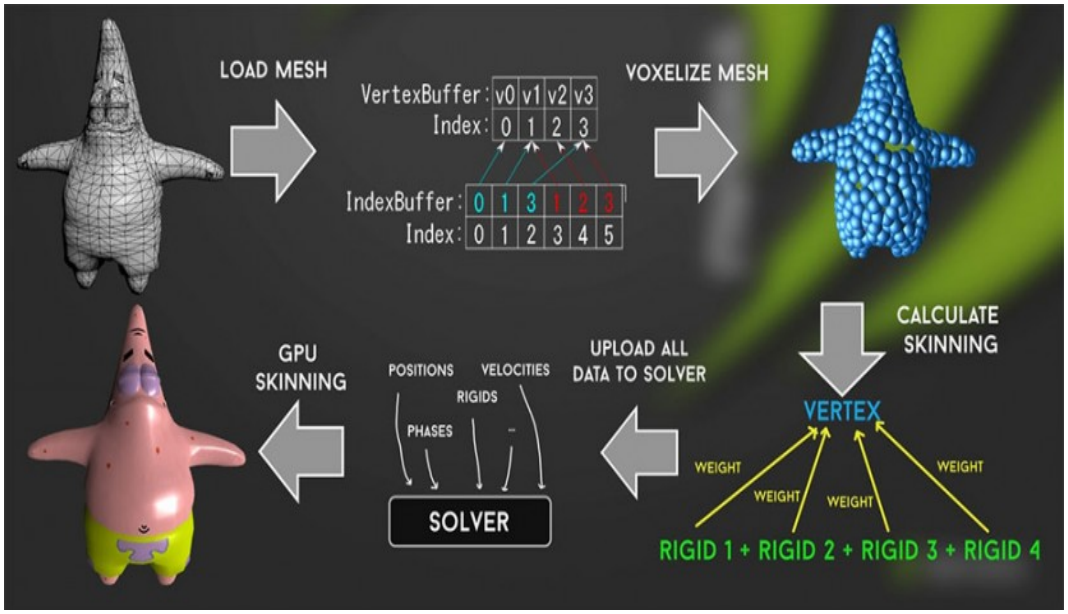

Figura 18: L'idea è quella di prendere una mesh e realizzarne una mesh col formalismo dei voxel, in questo modo si può tener conto del volume dell'oggetto e risolvere tutti i constraint tra le particelle del corpo

[10] https://en.wikipedia.org/wiki/Soft-body_dynamics
[11] https://www.digitalartsandentertainment.be/article/316/Graduation+Work%3A+Soft+body+physics+by+Simon+Coenen+
[12] https://docs.panda3d.org/1.10/python/programming/physics/bullet/softbodies

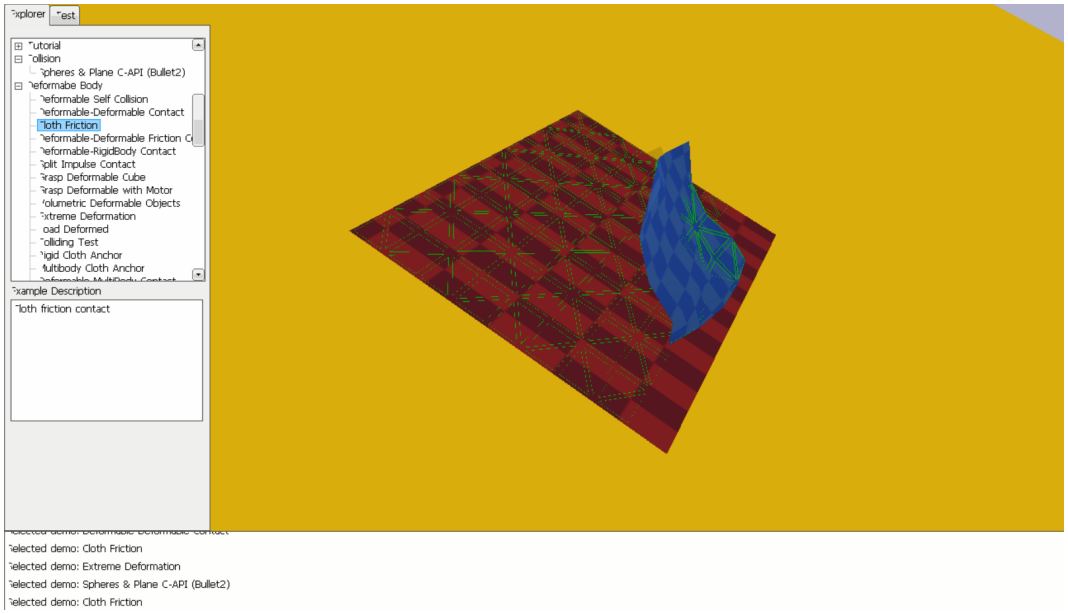

Figura 19: Esempio di simulazione di un tessuto tramite la Demo di OpenGL. Essendo un soft body in 2D si nota la composizione in triangoli su tutta la superficie della mesh. Nel caso di un rigidbody un plane sarebbe stato stato creato con un array di 6 vertici.