

UNIVERSITÀ DEGLI STUDI DI CASSINO E DEL LAZIO MERIDIONALE

Dipartimento di Ingegneria Elettrica e dell'Informazione



**Corso di laurea in Ingegneria Informatica e delle
Telecomunicazioni**

Simulazione della Dinamica di Corpi Rigidi Rettangolari

Relatore

Prof. Alessandro Bria

Candidato

Achille Cannavale
Matr. 0058721

A.A. 2022/2023

Indice

1 Rilevamento delle Collisioni	3
1 Circle Vs Circle	4
2 AABB	5
3 GJK (Gilbert Johnson Keerthi)	6
4 SAT (Separating Axis Theorem)	7
4.1 Calcolo dei Vertici	8
4.2 Matrice di Rototraslazione	8
4.3 Spiegazione	8
4.4 Implementazione in C++	9
4.4.1 Separazione degli Oggetti	11
4.4.2 Punti di Contatto	11
2 Risoluzione delle Collisioni	14
1 Spiegazione Matematica	15
2 Implementazione in C++	17
3 Progettazione e Implementazione	19
1 Ciclo Principale	21
2 Integrazione Numerica	23
2.1 Integrazione di Eulero	23
2.2 Verlet	25
2.3 Runge-Kutta	27
3 Oggetti di Gioco	28
4 Mondo di Gioco	34
5 Miglioramenti	37
5.1 Iterazioni	37
5.2 Vector Vs Array	38
5.3 Controlli Preliminari al Rilevamento delle Collisioni	38
5.4 Quad Tree	40
4 Conclusione	41
Lista delle Figure	43
Bibliografia	44

Introduzione

Negli ultimi decenni, lo sviluppo dei videogiochi si è evoluto a una velocità sorprendente, portando alla creazione di esperienze di gioco sempre più immersive e coinvolgenti, non solo nei giochi tridimensionali, ma anche nei giochi 2D. Tuttavia, i giochi 2D richiedono una solida base tecnologica per offrire una buona esperienza di gioco, e uno degli aspetti più critici è la gestione delle collisioni tra i vari oggetti presenti nel gioco.

La verifica delle collisioni è un'operazione fondamentale nei giochi 2D, in quanto consente di determinare se due o più oggetti si scontrano e di calcolare una risposta appropriata, come la ricostruzione della traiettoria degli oggetti o l'applicazione di una forza di reazione. Poiché la gestione delle collisioni può diventare molto complessa in presenza di oggetti irregolari o con forme complesse, è importante avere a disposizione algoritmi di verifica delle collisioni efficienti e precisi.

In questa tesi, ci concentreremo sulla progettazione e implementazione di un motore fisico specializzato nella verifica e nella risoluzione delle collisioni tra corpi rigidi rettangolari. Il motore fisico sarà sviluppato in C++ e potrà essere utilizzato per lo sviluppo di giochi 2D di differenti generi, come platformer, metroidvania, arcade e gestionali.

Come linea guida dell'implementazione seguiranno la serie di video "Let's Make a Physics Engine" [4] sulla creazione di un engine fisico in C# del canale YouTube Two-Bit Coding.

CAPITOLO 1

Rilevamento delle Collisioni

Il rilevamento delle collisioni in 2D è una tecnica fondamentale per molti tipi di applicazioni, specialmente nei giochi e nella simulazione grafica. In particolare è il processo di rilevamento dell'intersezione tra due oggetti, che possono essere rappresentati da poligoni, cerchi, rettangoli o altre forme geometriche.

Esso è uno degli elementi fondamentali per rendere i videogiochi realistici e coinvolgenti. Infatti permette ai giocatori di interagire con gli oggetti del gioco in modo naturale e fluido, creando una migliore esperienza di gioco.

L'obiettivo principale del rilevamento delle collisioni è quello di determinare se due oggetti si sovrappongono, ad esempio se un personaggio di un videogioco collide con un muro, oppure se due oggetti si scontrano in una simulazione fisica. Questo processo è essenziale per garantire un gioco fluido e realistico, in cui gli oggetti si comportano come ci si aspetta in base alle leggi della fisica.

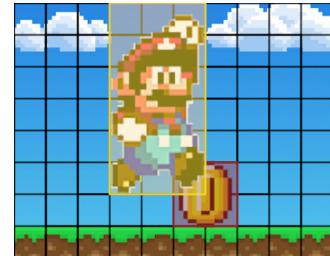


Figura 1.1: Hitbox Super Mario

[11]

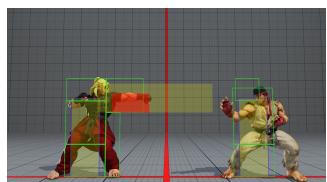


Figura 1.2: Hitbox Street Fighter
[2]

È possibile implementare la verifica delle collisioni utilizzando algoritmi di simulazione della dinamica di corpi rigidi. Questi algoritmi permettono di calcolare la posizione e la velocità di questi corpi, tenendo conto delle forze e delle accelerazioni che agiscono su di essi.

La simulazione della dinamica di corpi rigidi può essere utilizzata anche per creare effetti realistici di collisione e impatto, come ad esempio la repulsione di due auto dopo un urto o il rimbalzo di una pallina contro un muro.

Ci sono molti metodi diversi per effettuare il rilevamento delle collisioni in 2D e ogni metodo ha i propri vantaggi e svantaggi, a seconda del tipo di applicazione, del tipo di oggetti coinvolti nell'interazione e della complessità dell'interazione stessa.

I metodi più elementari sfruttano delle semplici relazioni tra le proprietà geometriche degli oggetti, come cerchi o rettangoli, mentre i metodi più complessi utilizzano tecniche come i test di separazione degli assi o la differenza di Minkowski per determinare se due oggetti si intersecano. Questi metodi sono spesso utilizzati per oggetti più complessi con forme irregolari.

In generale, il rilevamento delle collisioni è un processo computazionalmente intenso, soprattutto se si hanno molti oggetti in movimento nello stesso spazio. Per questo motivo, spesso si utilizzano tecniche di ottimizzazione per ridurre la complessità computazionale, ad esempio limitando il numero di oggetti che vengono controllati per le collisioni o utilizzando tecniche di suddivisione dello spazio.

Inoltre, può essere utilizzato anche in applicazioni tridimensionali, dove il processo è ancora più complesso, ma altrettanto fondamentale. In questo caso, uno dei metodi più utilizzati è il metodo di rilevamento di collisioni basato sulla geometria dei poligoni.

Nelle prossime sezioni andremo a spiegare il funzionamento di alcuni di questi metodi, focalizzandoci maggiormente sull'algoritmo SAT, in quanto costituisce il metodo impiegato nel progetto di questa tesi.

1.1 Circle Vs Circle

La verifica delle collisioni tra due sfere in 2D è un'operazione comune nei giochi e in questo processo, l'obiettivo è determinare se due sfere si stanno toccando o sovrapponendo.

Per effettuare la verifica delle collisioni tra due sfere, è necessario conoscere le loro posizioni e i loro raggi. Una volta che queste informazioni sono disponibili, è possibile calcolare la distanza tra i centri delle due sfere utilizzando la formula della distanza Euclidea (figura[1.4]):

$$dist(a, b) > r.a + r.b$$



Figura 1.3: Hitbox Zelda
[10]

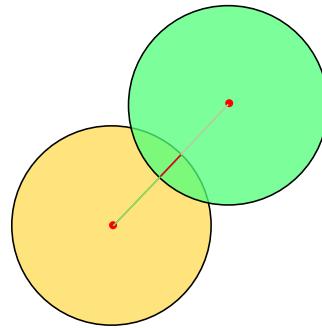


Figura 1.4: Circle Collision Detection

Se la distanza tra i centri delle due sfere è minore della somma dei loro raggi, allora le sfere si stanno toccando o sovrapponendo. In questo caso, si può concludere che c'è stata una collisione tra le due sfere.

1.2 AABB

Anche la verifica delle collisioni tra due rettangoli orientati agli assi (AABB) (figura[1.5]) è un'operazione comune nei giochi e per effettuarla è necessario conoscere le loro posizioni e le loro dimensioni. Una volta che queste informazioni sono disponibili, è possibile calcolare le aree dei rettangoli proiettati sugli assi X e Y. Questi valori saranno utilizzati per determinare se c'è stata una collisione.

Se le proiezioni dei rettangoli sugli assi X e Y si sovrappongono, allora i due rettangoli si stanno toccando o sovrapponendo. In questo caso, si può concludere che c'è stata una collisione tra i due rettangoli.

La formula di collisione tra due rettangoli AABB è solitamente implementata utilizzando la formula:

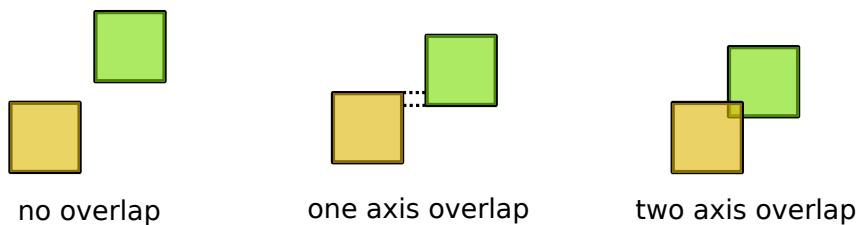


Figura 1.5: AABB Collision Detection

$$\begin{cases} a.x + a.width > b.x \\ a.y < b.y + b.height \\ a.y + a.height > b.y \end{cases}$$

Tuttavia per rettangoli non orientati lungo gli assi cartesiani, il tutto si complica e da qui nasce la necessità di un algoritmo più potente.

1.3 GJK (Gilbert Johnson Keerthi)

L'algoritmo GJK è nato nel 1988 dagli ingegneri Elmer Gilbert, Daniel Johnson e Sathiya Keerthi, ed è tutt'oggi una delle soluzioni più usate per il rilevamento delle collisioni tra oggetti in qualsiasi dimensione, in applicazioni di computer graphics e robotica.

Iniziamo distinguendo due grandi famiglie di figure, quelle convesse e quelle concave. Una figura si considera convessa se, per ogni retta che lo attraversa, la incida solo due volte:

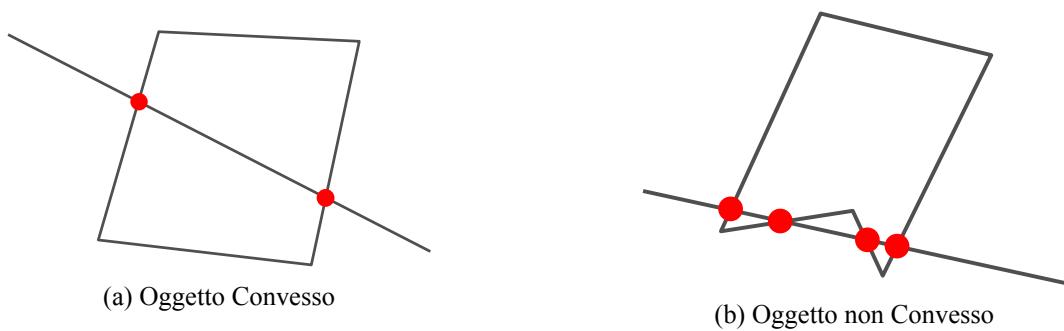


Figura 1.6: Convessità

Le figure convesse sono molto più facili da analizzare attraverso l'algoritmo GJK (e anche con l'algoritmo SAT, come vedremo più avanti), ma questo non vuol dire che sia impossibile lavorare con figure concave, dato che è possibile suddividere una figura concava in una serie di figure convesse.

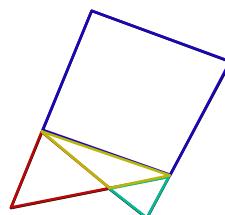


Figura 1.7: Trasformazione da figura concava a serie di figure convesse

Un'operazione fondamentale per questo algoritmo è la Differenza di Minkowski, ovvero sommare ogni punto della prima figura con ogni punto negato della seconda

figura:

$$(A - B) = \{a + (-b) \mid a \in A, b \in B\}$$

Questa operazione genera una figura risultante che sottostà principalmente alle seguenti due proprietà:

- se A e B convesse $\longrightarrow (A - B)$ convessa
- se A e B si sovrappongono $\longrightarrow (0, 0) \in (A - B)$

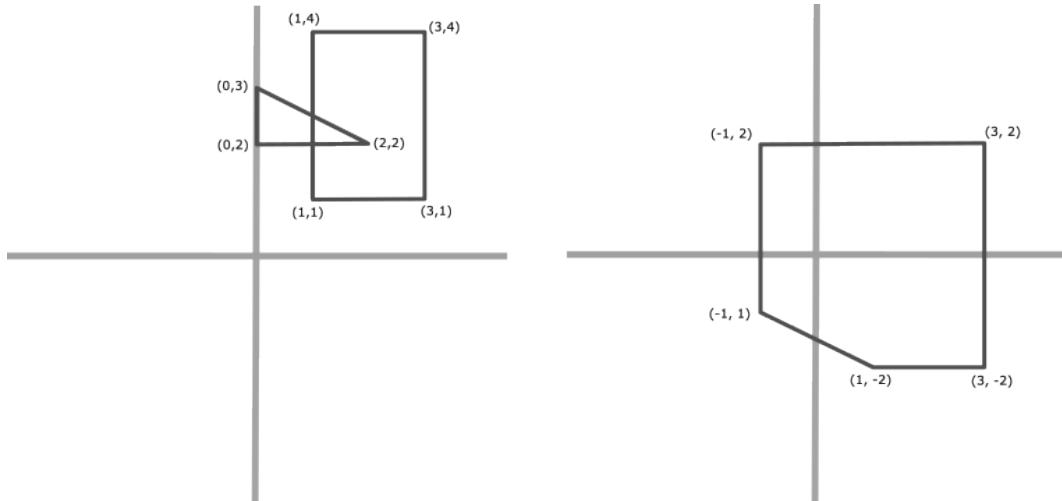


Figura 1.8: Differenza di Minkowski

1.4 SAT (Separating Axis Theorem)

Gli algoritmi di verifica delle collisioni avanzati sono tecniche sofisticate utilizzate in giochi e simulazioni per gestire la collisione tra oggetti complessi in modo efficiente ed accurato. Questi algoritmi consentono di determinare se due oggetti si stanno toccando o sovrapponendo nello spazio 2D o 3D.

A differenza degli algoritmi di verifica delle collisioni semplici, che utilizzano forme geometriche semplici come cerchi o rettangoli orientati agli assi per rappresentare gli oggetti, gli algoritmi di verifica delle collisioni avanzati lavorano con forme irregolari e complesse, rendendo la verifica delle collisioni più difficile rispetto agli oggetti geometrici semplici.

In questa tesi analizzeremo maggiormente l'algoritmo SAT (Separating Axis Theorem) essendo il teorema più adattabile alla nostra situazione e di più semplice implementazione rispetto all'ugualmente noto algoritmo GJK (Gilbert-Johnson-Keerthi).

La prima cosa da dire su questo algoritmo è che esso funziona maggiormente con figure convesse, quindi andiamo a darne una brevissima definizione.

1.4.1 Calcolo dei Vertici

Per utilizzare l'algoritmo SAT per il rilevamento di collisioni tra oggetti, è sufficiente conoscere le posizioni dei vertici degli oggetti coinvolti. Nel caso di rettangoli, i vertici possono essere ottenuti conoscendo la posizione del centro del rettangolo e il suo angolo di rotazione, attraverso l'utilizzo della matrice di rototraslazione.

1.4.2 Matrice di Rototraslazione

Per prima cosa possiamo ricavare i vertici del rettangolo non ruotato semplicemente con la seguente formula:

$$\text{vertex}_i = \left(x_c \pm \frac{w}{2}, y_c \pm \frac{h}{2} \right)$$

Una volta fatto ciò possiamo utilizzare la seguente formula che utilizza la matrice di rototraslazione per avere le coordinate dei vertici del rettangolo ruotato (figura[1.9]):

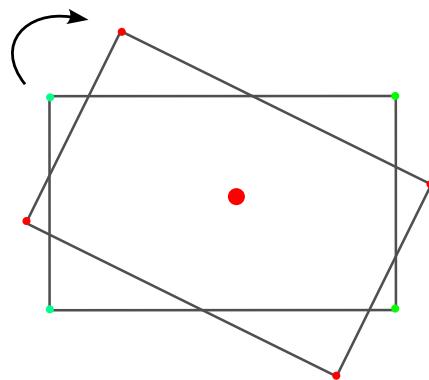


Figura 1.9: Rotazione

$$\text{rot_vertex}_i = \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

1.4.3 Spiegazione

Il SAT è un algoritmo che utilizza il concetto di assi di separazione per determinare se due oggetti si scontrano. L'idea di base del SAT è quella di proiettare gli oggetti su diversi assi e verificare se c'è sovrapposizione tra le proiezioni. Se c'è sovrapposizione su tutti gli assi, allora gli oggetti si scontrano effettivamente.

In particolare gli assi che dovranno essere testati saranno gli assi che hanno la stessa direzione delle normali dei lati degli oggetti (figura[1.11]).

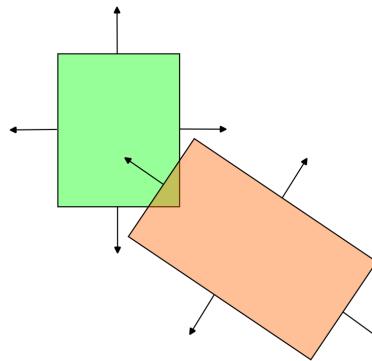


Figura 1.10: Oggetti Sovrapposti

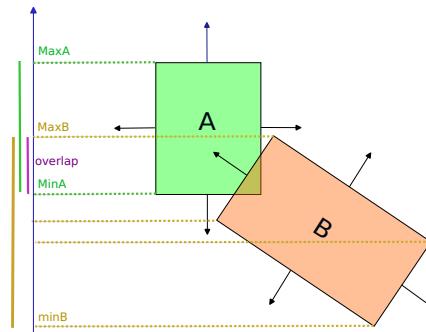


Figura 1.11: Proiezioni Lungo le Normali

Per verificare se l’asse i esimo è l’asse di separazione, dovremo proiettare il vertice massimo e il vertice minimo di entrambi gli oggetti.

Nel caso in cui ci sia una sovrapposizione tra il segmento proiettato dall’oggetto A e il segmento proiettato dall’oggetto B, dovremo proseguire iterativamente ad analizzare l’asse $i+1$ esimo.

Se, analizzati tutti gli assi, le proiezioni si sovrappongono, vorrà dire che i due oggetti stanno collidendo.

Altrimenti, non appena troveremo un asse con una separazione tra le proiezioni, potremo immediatamente asserire che i due oggetti non stanno collidendo.

1.4.4 Implementazione in C++

Per prima cosa calcoliamo le normali, per ogni lato, di ogni oggetto:

```

1 Vector2f CalculateNormal(Vector2f vertexA, Vector2f vertexB)
2 {
3     Vector2f directionVector = vertexB - vertexA;
4     return Vector2f(directionVector.y, -directionVector.x);
5 }
```

```

7
8 std::array <Vector2f, 4> CalculateNormals(std::array <Vector2f,
9     4>& vertices)
10 {
11     std::array <Vector2f, 4> normals(vertices.size());
12     for (int i = 0; i < vertices.size(); i++)
13     {
14         normals[i] = CalculateNormal(vertices[i + 1 % vertices.size
15             ()], vertices[i]);
16     }
17     normals[vertices.size() - 1] = CalculateNormal(vertices[0],
18             vertices[vertices.size() - 1]);
19     return normals;
20 }
```

Ora, per ogni normale trovata, proietteremo ogni vertice, cercando la proiezione minima e la proiezione massima per entrambi gli oggetti.

Una volta trovati, imporremo la condizione `if (maxA < minB || maxB < minA)` che se risulterà vera significherà che non c'è stata una collisione, viceversa i due oggetti sono in uno stato di collisione e indicheremo come profondità di penetrazione $\min(maxB - minA, maxA - minB)$, e come normale di collisione l'ultima normale controllata.

```

1 for (Vector2f normal : normals)
2 {
3     float minA = INFINITY;
4
5     float maxA = -INFINITY;
6
7     for (Vector2f vertex : verticesA)
8     {
9         float proj = Math::Dot(vertex, normal);
10        minA = std::min(minA, proj);
11        maxA = std::max(maxA, proj);
12    }
13
14    float minB = INFINITY;
15
16    float maxB = -INFINITY;
17
18    for (Vector2f vertex : verticesB)
19    {
20        float proj = Math::Dot(vertex, normal);
21        minB = std::min(minB, proj);
22        maxB = std::max(maxB, proj);
23    }
24
25    if (maxA < minB || maxB < minA)
26    {
27        result.setAreColliding(false);
28        result.setCollidingAxis(Vector2f(0, 0));
29        result.setDepth(0);
30        return result;
31 }
```

```
31 }
32
33     float axisDepth = (std::min(maxB - minA, maxA - minB));
34
35     if (axisDepth < depth)
36     {
37         depth = axisDepth;
38         result_normal = normal;
39     }
40
41 }
```

1.4.4.1 Separazione degli Oggetti

Prima di procedere verso la fase di risoluzione delle collisioni, dobbiamo separare i due oggetti, e questo può essere fatto grazie alla profondità di penetrazione e al vettore normale che abbiamo ricavato prima in questo modo:

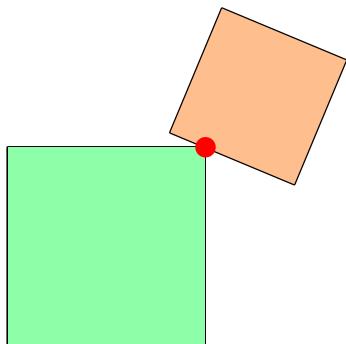
```
1 objA->MoveBy(-normal.x * depth / 2, -normal.y * depth / 2);  
2 objB->MoveBy(normal.x * depth / 2, normal.y * depth / 2);
```

1.4.4.2 Punti di Contatto

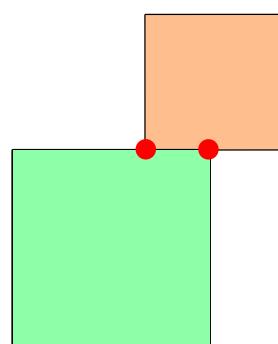
L'ultimo passo prima di arrivare alla risoluzione delle collisioni è quello di calcolare i punti di contatto dovuti all'urto e per fare ciò ci serviremo di questa funzione che calcola quale sia il punto più vicino ad un punto p dato:

```
1 PointToSegmentType PointToSegmentDistance(Vector2f p, Vector2f va,
2                                         Vector2f vb)
3 {
4     PointToSegmentType result;
5     Vector2f ab = vb - va;
6     Vector2f ap = p - va;
7
8     float proj = Math::Dot(ap, ab);
9     float abLenSq = (ab.x * ab.x + ab.y * ab.y);
10    float d = proj / abLenSq;
11
12    if (d <= 0)
13        result.closestPoint = va;
14
15    else if (d >= 1)
16        result.closestPoint = vb;
17    else
18        result.closestPoint = va + ab * d;
19
20    result.distSq = Math::GetDistanceSquared(p, result.
21                                              closestPoint);
22
23    return result;
24}
```

Grazie a questa funzione, possiamo calcolare i punti di contatto con due cicli innestati, che scorrono su tutti i vertici di entrambi gli oggetti valutando quali e quanti siano i punti di contatto (figura[1.12]):



(a) Un punto di Contatto



(b) Due punti di Contatto

Figura 1.12: Punti di Contatto

```
1 ContactType Collision::FindContactPoints(GameObject *objA,
2     GameObject *objB)
3 {
4     Vector2f contact1(0, 0);
5     Vector2f contact2(0, 0);
6     int contactsNumber = 0;
7     float minDistSq = INFINITY;
8
9     for (int i = 0; i < objA->GetVertices().size(); i++)
10    {
11        Vector2f p = objA->GetVertices()[i];
12        for (int j = 0; j < objB->GetVertices().size(); j++)
13        {
14            Vector2f va = objB->GetVertices()[j];
15            Vector2f vb = objB->GetVertices()[(j + 1) % objB->
16 GetVertices().size()];
17            PointToSegmentType p2s = PointToSegmentDistance(p, va,
18 vb);
19
20            if (Math::NearlyEqual(p2s.distSq, minDistSq))
21            {
22                if (!Math::NearlyEqual(p2s.closestPoint, contact1)
23 &&
24                    !Math::NearlyEqual(p2s.closestPoint, contact2))
25            )
26            {
27                contact2 = p2s.closestPoint;
28                contactsNumber = 2;
29            }
30        }
31        else if (p2s.distSq < minDistSq)
32        {
33
```

```
29             minDistSq = p2s.distSq;
30             contactsNumber = 1;
31             contact1 = p2s.closestPoint;
32         }
33     }
34 }
35
36
37 for (int i = 0; i < objB->GetVertices().size(); i++)
38 {
39     Vector2f p = objB->GetVertices()[i];
40     for (int j = 0; j < objA->GetVertices().size(); j++)
41     {
42         Vector2f va = objA->GetVertices()[j];
43         Vector2f vb = objA->GetVertices()[(j + 1) % objA->
44 GetVertices().size()];
45         PointToSegmentType p2s = PointToSegmentDistance(p, va,
46 vb);
47
48         if (Math::NearlyEqual(p2s.distSq, minDistSq))
49         {
50             if (!Math::NearlyEqual(p2s.closestPoint, contact1)
51                 &&
52                 !Math::NearlyEqual(p2s.closestPoint, contact2))
53             {
54                 contact2 = p2s.closestPoint;
55                 contactsNumber = 2;
56             }
57         }
58         else if (p2s.distSq < minDistSq)
59         {
60             minDistSq = p2s.distSq;
61             contactsNumber = 1;
62             contact1 = p2s.closestPoint;
63         }
64     }
65     return ContactType(contact1, contact2, contactsNumber);
66 }
```

CAPITOLO 2

Risoluzione delle Collisioni

La risoluzione delle collisioni rappresenta un aspetto fondamentale nell'ambito dello sviluppo dei videogiochi moderni. Si tratta di un processo complesso e cruciale che si occupa di gestire l'interazione tra gli oggetti all'interno del mondo di gioco. Quando un giocatore si immerge nell'universo virtuale di un videogioco, vuole essere coinvolto in un'esperienza fluida, realistica e coinvolgente.



Figura 2.1: Danno vitale in Zelda

Quindi l'obiettivo principale è quello di gestire le collisioni tra gli oggetti del gioco, come personaggi, oggetti, muri, e altri elementi ambientali. Ciò significa che quando un personaggio cammina, corre, salta o interagisce con il mondo di gioco, è necessario assicurarsi che i suoi movimenti e azioni siano correttamente influenzati dagli ostacoli presenti nell'ambiente. Senza una corretta gestione delle collisioni, gli oggetti potrebbero attraversarsi, cadere attraverso il terreno, rimanere bloccati all'interno delle pareti o avere comportamenti irrealistici, compromettendo l'esperienza di gioco.



Figura 2.2: Risposta agli urti in GTA

[14]

Ad esempio, in un gioco di guida, una correttissima risoluzione delle collisioni è essenziale per evitare che le auto si scontrino senza motivo o attraversino gli oggetti nel mondo virtuale. Nel contesto di un videogioco di combattimento, gli scontri tra personaggi devono essere gestiti con precisione per assicurare che i colpi vengano correttamente rilevati e valutati, influenzando le meccaniche di gioco in modo coerente.

È fondamentale implementare un sistema di risoluzione delle collisioni in modo efficiente ed ottimizzato poiché i moderni videogiochi spesso presentano ambienti vasti e complessi. Ciò può contribuire a ridurre il carico

computazionale e garantire che il gioco funzioni in modo fluido anche su hardware meno potenti.

Le tecniche fisiche di risoluzione delle collisioni nei videogiochi 2D si basano sulle leggi della fisica per gestire le interazioni tra gli oggetti virtuali. Ecco alcune delle principali tecniche fisiche utilizzate:

- Conservazione della Quantità di Moto: Quando avviene una collisione tra due oggetti, la loro quantità di moto totale prima e dopo la collisione rimane costante. Questo implica che la somma delle loro velocità prima della collisione è uguale alla somma delle loro velocità dopo la collisione. Utilizzando questa legge, è possibile calcolare le nuove velocità e direzioni degli oggetti dopo la collisione.
- Impulso e Forze: In questo approccio, gli oggetti del gioco sono trattati come masse fisiche con proprietà di massa e velocità. Durante una collisione, gli oggetti si scambiano impulsi in base alle loro caratteristiche fisiche. In questo modo, le forze risultanti dalla collisione possono essere utilizzate per calcolare le nuove velocità e direzioni degli oggetti coinvolti.
- Rilevamento della Penetrazione: Questa tecnica è utilizzata per correggere eventuali penetrazioni tra gli oggetti dopo una collisione. Quando due oggetti collidono, potrebbero sovrapporsi parzialmente. Una volta rilevata la penetrazione verranno spostati gli oggetti coinvolti in modo che non si intersechino più.
- Risoluzione delle Collisioni basata su matrici di trasformazione: Questa tecnica si basa sull'uso di matrici di trasformazione per gestire le collisioni. Esse sono utilizzate per definire la posizione, la scala e la rotazione degli oggetti. Durante una collisione, le matrici di trasformazione vengono aggiornate per garantire che gli oggetti si muovano correttamente e che le interazioni siano fisicamente coerenti.

2.1 Spiegazione Matematica

Per determinare l'impulso da applicare ai due oggetti a seguito dell'urto nel primo punto di contatto, si utilizza una formula che tiene conto dell'elasticità dei materiali coinvolti, della velocità relativa tra i due oggetti e delle loro masse e momenti di inerzia.

Il primo passo è definire la velocità relativa tra i due oggetti. Per fare ciò, si definiscono le rette che vanno dai centri delle figure A e B al primo punto di contatto, indicandole rispettivamente con \mathbf{r}^{AP_1} e \mathbf{r}^{BP_1} , come in figura [2.3].

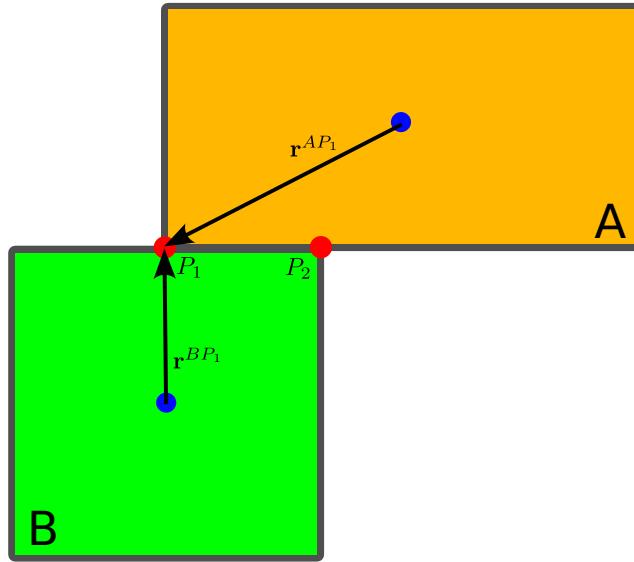


Figura 2.3: Rette verso i punti di contatto

Una volta definite queste rette, è possibile definire le componenti ortogonali:

$$\begin{cases} \mathbf{r}_\perp^{AP_1} \\ \mathbf{r}_\perp^{BP_1} \end{cases}$$

Possiamo quindi definire la velocità relativa tra i due oggetti come:

$$\mathbf{v}_1^{AB} = \left(\mathbf{v}_A + \omega_A * \mathbf{r}_\perp^{AP_1} \right) - \left(\mathbf{v}_B + \omega_B * \mathbf{r}_\perp^{BP_1} \right)$$

Per calcolare i momenti di inerzia dei rettangoli, si può utilizzare una formula semplice che tiene conto della massa dell'oggetto e delle sue dimensioni.

$$\begin{cases} I_A = \frac{M_A * (w_A^2 + h_A^2)}{12} \\ I_B = \frac{M_B * (w_B^2 + h_B^2)}{12} \end{cases}$$

E per finire useremo come parametro di restituzione, il minore dei due oggetti. A questo punto potremo utilizzare la seguente formula [7] che rappresenta l'impulso da applicare ai due oggetti a seguito dell'urto nel primo punto di contatto:

$$\mathbf{j}_1 = \frac{-(1+e) \mathbf{v}_1^{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n} \left(\frac{1}{M_A} + \frac{1}{M_B} \right) + \frac{(\mathbf{r}_\perp^{AP_1} \cdot \mathbf{n})^2}{I_A} + \frac{(\mathbf{r}_\perp^{BP_1} \cdot \mathbf{n})^2}{I_B}} \quad [7]$$

L'impulso viene quindi applicato alle velocità lineari degli oggetti, incrementando la velocità dell'oggetto A e diminuendo quella dell'oggetto B:

$$\begin{cases} \mathbf{v}_A = \mathbf{v}_A + \frac{\mathbf{j}_1}{M_A} \\ \mathbf{v}_B = \mathbf{v}_B - \frac{\mathbf{j}_1}{M_B} \end{cases}$$

Allo stesso tempo, la velocità angolare di ciascun oggetto viene incrementata proporzionalmente all'impulso applicato e al momento di inerzia dell'oggetto stesso:

$$\begin{cases} \omega_A = \omega_A + \frac{\mathbf{r}_{\perp}^{AP_1} \times \mathbf{j}_1}{I_A} \\ \omega_B = \omega_B + \frac{\mathbf{r}_{\perp}^{BP_1} \times -\mathbf{j}_1}{I_B} \end{cases}$$

Il procedimento per il secondo punto di contatto è analogo a quello descritto per il primo punto di contatto.

2.2 Implementazione in C++

Il codice per la risoluzione delle collisioni andrà in un loop che itererà su ogni punto di collisione:

```
1 for (int i = 0; i < contact_number; i++)
2 {
3     //Codice
4 }
```

Il primo passo sarà quello di calcolare la velocità relativa per mezzo delle rette $\mathbf{r}_{\perp}^{AP_i}$ e $\mathbf{r}_{\perp}^{BP_i}$:

```
1 Vector2f ra = m_contactList[i] - t_objA->GetCenter();
2
3 Vector2f rb = m_contactList[i] - t_objB->GetCenter();
4
5 Vector2f raPerp(-ra.y, ra.x);
6
7 Vector2f rbPerp(-rb.y, rb.x);
8
9 Vector2f angularLinearVelocityA = raPerp * t_objA->
    GetAngularVelocity();
10
11 Vector2f angularLinearVelocityB = rbPerp * t_objB->
    GetAngularVelocity();
12
13 Vector2f vrel = (t_objA->GetVelocity() + angularLinearVelocityA) -
    (t_objB->GetVelocity() + angularLinearVelocityB);
```

Fatto ciò possiamo calcolare direttamente l'impulso e dividerlo per il numero di punti di contatto:

```

1 float contactVelocityMag = Math::Dot(vrel, normal);
2
3 float raPerpDotN = Math::Dot(raPerp, normal);
4
5 float rbPerpDotN = Math::Dot(rbPerp, normal);
6
7 float denom = t_objA->GetInvMass() + t_objB->GetInvMass() + (
8     raPerpDotN * raPerpDotN) / (t_objA->GetMomentOfInertia()) + (
9     rbPerpDotN * rbPerpDotN) / (t_objB->GetMomentOfInertia());
10
11 float j = -(1 + e) * contactVelocityMag;
12
13 j /= denom;
14
15 Vector2f impulse = normal * j;
```

L'ultimo passo rimane quello di applicare ad entrambi gli oggetti questo impulso appena calcolato:

```

1 t_objA->SetVelocity(t_objA->GetVelocity() + impulse * t_objA->
2     GetInvMass());
3
4
5
6
7 t_objB->SetVelocity(t_objB->GetVelocity() - impulse * t_objB->
8     GetInvMass());
9
10 t_objB->SetAngularVelocity(t_objB->GetAngularVelocity() + Math::
11     Cross(rb, -impulse) / t_objB->GetMomentOfInertia());
```

CAPITOLO 3

Progettazione e Implementazione

In questo capitolo andremo a definire quelle che sono state le varie scelte progettuali, prese durante l'implementazione del progetto di tesi. L'obiettivo del progetto era quello di costruire un motore fisico che fosse in grado di gestire la dinamica di oggetti rettangolari ovvero i loro movimenti imperturbati e le loro risposte ad urti contro oggetti statici o contro altri oggetti dinamici, il tutto garantendo una sufficiente correttezza fisica.

Il linguaggio scelto è stato il C++, in quanto rimane tutt'oggi uno dei linguaggi di programmazione più utilizzati e più completi sotto molti punti di vista, e di conseguenza l'IDE Visual Studio 2022[5] è stato utilizzato come ambiente di sviluppo per la sua rinomata efficienza e compatibilità con il linguaggio.



Figura 3.1: Logo Visual Studio



Figura 3.2: Logo SDL
Per quanto riguarda la grafica del motore fisico occorreva una libreria leggera e di semplice utilizzo, per non inficiare eccessivamente sulle prestazioni della simulazione fisica. Per questi motivi la scelta è ricaduta sulla libreria SDL2 (Simple DirectMedia Layer 2)[8], che è una libreria multi-piattaforma per lo sviluppo di applicazioni multimediali, in particolare giochi, su diversi sistemi operativi. Essa fornisce un'interfaccia semplice e portabile per la gestione di grafica 2D, audio, input utente e altro ancora, ed inoltre è perfettamente compatibile con il C++.

Possiamo citare alcuni esempi di giochi creati grazie all'utilizzo della libreria SDL, come SuperTux[16] (3.3a), un gioco in stile SuperMario che ha come protagonista Tux, il pinguino logo di Linux. O ancora potremo citare OpenTTD[15] (3.3b), un gioco di simulazione dove si andranno a gestire le reti di trasporti di un'intera città. E sulla falsa riga dell'ultimo citato, possiamo nominare Unknown Horizons[1] (3.3c), ovvero un gioco di strategia in tempo reale e di costruzione di una città.



(a) SuperTux, The SuperTux Team



(b) OpenTTD, OpenTTD Team



(c) Unknown Horizons, Cj Birch

Figura 3.3: Esempi di giochi fatti con SDL

Per quanto riguarda invece la parte fisica del progetto, si è tentato di creare, in maniera seppur ridotta, una versione della ben nota libreria box2D[3], che fornisce un'ampia gamma di funzionalità per la gestione di corpi rigidi, collisioni, forze e attriti. Essa rappresenta tutt'oggi una delle librerie di fisica 2D più utilizzate, con una vasta comunità di sviluppatori che sono sempre al lavoro per migliorarla. Grazie a questa libreria sono nati giochi come Angry Birds[9] (3.4a), Limbo[13] (3.4b) e Hill Climb Racing[6] (3.4c).



(a) Angry Birds, Rovio Mobile



(b) Limbo, Playdead



(c) Hill Climb Racing, Finger-soft

Figura 3.4: Esempi di giochi fatti con box2D

3.1 Ciclo Principale

La prima cosa da fare per rendere funzionante il motore fisico è quella di creare un ciclo principale che gestirà l'aggiornamento fisico, grafico e la gestione degli eventi. La cosa più banale da fare sarebbe quella di creare un unico ciclo e inserire le seguenti funzioni:

```

1 while(running)
2 {
3     Update();
4     Render();
5 }
```

Ma così facendo, più il numero di "frames per secondo" aumentano e più il gioco andrà veloce, e questo non è sicuramente il comportamento desiderato, dato che la velocità del programma dovrebbe essere indipendente dall'hardware dell'utente.

Una cosa che è possibile fare per ovviare a questo problema è quella di calcolare il tempo trascorso da un frame all'altro e passare questa differenza (*frameTime*) alla funzione *Update(frameTime)* per far sì che qualsiasi operazione possa essere scalata:

```

1 while(running)
2 {
3     frameTime = t_now - t_last_update;
4     t_last_update += frameTime;
5     Update(frameTime);
6     Render();
7 }
```

Tra i più noti problemi di questo sistema, si eleva soprattutto il fatto che la precisione dei calcoli sia molto bassa, infatti potrebbe accadere che un oggetto possa del tutto non rilevare un ostacolo (3.5):

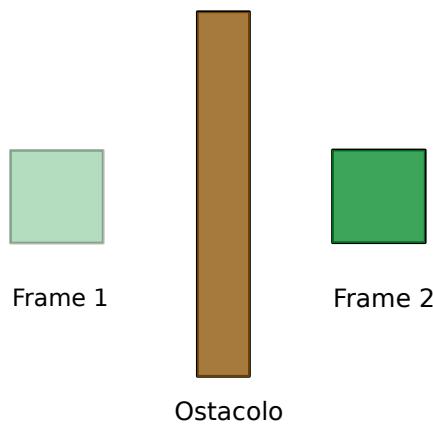


Figura 3.5: Mancato Rilevamento della Collisione

Per risolvere questo problema si è ricorsi al seguente schema:

```

1 int main(int argc, char** argv)
2 {
3     game->Init();
4     Timer<float> frameTimer;
5     float dt = 0.01f;
6     float accumulator = 0;
7
8     while (game->Running())
9     {
10         float frameTime = frameTimer.restart();
11
12         accumulator += frameTime;
13         while (accumulator >= dt)
14         {
15             game->Update(dt);
16             accumulator -= dt;
17         }
18         game->Render();
19     }
20     return 0;
21 }
```

Da come possiamo evincere dal codice soprastante, la prima cosa che viene fatta è inizializzare l'oggetto, che descriveremo nel dettaglio nelle prossime sezioni, e l'inizializzazione di alcuni parametri importanti per la gestione del ciclo di gioco:

- Una variabile *dt* che rappresenta l'intervallo di tempo tra gli aggiornamenti fisici del gioco, ovvero lo "step" fisico.
- Una variabile *accumulator* che verrà utilizzata per accumulare il tempo trascorso da quando l'ultimo aggiornamento fisico è stato effettuato.
- Un oggetto *Timer* che avrà il compito di misurare il tempo trascorso tra un frame e l'altro. Una volta fatto ciò si entrerà nel ciclo di gioco che verrà iterato per tutta la durata del programma.

Subito dopo verrà definita la variabile *frameTime* che verrà posta uguale a *frameTimer.restart()*, che non fa altro che resettare il timer e restituire quanto tempo è trascorso dal suo ultimo avvio, e a questo punto verrà incrementato l'accumulatore con *frameTime*.

Quindi si entrerà nel ciclo interno, che itererà finché l'accumulatore sarà maggiore di *dt*, dove l'oggetto game chiamerà la funzione *Update()* per aggiornare la fisica, e l'accumulatore verrà decrementato di *dt*.

Usciti dal ciclo interno, l'oggetto game chiamerà la funzione *Render()* per disegnare su schermo le modifiche effettuate dall'aggiornamento fisico.

Questo schema rende la velocità di gioco consistente, indipendente dall'hardware di utilizzo dell'utente e garantirà una simulazione deterministica, dato che lo

step che passiamo alla funzione *Update()* è un parametro fissato e questo significa che lo stesso input dato allo stesso istante garantirà lo stesso risultato. E infine, il problema più grave a essere risolto nel nostro caso sarà il tralasciamento delle collisioni.

3.2 Integrazione Numerica

Se pensiamo al semplice caso di un oggetto che cade nel vuoto, con moto uniformemente accelerato, possiamo descrivere il suo movimento attraverso la seguente legge oraria continua:

$$x = x_0 + v_0 t + \frac{1}{2} a t^2 \quad (3.1)$$

Tuttavia, i computer non hanno l'idea di "continuo", dato che essi lavorano nel campo discreto, quindi non possiamo utilizzare questa funzione continua e per questo introdurremo l'Integrazione Numerica. Questo metodo ha come scopo di discettizzare l'asse dei tempi, per rendere il computer capace di stimare la posizione dell'oggetto al frame successivo rispetto a quello corrente.

Alcuni giochi eseguono questi calcoli più di una volta per frame, come ad esempio Super Mario 64 [12], che svolge l'integrazione numerica quattro volte per frame.



Figura 3.6: Super Mario 64

3.2.1 Integrazione di Eulero

L'integrazione più semplice da adottare è l'Integrazione Esplicita di Eulero che prevede la posizione dell'oggetto nel frame successivo a quello corrente per mezzo del suo vettore velocità.

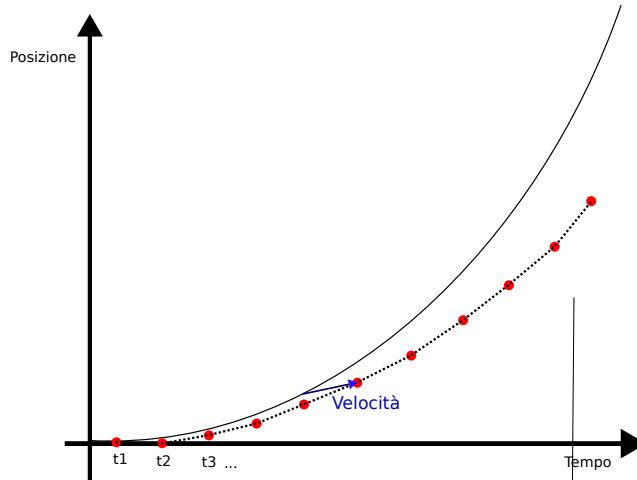


Figura 3.7: Grafico Integrazione Esplicita di Eulero

$$\begin{cases} x_{n+1} = x_n + v_n \Delta t \\ v_{n+1} = v_n + a \Delta t \end{cases} \quad (3.2)$$

Ovviamente abbiamo discusso il caso di un moto rettilineo uniforme per semplicità, ma tutto ciò che è stato detto può adattarsi a qualsiasi tipo di funzione. Quindi per adattare le equazioni già enunciate ad un caso generale, possiamo includere l'equazione della seconda legge della dinamica:

$$F = ma \longrightarrow a = \frac{F}{m} \quad (3.3)$$

Dunque, in maniera molto semplice possiamo implementare in codice ciò che abbiamo esposto:

```

1 class Object
2 {
3 public:
4     Object(float t_x; float t_y; float w; float t_h; Vector2f
5             t_vel; float t_mass)
6     {
7         x = t_x;
8         y = t_y;
9         w = t_w;
10        h = t_h;
11        vel = t_vel;
12        mass = t_mass;
13    }
14
15    update(dt)
16    {
17        float a = sum_forces / mass;
18        x += vel.x *dt;
19        y += vel.y *dt;
20        vel.x += a *dt;
21    }
22}
```

```

20     vel.y += a *dt;
21 }
22
23 private:
24     float x, y, w, h, mass;
25     Vector2f vel;
26 };

```

Ma come abbiamo potuto constatare dalla figura, la stima non sarà perfetta poiché abbiamo un Δt non infinitesimale tra un frame e l'altro che, all'aumentare, genererà una stima peggiore. Inoltre, il sistema in cui verrà utilizzato questo metodo di integrazione, guadagnerà energia con il passare del tempo, essendo questa una proprietà intrinseca del metodo stesso.

Per ovviare a questi problemi introduciamo il metodo di Integrazione Semi-Implicita di Eulero, che consiste semplicemente nello scambiare di posto le seguenti righe di codice:

```

1 update(dt)
2 {
3     float a = sum_forces / mass;
4     vel.x += a *dt;
5     vel.y += a *dt;
6
7     x += vel.x *dt;
8     y += vel.y *dt;
9 }

```

Facendo ciò, miglioreremo la precisione di previsione della posizione e il sistema perderà energia con il passare del tempo e ciò è fisicamente giusto.

Questo metodo di integrazione è il più semplice da implementare, in quanto è un diretto impiego delle formule della fisica, e il piccolo errore di approssimazione che va a crearsi spesso non viene rilevato dall'occhio dell'utente finale, pertanto è stato scelto come metodo di integrazione per l'implementazione del progetto di questa tesi.

Tuttavia, faremo ora un rapido *excursus* su altri due metodi di integrazione numerica più precisi e utilizzati per programmi in cui la simulazione della fisica deve essere pressoché indistinguibile da quella reale.

3.2.2 Verlet

Il metodo di Integrazione di Verlet venne usato per la prima volta nel 1791 da Jean Delambre, ma fu riscoperto molte altre volte, più recentemente da Loup Verlet negli anni 60 del novecento.

Tra gli esempi che possono essere fatti sull'utilizzo di questo metodo, di sicuro il più importante risulta essere la sua applicazione, a cura di P. H. Cowell e A. C. Crommelin nel 1909, per il calcolo dell'orbita della Cometa di Halley.



Figura 3.8: Cometa di Halley

Al posto di memorizzare la posizione e la velocità degli oggetti, come nel metodo di Integrazione di Eulero, ciò che viene memorizzato è la posizione corrente e quella precedente.

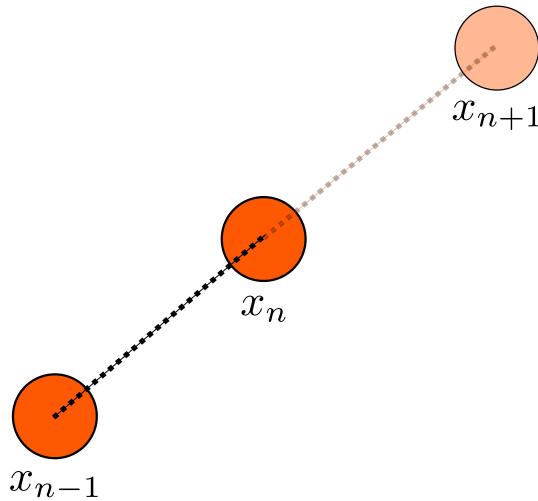


Figura 3.9: Integrazione di Verlet

Per quanto riguarda la velocità dell'oggetto, essa verrà stimata attraverso le due posizioni memorizzate.

Per arrivare a una definizione formale, partiamo dalle equazioni del metodo di Eulero Semi-Implicito:

$$\begin{cases} v_{n+1} = v_n + a\Delta t \\ x_{n+1} = x_n + v_{n+1}\Delta t \end{cases} \quad (3.4) \quad (3.5)$$

Spostiamo la (3.5) a un istante precedente:

$$x_n = x_{n-1} + v_n\Delta t \quad (3.6)$$

Esplicitiamo la velocità v_n

$$v_n = \frac{x_n - x_{n-1}}{\Delta t} \quad (3.7)$$

Ora sostituiamo la (3.7) nella (3.5):

$$x_{n+1} = x_n + (v_n + a\Delta t)\Delta t = x_n + v_n\Delta t + a\Delta t^2 \quad (3.8)$$

E infine sostituiamo la (3.7) nella (3.8):

$$x_{n+1} = x_n + \left(\frac{x_n - x_{n-1}}{\Delta t} \right) \Delta t + a\Delta t^2 \quad (3.9)$$

Semplificando avremo:

$$x_{n+1} = x_n + (x_n - x_{n-1}) + a\Delta t^2 \quad (3.10)$$

3.2.3 Runge-Kutta

Il Metodo Runge-Kutta è una delle tecniche di integrazione migliori per approssimare l'equazione del moto. Essa è usato per simulare la dinamica dei reattori nucleari, per simulazioni climatiche e anche per il calcolo delle orbite di pianeti e asteroidi. Infatti, tra gli usi più importanti di questo metodo, dobbiamo sicuramente parlare del suo utilizzo nel programma Apollo 11. Nel 1969 gli scienziati della NASA, per calcolare la traiettoria della navicella spaziale, usarono questo metodo, utilizzando i dati sulla posizione, sulla velocità della navicella stessa e sull'attrazione gravitazionale della Terra e della Luna(3.10).

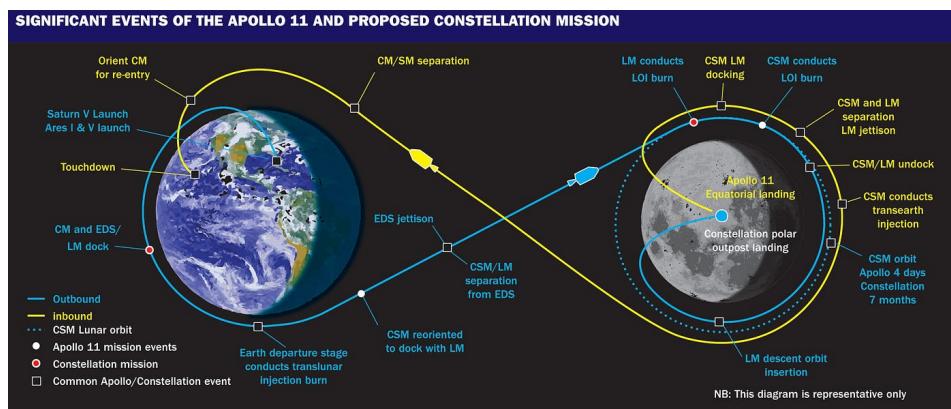


Figura 3.10: Traiettoria dell'Apollo 11, The Great Indian Derby, Medium.com

Ora accenneremo brevemente il suo funzionamento che calcola la pendenza della curva quattro volte ad intervallo, a differenza del metodo di Eulero che calcola la pendenza della curva una sola volta nel lasso di tempo Δt , e userà una media pesata delle stesse per prevedere la posizione successiva dell'oggetto.

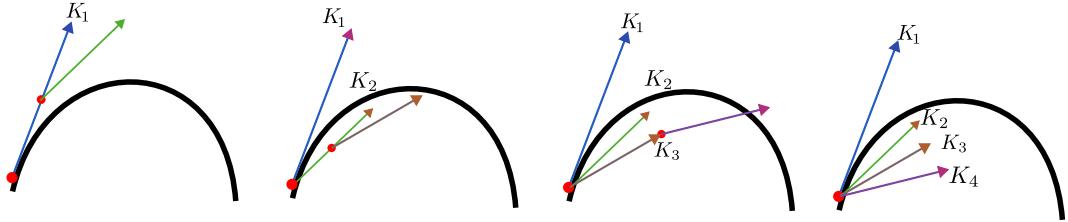


Figura 3.11: Metodo Runge-Kutta

Queste quattro pendenze sono comunemente chiamate K_1 , K_2 , K_3 e K_4 . La prima delle quattro è semplicemente il metodo di Eulero, tuttavia a metà della retta tangente verrà ricalcolata la pendenza, ottenendo K_2 .

Una volta traslata K_2 nel punto della prima retta, useremo lo stesso metodo per calcolare K_3 .

Traslata anche K_3 nel punto del primo calcolo della pendenza, la ricalcoleremo in corrispondenza della fine della retta K_3 , ottenendo così K_4 , che sposteremo infine come le altre rette.

Dunque, l'ultimo passo sarà quello di calcolare la nuova posizione x_{n+1} median-
do tutti i K_i trovati, seguendo la formula sottostante:

$$x_{n+1} = x_n + \frac{1}{6} (K_1 + 2K_2 + 2K_3 + K_4) \Delta t \quad (3.11)$$

Tuttavia, il problema principale di questo metodo di integrazione risiede nel fatto che è computazionalmente oneroso e quindi non è ottimale per la fisica di un videogioco, in cui tutto deve muoversi in maniera fluida e rapida, anche a costo di una più debole rigorosità fisica.

3.3 Oggetti di Gioco

Per implementare la creazione e la gestione dei corpi rigidi rettangolari nel progetto, si è ricorsi al concetto di eredità che fa parte della programmazione ad oggetti. Quindi si è costruita una classe base *GameObject* che ha il compito di definire alcune variabili e alcune funzioni che sono in comune tra le classi derivate *StaticObject* e *DynamicObject*, come ad esempio la posizione, le dimensioni, la massa, il calcolo dei vertici e anche il disegno su schermo. Qui di seguito verrà riportato il file header della classe *GameObject*, preso direttamente dal progetto.

```

1 #include <SDL_render.h>
2 #include <array>
3
4 #include "Vector2.h"
5 #include "AABB.h"
6
7 class GameObject

```

```

8  {
9  public:
10 GameObject(const char *t_textureSheet, SDL_Renderer *t_renderer,
11             float t_xpos, float t_ypos, float t_width,
12             float t_height, float t_mass, float t_theta, float
13             t_restitution, float t_invMass, SDL_Color t_border_color);
14 virtual ~GameObject() {}
15
16 // SETTERS
17 void SetCenterPosition(float t_xpos, float t_ypos)
18 {
19     m_xpos = t_xpos - m_width;
20     m_ypos = t_ypos - m_height;
21 }
22 void SetPosition(Point t_pos)
23 {
24     m_xpos = t_pos.x;
25     m_ypos = t_pos.y;
26 }
27
28 // GETTERS
29 float GetX() { return m_xpos; }
30 float GetY() { return m_ypos; }
31 float GetW() { return m_width; }
32 float GetH() { return m_height; }
33 Point GetPos() { return Point(m_xpos, m_ypos); }
34 Vector2f GetCenter() { return Vector2f(m_xpos + m_width / 2,
35                                         m_ypos + m_height / 2); }
36 float GetMass() { return m_mass; }
37 SDL_FRect *GetRect() { return &m_dstRect; }
38 SDL_Renderer *GetRenderer() { return m_renderer; }
39 float GetMomentOfInertia() { return m_momentOfInertia; }
40 SDL_Texture *GetTex() { return m_objectTexture; }
41 float GetRotation() { return m_theta; }
42 std::array<Vector2f, 4> GetVertices() { return m_vertices; }
43 std::array<Vector2f, 4> GetNormals() { return m_normals; }
44 float GetRestitution() { return m_restitution; }
45 float GetInvMass() { return m_invMass; }
46 AABB GetAABB() { return m_aabb; }
47 bool GetIsStatic() { return m_is_static; }
48
49 // Functions
50 void CalculateVertices();
51 void CalculateAABB();
52 void Render();
53
54 protected:
55 // Physics
56 float m_xpos, m_ypos, m_width, m_height;
57 float m_mass;
58 float m_invMass;
59 float m_momentOfInertia;
60 float m_theta = 0;

```

```
58     AABB m_aabb;
59     float m_restitution;
60     std::array<Vector2f, 4> m_vertices;
61     std::array<Vector2f, 4> m_normals;
62     bool m_is_static;
63
64     // Rendering
65     SDL_Texture *m_objectTexture;
66     SDL_FRect m_dstRect;
67     SDL_Renderer *m_renderer;
68     SDL_Color m_border_color;
69 };
```

In questo oggetto, le funzioni *CalculateVertices()* e *CalculateAABB()* sono di particolare interesse, quindi andremo qui di seguito a trascrivere la loro implementazione.

La funzione *CalculateVertices()* ha lo scopo di calcolare i vertici degli oggetti, che sono di maggiore interesse per corpi dinamici, in quanto oltre ad essere in movimento, hanno la possibilità di ruotare. Tutto ciò è calcolato attraverso la *VectorRotation()* funzione, definita in una classe chiamata *Math*, creata per gestire la maggior parte delle operazioni matematiche complesse.

Qui di seguito riportiamo l'implementazione della funzione *VectorRotation()*, il cui scopo è quello di implementare la rototraslazione, vista nel capitolo 2 di questa tesi.

```
1 Vector2f Math::VectorRotation(const Vector2f &point, const
2     Vector2f &center, float rotation_deg)
3 {
4     float angle = Rad(rotation_deg);
5     float cos_angle = cos(angle);
6     float sin_angle = sin(angle);
7
8     Vector2f centered_point(point.x - center.x, point.y - center.y
9 );
10
11    Vector2f rot_centered_point(cos_angle * centered_point.x -
12        sin_angle * centered_point.y, sin_angle * centered_point.x +
13        cos_angle * centered_point.y);
```

```

10
11     return (rot_centered_point - center);
12 }
```

Ora visioneremo la funzione *CalculateAABB()*, in cui AABB sta per Aligned-Axis Bounding Box, ovvero dei rettangoli allineati agli assi cartesiani che racchiudono gli oggetti, indipendentemente dalla loro rotazione.

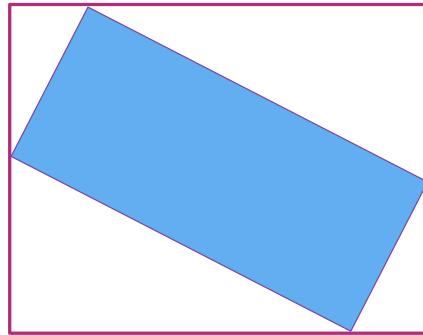


Figura 3.12: AABB

```

1 void GameObject::CalculateAABB()
2 {
3     float minX = INFINITY;
4     float minY = INFINITY;
5     float maxX = -INFINITY;
6     float maxY = -INFINITY;
7
8     for (const auto &v : m_vertices)
9     {
10         minX = std::min(minX, v.x);
11         maxX = std::max(maxX, v.x);
12         minY = std::min(minY, v.y);
13         maxY = std::max(maxY, v.y);
14     }
15
16     m_aabb.x = minX;
17     m_aabb.y = minY;
18     m_aabb.w = maxX - minX;
19     m_aabb.h = maxY - minY;
20 }
```

Come primo oggetto derivato dalla classe *GameObject*, esamineremo ora gli oggetti statici, ovvero quei corpi che non si muovono per tutta la durata del programma e che quindi possono essere gestiti più semplicemente rispetto agli oggetti dinamici. Questi non hanno particolari ridefinizioni, rispetto alla classe base, in quanto oggetti inerti. Qui di seguito abbiamo una loro rappresentazione, in posizioni e angolature arbitrarie:

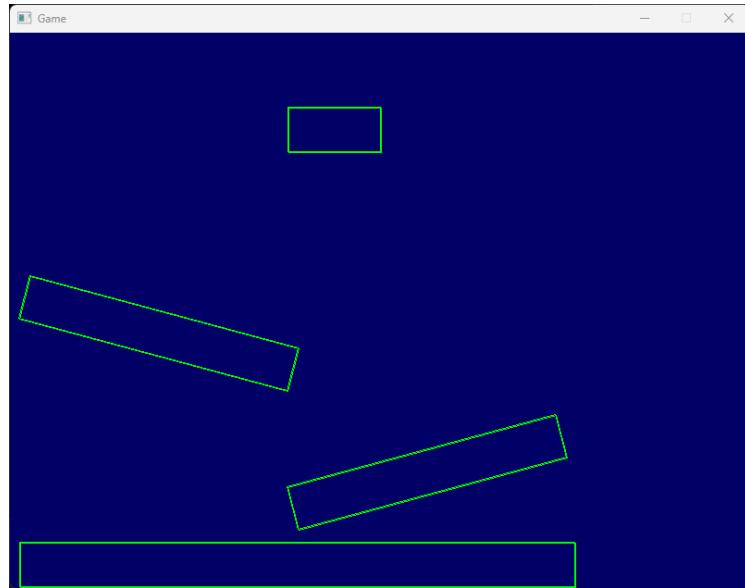


Figura 3.13: AABB

Per quanto riguarda gli oggetti dinamici, essi sono soggetti a delle forze, e di conseguenza possono essere in movimento a differenza degli oggetti statici. Pertanto qui di seguito verrà riportato il file header della classe derivata *DynamicObject*, e in seguito analizzeremo le funzioni più importanti che la caratterizzano:

```

1 #include <SDL_render.h>
2 #include <vector>
3 #include <tuple>
4
5 #include "GameObject.h"
6 #include "Collision.h"
7 #include "Constants.h"
8 #include "Vector2.h"
9
10 class DynamicObject : public GameObject
11 {
12
13 public:
14     DynamicObject(const char *t_textureSheet, SDL_Renderer *
15                 t_renderer, float t_xpos, float t_ypos,
16                 Vector2f t_velocity, float t_width, float t_height, float
17                 t_mass, float t_restitution, SDL_Color t_border_color =
18                 default_color);
19
20     // SETTERS
21     void SetVelocity(Vector2f t_velocity) { m_velocity = t_velocity;
22     }
23     void SetVelocityX(float t_x) { m_velocity.x = t_x; }
24     void SetVelocityY(float t_y) { m_velocity.y = t_y; }
25     void SetMoving(bool t_m) { m_moving = t_m; }
26     void SetAngularVelocity(float t_angularVelocity) {
27         m_angularVelocity = t_angularVelocity; }
```

```

23
24 // GETTERS
25 Vector2f GetVelocity() { return Vector2f(m_velocity.x,
26   m_velocity.y); }
26 bool GetMoving() { return m_moving; }
27 float GetAngularVelocity() { return m_angularVelocity; }
28
29 // Functions
30 void Update(float t_dt, float t_gravity);
31 void MoveTo(Vector2f t_pos);
32 void MoveBy(float t_xpos, float t_ypos);
33 void AddVel(Vector2f t_value) { m_velocity += t_value; }
34
35 private:
36   // Physics
37   Vector2f m_velocity;
38   float m_momentOfInertia;
39   float m_angularAcceleration;
40   float m_angularVelocity;
41   bool m_moving;
42 };

```

Di sicuro la funzione più importante è *Update()*, che gestisce l'avanzamento fisico dell'oggetto dinamico in base alla variabile *dt* che abbiamo definito nella sezione del Ciclo Principale [1].

```

1 void DynamicObject::Update(float t_dt, float t_gravity)
2 {
3   // Euler's Integration
4   m_xpos += m_velocity.x * t_dt;
5   m_ypos += m_velocity.y * t_dt;
6
7   m_velocity += Vector2f(0, t_gravity * t_dt);
8   m_theta += m_angularVelocity * t_dt;
9
10  // New Position Rect
11  Utils::SetRectSizeAndPosition(m_dstRect, m_xpos, m_ypos, m_width
12    , m_height);
13
14  // Calculate New Vertices
15  CalculateVertices();
16  // Calculate New AABB
17  CalculateAABB();
18 }

```

In questa funzione viene applicata l'Integrazione di Eulero, vista nella sezione dell'Integrazione Numerica [2], aggiornando la posizione, la velocità lineare e quella angolare.

Una volta fatto ciò vengono ricalcolati i vertici dell'oggetto e il suo Bounding Box. Quindi, per gli oggetti dinamici, queste operazioni vengono effettuate ad ogni frame, mentre per gli oggetti statici verranno effettuate soltanto all'avvio del programma, in quanto la loro posizione e la loro angolazione non potrà variare.

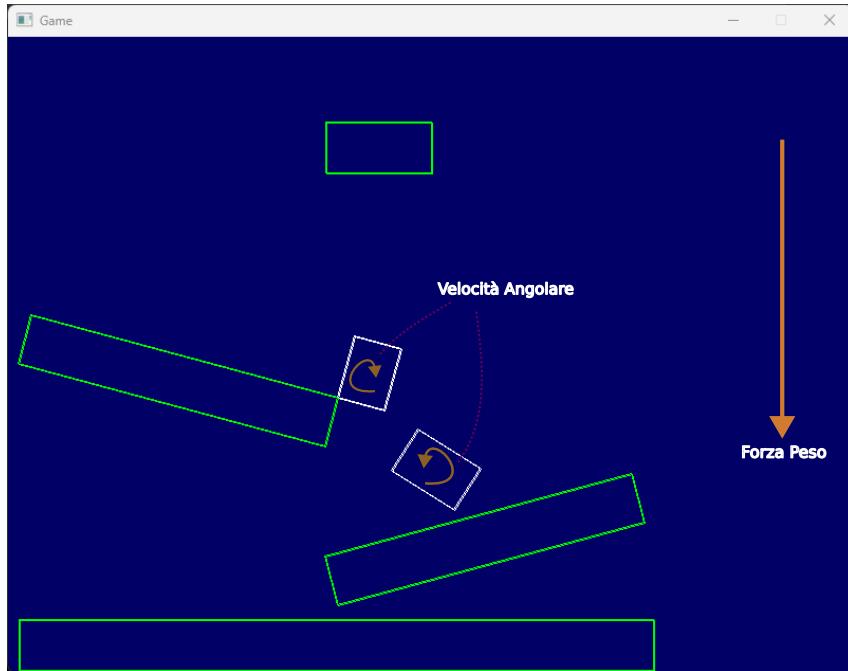


Figura 3.14: Oggetti Dinamici nel Progetto

3.4 Mondo di Gioco

Per gestire al meglio la componente fisica del programma, si è ricorsi ad una classe di controllo, chiamata *World*, che è stata usata come componente della classe *Game*. Qui di seguito riporteremo il file header della classe *World* e ne andremo poi a descrivere le principali funzioni che sono state implementate.

```

1 #include <vector>
2
3 #include "GameObject.h"
4 #include "DynamicObject.h"
5 #include "StaticObject.h"
6 #include "Vector2.h"
7 #include "Collision.h"
8
9 class World
10 {
11 private:
12     float m_G = 9.81f;
13     std::vector<GameObject *> m_objects;
14
15 public:
16     World(float t_gravity);
17     ~World();
18
19     void AddGameObject(GameObject *t_obj);
20
21     void DeleteObject(GameObject *t_obj);

```

```

22 std::vector<GameObject *> &GetGameObjects();
23
24 void Update(float t_dt);
25
26 void CheckCollisions(int it, int t_iterationNumber);
27
28 void SolveDynVsDynCollisionRotation(DynamicObject *t_objA,
29                                     DynamicObject *t_objB, CollisionType t_check);
30
31 void SolveDynVsStaticCollisionRotation(DynamicObject *t_objA,
32                                         StaticObject *t_objB, CollisionType t_check);
33
34 std::array<Vector2f, 2> m_contactList;
35 };

```

In particolare possiamo notare alcune funzioni che sono di un certo interesse. La prima è di sicuro la funzione *Update()*, che verrà invocata dall'oggetto *Game* a seguito della chiamata della sua funzione *Update()*, come abbiamo visto nella sezione [1].

```

1 void World::Update(float t_dt)
2 {
3     for (auto obj : m_objects)
4     {
5         if (Utils::isInstanceOf(obj, typeid(DynamicObject)))
6         {
7             DynamicObject *dynamic_obj = dynamic_cast<DynamicObject *>(obj);
8             dynamic_obj->Update(t_dt, m_G, t_iterationNumber);
9         }
10    }
11    CheckCollisions();
12 }

```

Il corpo di questa funzione è un ciclo che scorre su tutti gli elementi del vettore degli oggetti, invocando le relative funzioni *Update()* degli oggetti dinamici e verificando le eventuali collisioni avvenute.

Focalizziamoci per l'appunto sulla funzione di verifica delle collisioni, ovvero la funzione *CheckCollisions()*. Essa è composta da un doppio ciclo che scorre sullo stesso vettore di oggetti, in modo tale da verificare tutte le coppie possibili.

```

1 void World::CheckCollisions()
2 {
3     for (int i = 0; i < m_objects.size(); ++i)
4     {
5         for (int j = i + 1; j < m_objects.size(); ++j)
6         {
7             CollisionType check = Collision::IntersectSAT(m_objects[i],
8 m_objects[j]);
9             if (check.areColliding)
10            {

```

```
10     switch (check.mode)
11     {
12     case DynVsDyn:
13     {
14         DynamicBody *objA = dynamic_cast<DynamicBody *>(
15             m_objects[i]);
16         DynamicBody *objB = dynamic_cast<DynamicBody *>(
17             m_objects[j]);
18
18         // Move the Rectangles by the depth collision
19         objA->MoveBy(-check.CollidingAxis.x * check.depth / 2, -
20             check.CollidingAxis.y * check.depth / 2);
21         objB->MoveBy(check.CollidingAxis.x * check.depth / 2,
22             check.CollidingAxis.y * check.depth / 2);
23
24         // Resolve Collision
25         SolveDynVsDynCollisionRotation(objA, objB, check);
26         break;
27     }
28     case DynVsStatic:
29     {
30         DynamicBody *objA = dynamic_cast<DynamicBody *>(
31             m_objects[i]);
32         StaticBody *objB = dynamic_cast<StaticBody *>(m_objects[
33             j]);
34
35         // Move the Rectangles by the depth collision
36         objA->MoveBy(-check.CollidingAxis.x * check.depth, -
37             check.CollidingAxis.y * check.depth);
38
39         // Resolve Collision
40         SolveDynVsStaticCollisionRotation(objA, objB, check);
41         break;
42     }
43     case StaticVsDyn:
44     {
45         StaticBody *objA = dynamic_cast<StaticBody *>(m_objects[
46             i]);
47         DynamicBody *objB = dynamic_cast<DynamicBody *>(
48             m_objects[j]);
49
50         // Move the Rectangles by the depth collision
51         objB->MoveBy(check.CollidingAxis.x * check.depth, check.
52             CollidingAxis.y * check.depth);
53
54         // Resolve Collision
55         SolveDynVsStaticCollisionRotation(objB, objA, check);
56         break;
57     }
58 }
```

```

53     }
54 }
55 }
```

Come possiamo vedere è stata utilizzata una classe di supporto per la gestione dei calcoli relativi all'algoritmo SAT, ovvero la classe Collision. Questa classe ha il compito di verificare l'avvenuta collisione tra due oggetti, il tipo di collisione che è avvenuta (dinamico con dinamico, statico con dinamico), e di calcolare alcuni parametri fondamentali per la successiva fase di risoluzione delle collisioni, come ad esempio il vettore di profondità di penetrazione e i punti di contatto tra gli oggetti. La prima cosa che viene fatta, una volta rilevata una collisione, è separare i due oggetti grazie ai parametri calcolati, come visto nella sezione [4.4.1], e in seguito invocare la funzione di risoluzione.

Per quanto riguarda la risoluzione delle collisioni, questo aspetto è stato già ben approfondito nel capitolo [2], in cui venivano identificate due fasi, una in cui veniva calcolato l'impulso di risposta all'urto e una in cui veniva applicato questo impulso, sia per quanto riguarda la velocità lineare e sia per quanto riguarda la velocità angolare.

3.5 Miglioramenti

3.5.1 Iterazioni

Un primo miglioramento ad essere stato effettuato, riguarda la precisione con la quale le proprietà dinamiche degli oggetti vengono gestite e ciò è stato fatto aumentando il numero di volte in cui viene aggiornata la fisica del mondo di gioco nello stesso frame.

Nella funzione *Update()* dell'oggetto World, viene passato un nuovo parametro, ovvero il numero di iterazioni da effettuare ad ogni frame.

```

1 void Game::Update(float dt)
2 {
3     m_world->Update(dt, 60);
4 }
```

Una volta fatto ciò, la funzione *Update()* della classe *World* invocherà tutte le funzioni *Update()* dei vari oggetti dinamici, dove il parametro *dt* verrà diviso per il numero di iterazioni.

```

1 void DynamicObject::Update(float t_dt, float t_gravity, int
2                             t_iterations)
3 {
4     t_dt /= t_iterations;
5     // code ...
```

Questo espediente migliorerà significativamente la precisione della simulazione fisica degli oggetti, garantendo una risposta agli urti più fedele alla realtà fisica.

3.5.2 Vector Vs Array

In principio il programma era in grado di gestire soltanto fino a dodici oggetti dinamici, causando un eccessivo calo di frame-rate all'aggiunta di nuovi. Per questo un miglioramento significativo è stato quello di sostituire tutte le strutture dati, di cui si era a conoscenza della dimensione, da *std::vector* a *std::array*. Questo perché la struttura dati *std::vector* è di natura dinamica, pertanto non ha bisogno di una lunghezza prefissata, e ciò implica un maggiore utilizzo di risorse. Viceversa la struttura dati *std::array* possiede una lunghezza prefissata e ha un ottimo utilizzo della memoria. Questo semplice miglioramento ha portato il programma a poter gestire fino a trenta oggetti dinamici.

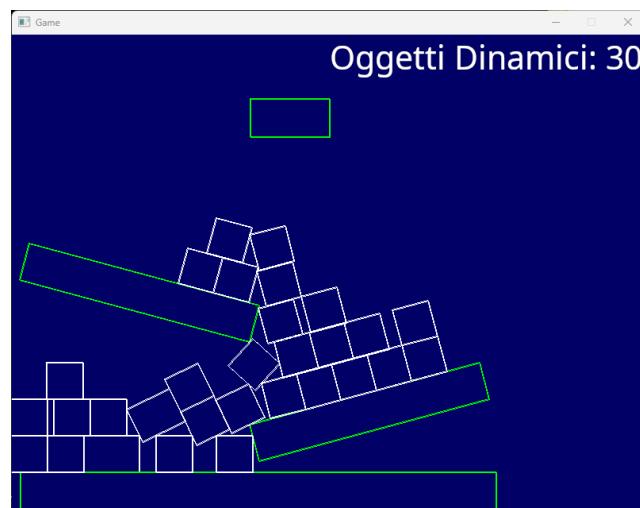


Figura 3.15: Oggetti Dinamici nel Progetto V.1

3.5.3 Controlli Preliminari al Rilevamento delle Collisioni

Di sicuro tra le operazioni più onerose in termini di risorse computazionali vi sono quelle relative al rilevamento delle collisioni, essendo le stesse ripetute numerose volte nello stesso frame. Per questo motivo è possibile migliorare ulteriormente il codice inserendo delle verifiche sugli oggetti che saranno sottoposti al controllo di collisione, per evitare di procedere all'analisi di coppie di oggetti che sicuramente non sono in uno stato di collisione.

```

1 void World::CheckCollisions()
2 {
3     for (int i = 0; i < m_objects.size(); ++i)
4     {
5         for (int j = i + 1; j < m_objects.size(); ++j)
6         {
7             // Verifiche Preliminari
8             // Rilevamento delle collisioni
9         }
10    }
11 }
```

La prima verifica fatta è stata quella di appurare se i due oggetti di cui si vuole dimostrare l'avvenuta collisione siano entrambi statici.

```

1 if (Utils::isInstanceOf(m_objects[i], typeid(StaticBody)) && Utils
2   ::isInstanceOf(m_objects[j], typeid(StaticBody)))
3 {
4   continue;
5 }
```

Nel caso in cui ciò si verificasse, queste righe di codice porterebbero il ciclo a saltare la coppia di oggetti corrente, dato che due oggetti statici non potrebbero mai collidere.

La seconda verifica che è stata effettuata riguarda gli Axis Aligned Bounding Box, che abbiamo introdotto in precedenza. Il loro coinvolgimento nasce dal concetto che se i due AABB dei due oggetti in esame non collidono, allora ciò vuol dire che è impossibile che i due oggetti possano essere in collisione.

```

1 if (!Collision::IntersectAABB(m_objects[i]->GetAABB(), m_objects[j]
2   ]->GetAABB()))
3 {
4   continue;
5 }
```

Queste migliorie hanno portato il programma a poter essere in grado di gestire fino a settanta oggetti dinamici in movimento, come mostrato in figura [3.16]:

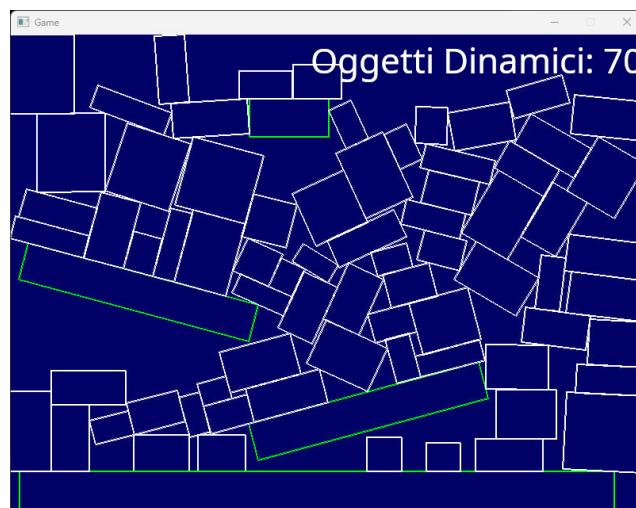


Figura 3.16: Oggetti Dinamici nel Progetto V.2

Nella figura si è anche dimostrata l'efficienza del progetto nella gestione di oggetti rettangolari di diversa forma e di diversa conformazione fisica.

3.5.4 Quad Tree

Infine, un ulteriore miglioramento che può essere effettuato per ottimizzare il programma, è l'implementazione di un Quad Tree.

Il Quad Tree è una struttura dati spaziale che viene utilizzata per la gestione di grandi quantità di oggetti in uno spazio bidimensionale. L'obiettivo del Quad Tree è quello di organizzare gli oggetti in un albero a quattro figli, in modo da poter effettuare ricerche e operazioni di rilevamento delle collisioni in modo efficiente.

Il Quad Tree divide lo spazio in quadranti di dimensioni uguali, ciascuno dei quali rappresenta un nodo dell'albero. La divisione continua finché ogni quadrante contiene un numero massimo di oggetti o fino a quando la dimensione del quadrante raggiunge una soglia minima.

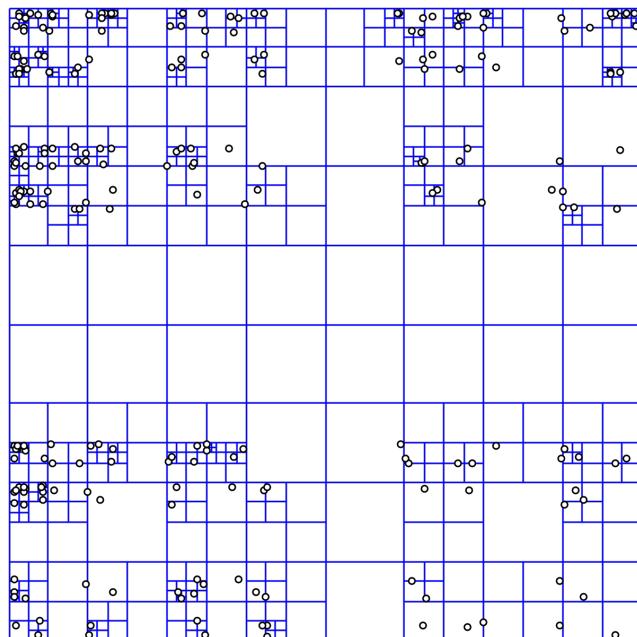


Figura 3.17: Quad Tree

L'uso del Quad Tree può migliorare significativamente le prestazioni della verifica delle collisioni in giochi e applicazioni che coinvolgono un gran numero di oggetti, in quanto consente di limitare la ricerca solo alle aree in cui gli oggetti sono presenti e di evitare ricerche inutili in aree vuote.

Inoltre, il Quad Tree può essere utilizzato per altre operazioni spaziali, come il rilevamento di oggetti vicini o la ricerca di oggetti all'interno di un'area specifica. Tuttavia, la creazione e la gestione di un Quad Tree richiedono una certa quantità di elaborazione iniziale e di memoria, quindi è importante trovare un equilibrio tra le prestazioni e la complessità dell'algoritmo.

CAPITOLO 4

Conclusione

Nel corso di questa tesi, abbiamo intrapreso un'esplorazione dei metodi avanzati per la simulazione della dinamica dei corpi rigidi rettangolari, ponendo un focus particolare sulla verifica e sulla risoluzione delle collisioni in un contesto bidimensionale (2D).

Abbiamo delineato in modo esaustivo i principali approcci per la verifica delle collisioni, concentrando in particolar modo sull'ampiamente utilizzato algoritmo del Separating Axis Theorem (SAT). Questo algoritmo, attraverso l'analisi delle proiezioni degli oggetti su assi specifici, è in grado di rilevare con precisione se due corpi rigidi rettangolari stiano intersecandosi o meno. Attraverso il calcolo dei vettori normalizzati noti come assi di separazione, l'algoritmo SAT offre una soluzione affidabile e veloce per identificare gli impatti tra oggetti, fondamentale nell'ambito delle simulazioni fisiche.

In seguito, ci siamo dedicati con rigore all'esplorazione della risoluzione delle collisioni rilevate. Abbiamo esaminato il processo di calcolo dell'impulso risultante generato dall'urto e la sua successiva applicazione su entrambi gli oggetti coinvolti. Questo calcolo, che si basa su parametri fisici quali la massa, la velocità e la restituzione elastica, assicura che gli oggetti coinvolti si separino con fluidità e realismo dopo l'urto, evitando qualsiasi penetrazione reciproca.

In seguito siamo poi approdati alla presentazione di un progetto pratico, creato appositamente per questa tesi. In quest'ambito, abbiamo applicato concretamente le competenze acquisite, sviluppando una simulazione 2D che mette in evidenza le dinamiche di collisione. Attraverso la creazione di scenari, la modellazione di oggetti con proprietà uniche e la realizzazione di un ambiente interattivo, il progetto ha sottolineato l'efficacia e l'importanza cruciale degli algoritmi di verifica e risoluzione delle collisioni nell'ambito delle simulazioni fisiche di corpi rigidi.

In conclusione, il percorso intrapreso in questa tesi ha offerto un'approfondita analisi sulla simulazione della dinamica dei corpi rigidi rettangolari nel contesto 2D, esplorando a fondo i meccanismi di verifica e risoluzione delle collisioni. Il proget-

to realizzato ha dimostrato in modo tangibile la validità di tali teorie, trasformando il sapere teorico in un'applicazione pratica di grande significato nell'ambito delle simulazioni fisiche.

Achille Cannavale
matr. 0058721

Elenco delle figure

1.1	Hitbox SuperMario	3
1.2	Hitbox Street Fighter	3
1.3	Hitbox Zelda	4
1.4	Circle Collision Detection	5
1.5	AABB Collision Detection	5
1.6	Convessità	6
1.7	Trasformazione da figura concava a serie di figure convesse	6
1.8	Differenza di Minkowski	7
1.9	Rotazione	8
1.10	Oggetti Sovrapposti	9
1.11	Proiezioni Lungo le Normali	9
1.12	Punti di Contatto	12
2.1	Danno vitale in Zelda	14
2.2	Risposta agli urti in GTA	14
2.3	Rette verso i punti di contatto	16
3.1	Logo Visual Studio	19
3.2	Logo SDL	19
3.3	Esempi di giochi fatti con SDL	20
3.4	Esempi di giochi fatti con box2D	20
3.5	Mancato Rilevamento della Collisione	21
3.6	Super Mario 64	23
3.7	Grafico Integrazione Esplicita di Eulero	24
3.8	Cometa di Halley	26
3.9	Integrazione di Verlet	26
3.10	Traiettoria dell'Apollo 11, The Great Indian Derby, Medium.com	27
3.11	Metodo Runge-Kutta	28
3.12	AABB	31
3.13	AABB	32
3.14	Oggetti Dinamici nel Progetto	34
3.15	Oggetti Dinamici nel Progetto V.1	38
3.16	Oggetti Dinamici nel Progetto V.2	39
3.17	Quad Tree	40

Bibliografia

- [1] Cj Birch. Unknown horizons. URL: <https://unknown-horizons.org/>. 19
- [2] Capcom. Street fighter. URL: <https://mario.nintendo.com/>. 3
- [3] Erin Catto. Box2d. URL: <https://box2d.org/>. 20
- [4] Two-Bit Coding. Let's make a physics engine. URL: <https://www.youtube.com/playlist?list=PLSlpr6o9vURwq3oxVZSimY8iC-cdd3kIs>. 2
- [5] Microsoft Corporation. Visual studio 2022. URL: <https://visualstudio.microsoft.com/it/>. 19
- [6] Fingersoft. Hill climb racing. URL: <https://fingersoft.com/games/hill-climb-racing/>. 20
- [7] Chris Hecker. Physics, part 3: Collision response. page 5, march 1997. URL: <http://www.chrishecker.com/images/e/e7/Gdmphys3.pdf>. 16
- [8] Sam Lantinga. Sdl (simple directmedia layer). URL: <https://www.libsdl.org/license.php>. 19
- [9] Rovio Mobile. Angry birds. URL: <https://www.angrybirds.com/>. 20
- [10] Nintendo. The legend of zelda: a link to the past. URL: <https://www.nintendo.co.uk/Games/Super-Nintendo/The-Legend-of-Zelda-A-Link-to-the-Past-841179.html>. 4
- [11] Nintendo. Super mario. URL: <https://mario.nintendo.com/>. 3
- [12] Nintendo. Super mario 64. URL: <https://www.nintendo.it/Giochi/Nintendo-64/Super-Mario-64-269745.html>. 23
- [13] Playdead. Limbo. URL: <https://playdead.com/games/limbo/>. 20
- [14] Rockstar. Gta 5. URL: <https://www.rockstargames.com/it/gta-v>. 14
- [15] OpenTTD Team. Openttd. URL: <https://www.openttd.org/>. 19
- [16] The SuperTux Team. Supertux. URL: <https://www.supertux.org/>. 19